SAVCBS 2001 Proceedings
Specification and Verification of Component-Based Systems
Workshop at OOPSLA 2001

Dimitra Giannakopoulou, Gary T. Leavens, and Murali Sitaraman (editors)

TR #01-09a
October 14, 2001, revised November 6, 2001

Keywords: Specification, verification, component-based systems.

2000 CR Categories: D.1.m [*Programming Techniques*] Miscellaneous — component-based programming, reflection; D.2.1 [*Software Engineering*] Requirements/Specifications — languages, methodology, theory, tools; D.2.4 [*Software Engineering*] Software/Program Verification — assertion checkers, class invariants, correctness proofs, formal methods, model checking, programming by contract, reliability, validation; D.2.5 [*Software Engineering*] Testing and Debugging — testing tools; D.2.11 [*Software Engineering*] Software Architecture — languages; D.2.m [*Software Engineering*] Miscellaneous — component-based systems, reusable software; D.3.1 [*Programming Languages*] Formal Definitions and Theory — semantics; D.3.3 [*Programming Languages*] Language Constructs and Features — data types and structures; F.3.1 [*Logics and Meaning of Programs*] Specifying and verifying and reasoning about programs — assertions, invariants, logics of programs, pre- and post-conditions, specification techniques; F.3.m [*Logics and Meaning of Programs*] Miscellaneous — reasoning about performance.

Each paper's copyright is held by its author.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Table of Contents

Other Accepted Papers

Preface

The goal of this workshop was to explore how formal (i.e., mathematical) techniques can be or should be used to establish a suitable foundation for specification and verification of component-based systems. Component-based systems are a growing concern for the object-oriented community. Specification and reasoning techniques are urgently needed to permit composition of systems from components, for which source code is unavailable.

We wanted to bring together researchers and practitioners in the areas of component-based software and formal methods, to address the specification and verification problems. Several representatives from Microsoft research attended the workshop, and presented their approach to specification and verification in the context of Microsoft products. However, it was generally agreed that a lot remains to be done to address the needs of industry. On the other hand, papers on testing, run-time checking of assertions, and the use of message sequence charts addressed more practical concerns. Another goal was to focus more of the effort in formal methods on component-based systems; time will tell if we have contributed to realizing this goal.

The main expected result of the meeting would be an outline of collaborative research topics and a list of areas for further exploration. Some of these ideas were presented in our OOPSLA poster.

The papers at the workshop and those included in the proceedings were selected from papers submitted by researchers worldwide. Due to time limitations at the workshop, only a few papers could be presented

The discussion at the workshop itself was quite interesting. All agreed that compositional, modular reasoning is a necessary goal in this area. We discussed several strategies for making reasoning more tractable, including proving less, checking parts of a proof at run-time (as in run-time assertion checking), decomposing proofs by using stronger specifications, and writing components in ways that make proofs easier (e.g., by limiting the use of pointers). We also discussed ways to add value to specifications, including providing support for testing and run-time assertion checking. Barnett and Schulte pointed out that in one case at Microsoft, a specification was “orders of magnitude” smaller than the code it specified. We discussed ways to extend type systems, to incorporate architectural constraints and message sequence information. Several of the techniques discussed focused on component interaction at interface boundaries, which is helpful in reasoning about compositions.

We also identified several areas that seem ripe for future work. One is putting together trace-based concurrency reasoning with reasoning about data values. Another is how to reason about performance (i.e., time and space behavior); one paper at the workshop discussed this, but there is more to be done, and this kind of reasoning is important for embedded systems. Another area is how to make reasoning easier. One direction for making reasoning easier is finding limits on programs that have a big impact on ease of reasoning. There was a lot of discussion of the idea of Weide and Heym to encapsulate references (pointers) in components, so that all variables in a program are not general references. We also talked about finding the right abstractions for reasoning about compositions. And we discussed extending type systems to incorporate more specification information, while still allowing them to be decidable and efficiently checkable.

The workshop was organized by Dimitra Giannakopoulou (NASA Ames/RIACS), Gary T. Leavens (Iowa State University), and Murali Sitaraman (Clemson University). The program committee that selected papers consisted of the organizers and Betty H. C. Cheng (Michigan State University), Steve Edwards (Virginia Tech), K. Rustan M. Leino (Compaq Systems Research Center), and Markus Lumpe (Iowa State University). We thank the organizers of OOPSLA 2001 for hosting the workshop.

Testing Components

Neelam Soundarajan and Benjamin Tyler
Computer and Information Science
Ohio State University, Columbus, OH 43210
e-mail: {neelam,tyler}@cis.ohio-state.edu

Abstract

Our goal is to investigate specification-based approaches to testing OO components. That is, given a class C and its specification, how do we test C to see if it meets its specification? Two important requirements that we impose on the testing approach are that it must not require access to the source code of the class under test; and that it should enable us to deal incrementally with derived classes, including derived classes that exploit polymorphism to extend the behavior of the base class. In this paper, we report on our work towards developing such a testing approach.

1. INTRODUCTION

Our goal is to investigate specification-based approaches to testing OO components. Suppose we are given an implementation of a class C and the specifications of its methods in the form of pre- and post-conditions (and possibly a class invariant). How do we test the implementation of C to see if it meets its specifications? We are not specifically interested in the question of how to choose a broad enough range of test cases [12] although that would, of course, have to be an important part of a complete testing methodology for OO systems. Rather, we want to develop a general approach that can be used to test that C meets its specifications. Once we do this, we should be able to combine it with an appropriate methodology for choosing test cases.

We impose two important requirements on the testing approach. First, as far as possible it must not require access to the source code of the class under test. This is important if we are to be able to test not just components we designed and implemented but components that we may have purchased from a software vendor. Second, the testing approach should enable us to deal incrementally with derived classes, including derived classes that exploit polymorphism to extend the behavior of the base class. Much of the power of the OO approach derives from the ability to develop systems incrementally, using inheritance to implement derived classes that extend the behavior of their base classes. To best exploit this incremental nature of OO, our approach to reasoning about and testing the behavior of such classes should also be correspondingly incremental. In this paper, we report on our work towards developing such a testing approach.

In the next section, we provide a more detailed statement of the problem. In Section 3, we outline how the behavior of derived classes that use polymorphism to enrich base class behavior may be established in a verification system. In Section 4 we show how our testing approach can work with the kind of specifications used in the verification system. In Section 5, we briefly consider some problems related to testing classes that have components that may themselves exploit polymorphism.

2. BACKGROUND AND MOTIVATION

An important tenet of the OO approach is *abstraction*. Thus a client of a class should have an abstract view of the class, rather than thinking in terms of the concrete structure, i.e., the member variables, of the class. Correspondingly, the specification of a class C usually consists of pre- and post-conditions of the methods of C, in terms of an abstract or conceptual model of C. But abstraction causes an important difficulty [2] for specification-based testing[1]. When testing, we have to analyze how the values of the member variables of the class change as various member operations are invoked, so we have the problem of matching these values to the abstract specification. Inheritance exacerbates the problem since the set of variables and operations in the derived class is a complex mix of items defined in the base and derived classes.

Given this, in our approach to testing, we work with *concrete* specifications for the classes. This is not to suggest that abstract specifications are not important. It is just that when considering and testing the behavior of the *implementation* of C, the concrete state of the class has to play an important role since that is what the implementation works with. Similarly, when considering the behavior of the derived class, we (the designer of the derived class as well as the tester) must keep in mind the concrete state of both the base and derived classes. When dealing with the behavior of some *client* code *cc* that uses C, we should of course not think in terms of the concrete state of C; later in the paper, we will see how abstract specifications (of C) enter the picture when considering testing of *cc*.

[1]In this paper, by 'component', we will generally mean 'class' as in a typical OO language.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA 2001 Workshop on Specification and Verification of Component Based Systems Oct. 2001 Tampa, FL, USA
Copyright 2001 N. Soundarajan and B. Tyler.

The concrete specification of C characterizes the behavior of each method of C in terms of pre- and post-conditions that are assertions on the member variables of C. The specification may also include an invariant, although for simplicity we will usually ignore it in our discussion. Our goal is to create a *testing class* TC corresponding to C that will allow us to test the class C against this concrete specification. We note that the word 'testing' in the title of the paper may be considered a verb since we are interested in testing the behavior of C; it may also be considered an adjective since our approach to testing is to construct the testing component TC.

In [16], we had suggested the following simple approach to the construction of TC: For each method m() of C, include a corresponding test method test_m() in TC that will invoke m(). To do this we need an instance object, call it tc, of type C. More precisely tc is a member variable of TC of type C, and its value will be (a reference to) an object of type C. Let us assume that the constructor of TC has initialized tc to such a value. We can now write test_m() to simply consist of a call to m() followed by an assert statement in which we require that the post-condition *post.m* is satisfied. Now m() is required to work, that is ensure that its post-condition is satisfied when it finishes, only if its pre-condition was satisfied at the time of the call to it. Thus a natural definition of the body of test_m() is:

```
if ( pre.m ) {  tc.m(); assert( post.m );  }
```

Since the object here is tc, references to a variable x of C in *pre.m*, *post.m* should be replaced by to tc.x. Are these references legal? Member variables are typically protected, and accessible only within C (and derived classes). In *Java* [1], we could put TC in the same *package* with C and give data members package scope. It is not clear how to address this in other languages; we had suggested in [16] that it may be useful to introduce the privileged notion of *test class* into the language, with the methods of the test class being given access to the members of the class[2]. Another point is that *post.m* may contain references to the value of tc.x at the time of the *call*. So we need to *save* this value in, say, xold, and replace (in *post.m*) x@pre by xold; in general, we need to use a *cloning* operation [11] for this purpose. Yet another issue has to do with the form of the assertions. Given that we want the assertions to be machine checkable, they have to have a somewhat restricted form [5, 8]. One possibility [11] would be to require that the assertions be legal boolean expressions allowed by the language. Here we will just assume that simple assertions, including quantifiers over finite domains, are allowed.

In this paper, we want to focus on a different issue. Suppose again that D is a derived class of C. Some methods may be defined (or redefined) in D while others may be inherited from C. Most importantly, even some of the inherited methods may exhibit behavior that is different from their behavior in the base class because of calls to methods that are redefined in D. Following the design patterns literature [6], we will call such methods *template* methods, the methods they invoke that may be redefined in D being called *hook* methods. Let $t()$ be a template method of C, and $h()$ a hook method that $t()$ invokes. As we just noted, redefining $h()$ in the derived class enriches also the behavior of $t()$. When reasoning about $t()$ in the base class, we would have appealed to the base class specification of $h()$ to account for the effects of the calls that $t()$ makes to $h()$. In order to be sure that the conclusions we have reached about the behavior of $t()$ apply also to its behavior in the derived class despite the redefinition of $h()$, we have to require that the redefined $h()$ satisfies its base class specification; this requirement is the essence of *behavioral subtyping* [10, 4]. But ensuring that $t()$ continues to behave in a way that is consistent with it its base class specification is only part of our concern. The reason that we redefined $h()$ in the derived class was to thereby enrich (as we will see even in the simple example later in the paper) the behavior of $t()$. Therefore we need to be able to reason incrementally about this enriched behavior and, more to the point of this paper, we need to be able to test the enriched behavior that $t()$ exhibits (or is expected to exhibit) in the derived class as a result of the redefinition of $h()$.

First let us consider how the behavior of $t()$ may be specified in the base class so that we can reason incrementally about it in the derived class. The approach used in [3, 15] to specify the behavior of $t()$ in the base class is to include suitable information, in its specification, about the sequence of hook method calls $t()$ makes during its execution. This information is in the form of conditions on the value of the *trace* variable τ associated with $t()$ that records information about these calls. This information can then be used [15] to arrive at the richer behavior to $t()$ in the derived class[3] by combining it with the derived class specification of $h()$. In this paper we will see how we can test the implementation of C and D, in particular the code of $t()$ (and $h()$), to check whether it satisfies this richer specification about its behavior.

This is a challenging task because we need to keep track of the value of τ. Every time $t()$ makes a call to $h()$ (or another hook method), τ has to be updated to record information about this call (and return) but, of course, there is nothing in the code of $t()$ to do so. After all, τ is a variable introduced by us in order to help reason about the behavior of $t()$, not something included by the designer of C. One possible solution to this problem would be to *modify* the code of $t()$ to include suitable (assignment) statements, immediately before and after each hook method call, that would update τ appropriately. But this would violate our requirement that we not assume access to the body of C, and certainly not modify it. As we will see, it turns out that we can, in fact, exploit polymorphism in the same way that template methods do, to address this problem.

[2] In *C++*, we could simply declare TC a friend of C but, as is widely recognized, the friend mechanism is subject to serious abuse.
It is also worth noting that if the state of tc is such that *pre.m* is not satisfied, the body of test_m() would be entirely skipped; this may be considered a truly extreme instance of poor test-case-choice!

[3] Ruby and Leavens [14] (see also earlier work by Kiczales and Lamping [7, 9]) present a formalism where *some* additional information about a method beyond its functional behavior is provided; this may include, for example, information about the variables the given method accesses, the hook methods it invokes, etc. While this is not as complete as the information we can provide using traces, it has the important advantage that it is relatively easy to build tools that can exploit this information, or indeed even mechanically extract this type of information from the code, rather than having to be specified by the designer.

3. INCREMENTAL REASONING

Let us consider a simple example consisting of a bank account class as the base class (and a derived class we will define shortly). The definition (in *Java*-like syntax) of the Account class appears in Figure 1. The member variable bal

```
class Account {
  protected int bal; // current balance
  protected int nautos; // no. of 'automatic' transactions
  protected int autos[]; // array of automatic transactions
  public Account() { bal = 0; nautos = 0 ; }
  public int getBalance() { return bal; }
  public void deposit(int a) { bal += a; }
  public void withdraw(int a) { bal −= a; }
  public final void addAuto(int a) {
    autos[nautos] = a; nautos++; }
  public final void doAutos( ) {
    for (int i=0; i < nautos; i++) {
      if (autos[i] > 0) { deposit(autos[i]); }
      else { withdraw(autos[i]); } } }
}
```

Figure 1: Base class Account

maintains the current balance in the account. The methods deposit(), withdraw(), and getBal() are defined in the expected manner. Their concrete specifications[4] are easily given:

$$\begin{aligned}
&\text{pre.Account.getBalance}() \equiv \mathit{true}\\
&\text{post.Account.getBalance}() \equiv\\
&\quad [\ !\{\text{nautos, autos, bal}\} \wedge (\text{result} = \text{bal})\ ]\\
&\text{pre.Account.deposit(a)} \equiv (\text{a} > 0)\\
&\text{post.Account.deposit(a)} \equiv\\
&\quad [\ !\{\text{nautos, autos, a}\} \wedge (\text{bal} = \#\text{bal}+\text{a})\ ]\\
&\text{pre.Account.withdraw(a)} \equiv (\text{a} > 0)\\
&\text{post.Account.withdraw(a)} \equiv\\
&\quad [\ !\{\text{nautos, autos, a}\} \wedge (\text{bal} = \#\text{bal}-\text{a})\ ] \qquad (1)
\end{aligned}$$

In the post-conditions we use the notation "!S" to denote that the value of each of the variables that appears in the set S is the same as it was at the start of the method in question. The "#"notation, also in the post-condition, is used to refer to the value of the variable at the start of the execution of the method. Thus these specifications simply tell us that deposit() and withdraw() update the value of bal appropriately and leave the other variables unchanged; and getBalance() returns the balance in the account and leaves all variables unchanged. The notation result [11] in the post-condition refers to the value returned by the function in question.

More interesting are the 'automatic transactions'. The autos[] array maintains the current set of automatic transactions, nautos being a count of the number of these transactions. doAutos() is the (only) template method of this class. Whenever it is invoked, it performs each of the transactions in the autos[] array by invoking the hook methods deposit() and withdraw(). A positive value for an array element denotes a deposit, a negative value denotes a withdrawal. Thus doAutos() iterates through the elements of this array, invoking deposit() if the element in question is positive and withdraw() if it is negative. addAuto() allows us to add another transaction to the autos[] array. We will leave the precise specification of addAuto() to the interested reader; its pre-condition would require the parameter value to be not equal to 0, the post-condition would say that autos[] array is updated to include this value at the end of the array (and nautos is incremented by 1).

Let us now consider the specification of doAutos(). An obvious specification for this method would be:

$$\begin{aligned}
&\text{pre.Account.doAutos}() \equiv \mathit{true}\\
&\text{post.Account.doAutos}() \equiv\\
&\quad [\ !\{\text{nautos, autos}\} \wedge\\
&\quad (\text{bal} = \#\text{bal}+(\Sigma(k = 0 \ldots \text{nautos} - 1).\, \text{autos}[k])\ ] \qquad (2)
\end{aligned}$$

This specifies that doAutos() updates bal appropriately. What is missing is information about the hook method calls that it makes during execution. As a result, although (2) is correct in what it specifies, it proves inadequate in allowing us to reason about the enriched behavior that this method will exhibit in the derived class, to which we turn next.

```
class NIAccount extends Account {
  protected int tCount; // transaction count
  public NIAccount() { tCount := 0; }
  public void deposit(int a) { bal += a; tCount++; }
  public void withdraw(int a) { bal −= a; tCount++; }
  public int getTC() { return tCount; }
}
```

Figure 2: Derived class NIAccount

The enrichment provided by NIAccount (for 'New and Improved account'!) is fairly simple: it keeps a count of the number of transactions (deposits and withdrawals) performed on the account. This is achieved by redefining deposit() and withdraw() appropriately[5]. The newly defined method, getTC() allows us to find the value of the transaction count. The specifications of these methods are straightforward modifications of (1). We will only write down the specs for getTC() and deposit():

$$\begin{aligned}
&\text{pre.NIAccount.getTC}() \equiv \mathit{true}\\
&\text{post.NIAccount.getTC}() \equiv\\
&\quad [\ !\{\text{nautos, autos, bal, tCount}\} \wedge (\text{result} = \text{tCount})\ ]\\
&\text{pre.NIAccount.deposit(a)} \equiv (\text{a} > 0)\\
&\text{post.NIAccount.deposit(a)} \equiv\\
&\quad [\ !\{\text{nautos, autos, a}\} \wedge (\text{bal} = \#\text{bal}+\text{a})\\
&\quad \wedge (\text{tCount} = \#\text{tCount}+1)\ ] \qquad (3)
\end{aligned}$$

Let us now turn to the behavior of doAutos() in the NIAccount class. It is clear from the body of this template method, as defined in the base class, that during its execution, the value of tCount will be incremented by the number of transactions in the autos[] array, i.e., by the value of nautos, since doAutos() carries out each of these transactions by invoking deposit() or withdraw(). But we cannot arrive at this conclusion from its specification (2), not even given the specification (3) for the behavior of the redefined

[4]This class is so simple that its abstract specification would essentially be the same as its concrete specification. Note also that we have included the name of the class in the specs since we will also consider the behavior of these methods in the derived class. Thus, (1) specifies the behavior of these methods when applied to an instance of the Account class.

[5]If these methods were at all complex, it would have been appropriate to invoke the base class methods in their definitions; here, the only task to be performed by the base class portion is to update bal, so we have just repeated the code.

hook methods that doAutos() invokes. The problem is that there is nothing in (2) that in fact tells us that doAutos() invokes deposit() or withdraw(). Indeed, if we rewrote the body of doAutos() so that it directly added each element of the autos[] array to bal, instead of invoking deposit() and withdraw() to perform the transactions, it would still satisfy the specification (2) but, of course, this rewritten method, in the NIAccount class (i.e., when applied to a NIAccount object) would *not* change the value of tCount.

Consider the following more informative specification:

$$\begin{aligned}&\text{pre.Account.doAutos( )} \equiv (\tau = \varepsilon)\\&\text{post.Account.doAutos()} \equiv\\&\quad [\ !\{\text{nautos, autos}\} \wedge (|\tau| = \text{nautos})\\&\quad \wedge (\text{bal} = \#\text{bal} + (\Sigma(k = 0 \ldots \text{nautos} - 1).\ \text{autos}[k]))\\&\quad \wedge (\forall k : (1 \leq k \leq |\tau|) :\\&\qquad \tau[k].m \in \{\text{deposit, withdraw}\})\ ] \qquad (4)\end{aligned}$$

τ denotes the *trace* of hook method calls that doAutos() makes during its execution. At its start, doAutos() has not made any hook method calls, so τ is ε, the empty sequence. Each hook method call (and corresponding return) is recorded by appending a single element to τ. This element consists of a number of components, including the name of the method in question, the parameter values passed in the call, the returned results, etc.; for full details, we refer the reader to [15]. Here we are interested only in the identity of the method; $\tau[k].m$ gives us the identity of the method invoked in the call recorded in the k^{th} element of τ. Thus the post-condition in (4) states that when doAutos() finishes, it would have made as many hook method calls as nautos, the number of automatic transactions in the autos[] array, and that each of these calls will be to either deposit() or withdraw(). This specification can, using the *enrichment rule* of [15], then be combined with the specification (3) to arrive at the following:

$$\begin{aligned}&\text{post.NIAccount.doAutos()} \equiv\\&\quad [\ !\{\text{nautos, autos}\} \wedge (|\tau| = \text{nautos})\\&\quad \wedge (\text{bal} = \#\text{bal} + (\Sigma(k = 0 \ldots \text{nautos} - 1).\ \text{autos}[k]))\\&\quad \wedge (\forall k.(1 \leq k \leq |\tau|).\ \tau[k].m \in \{\text{deposit, withdraw}\})\\&\quad \wedge (\text{tCount} = \#\text{tCount} + \text{nautos})\ ] \qquad (5)\end{aligned}$$

This asserts, as expected, that doAutos() increments the transaction count appropriately. Informally speaking, what we have done here is to 'plug-in' the additional information provided by the derived class specs (3) of the hook methods, into the specification (4) of the template method, to arrive at the enriched behavior of the template method in the derived class.

4. TESTING POLYMORPHIC BEHAVIOR

Suppose we wanted to test the class Account to ensure that it behaves as expected, i.e., according to its specifications. We could use the approach outlined in Section 2 to define the corresponding test class, TAccount shown partially in Figure 3. tAccount is the test account object. rg as an object of type Random, to be used for generating random values (for use as parameter values). t_deposit() is the test method corresponding to deposit(). We generate a random amount rd to deposit into tAccount, and if the pre-condition of deposit() (as specified in (1)) is satisfied, we invoke deposit(rd) on tAccount, and then assert that the post-condition of deposit() must be satisfied, with appropriate substitutions such as replacing bal by tAccount.bal being made. Note that we also need the save the starting values of the data members of tAccount since the post-condition refers to these values. We have shown only one of these in the figure, oldbal being the variable in which the starting balance in tAccount is saved. Of course, when the data member in question is more complex, such as the array autos[], this becomes somewhat more involved; and if the member is an object (of a type defined by the user), this will require, as we noted in Section 2, that the corresponding class provide a *cloning* operation.

```
class TAccount {
  protected Account tAccount; // test object
  Random rg;
  public void t_deposit( ) {
    int rd = rg.nextInt(); int oldbal = tAccount.bal; ...
    if( rd > 0 ) { tAccount.deposit(rd);
      assert( (tAccount.bal = oldbal+rd) ∧ ... ); }
}
```

Figure 3: Test class TAccount

The test methods t_withdraw() and t_getBal() are similarly written, and we will omit them. Let us consider the template method doAutos(). If we were only interested in the specification (2) which gives us information only about the functional effect that doAutos() has on the data members of the Account class, this too would be straightforward[6]. But a key aspect of the behavior of doAutos(), indeed the aspect that qualifies it as a template method and makes it possible to define derived classes that enrich its behavior by simply redefining deposit() and/or withdraw(), is of course the calls it makes to these hook methods. Thus if we are to really test the implementation of doAutos() against its expected behavior, the testing must be against the trace-based specification (4).

However, we face an important difficulty in doing this. The problem is that the trace variable τ which plays a key role in this specification is not an actual member variable of the Account class. We could, of course, introduce such a variable in the test class TAccount but this won't serve our purpose. The problem is that τ has to record appropriate information about the hook method calls that doAutos() makes *during* its execution; this cannot be done in the test method t_doAutos() before it calls doAutos() or after doAutos() returns. In other words, what we need to do is to 'track' doAutos() *as it executes*; whenever it it gets ready to make a hook method call, we have to 'intervene', record appropriate information about the call – in particular, the name of the method called, the parameter values, the state of the object at the time of the call – and then let the call proceed; once the hook method finishes execution and returns control to doAutos(), we again need to intervene and record information about the results returned and the (current) state of the object. One possible way to do this would be to insert the appropriate statements to update the value of τ before and after each hook method call in the body of doAutos(); but this would not only require access to the source code of doAutos(), it will require us to *modify* that source code, and

[6]One question here would be that of generating a random value in the tAccount.autos[] array; indeed, in general, the test object should be in a random (reachable) state, rather than being initialized to some 'standard' state; but this question is independent of inheritance and polymorphism, so we will ignore it here.

this is clearly undesirable.

The solution turns out to be provided by polymorphism itself. The key is to define TAccount not as a class that includes a member variable of type Account but rather to have TAccount as a derived class of Account. We call this new test class T2Account in order to distinguish it from the original test class TAccount. T2Account appears in Figure 4. The variable tau of T2Account is the trace variable in which

```
class T2Account extends Account {
  protected trace tau; // trace variable
  public void deposit(int aa) {
    // add element to tau to record info such as
    // name of method called (deposit),
    // parameter value (aa) etc., about this call;
    super.deposit(aa);
    // add info to tau about the result returned
    // and current state.
  }
  // withdraw() will be similarly defined.
  public void t_doAutos( ) {
    tau = ε;
    // check pre-condition, then call doAutos(),
    // assert post-condition.
  }
}
```

Figure 4: Test class T2Account

we record information about the sequence of hook method calls that doAutos() will make during its execution.

The t_doAutos() method starts by initializing tau to ε, then calls doAutos() (on the *self* object). Let us consider what happens when doAutos() executes, in particular when it invokes the deposit() method (withdraw() is, of course, similar, so we won't discuss it). We have redefined deposit() in T2Account, so this call in doAutos() to deposit() will be dispatched to T2Account.deposit() since the object that doAutos() is being applied to is of type T2Account. Now T2Account.deposit() is simply going to delegate the call to the Account.deposit() but before it does so, it records appropriate information, such as the name of the hook method called ('deposit'), the parameter value (aa), etc., about this call on tau. Next, T2Account.deposit() calls the deposit() defined in Account; when Account.deposit() finishes, control comes back to T2Account.deposit(); T2Account.deposit() now records additional information (about the result returned, current state of the object, etc.), and finishes, so control returns to Account.doAutos(). The net effect is that the original code, Account.deposit(), of the hook method invoked has been executed but, in addition, information about this call has been recorded on the trace. And to do this, we did not have to modify the code of any of the methods of Account, indeed we did not even need to be able to see that code.

One point might be worth stressing: T2Account.deposit() is *not* the test method corresponding to deposit(); rather, it is a redefinition of the hook method Account.deposit() in order to record information about calls that template methods might make to this hook method, the information being recorded on the trace of the template method. If there is more than one template method, we might consider introducing more than one trace variable, and yet another variable to keep track of which template method is currently being tested so that the redefined hook methods can record the information on the correct trace variable. This is in fact not necessary since only one template test method will be executing at a time, and it starts by initializing tau to ε. Of course we have assumed that we can declare tau to be of type "trace". If we really wanted to record all the information that tau has to contain in order to ensure completeness of the reasoning system [15], things would be quite complex. We can simplify matters somewhat by only recording the identities of the hook methods called and the parameter values and results returned. This is a topic for further work.

This approach can also be used for testing *abstract* classes, i.e., classes in which one or more of the hook methods may be abstract (in *Java* terminology; pure virtual in *C++*, deferred in *Eiffel*). The only change we have to make is that in T2Account.deposit(), we cannot invoke super.deposit(); instead, we would just record information in tau and return to doAutos(). Note that the specifications (2) and (4) would also be quite different. For one thing, we cannot really establish (2) because, if Account.deposit() (and, presumably, Account.withdraw() as well) is abstract, there is no way to tell what effect doAutos() will have on bal, etc. Nevertheless, the portion of (4) that refers to the hook methods invoked can still be specified since the basis for this can be seen from the body of the template method, so the designer of the Account class could have written this down as part of the specification of doAutos(). The t_doAutos() method will then test that doAutos() does indeed satisfy the expectation about the hook methods it will call[7].

Let us now consider the derived class NIAccount. How do we construct the test class TNIAccount? We cannot define it as a derived class of T2Account because then the redefinitions of the hook methods in NIAccount would not be used by the test methods in TNIAccount. In fact, in general, test classes should be *final*; i.e., a given test class TC is only intended to test that the methods of the corresponding class C meet their specs. A different class D, even if D is a derived class of C, would have to have its own test class defined for it. Of course, TNIAccount would be quite similar to T2Account. The important differences would be that we would have test methods corresponding to any new methods defined in NIAccount, and pre- and post-conditions would be the ones from the specifications (such as (3) and (5)) of this class.

Before concluding this section, we should note one other point. An important assumption we have made is that hook methods obey behavioral subtyping [10], i.e., any redefinitions of hook methods in the derived class must continue to satisfy their base class specifications. If this were not the case, the reasoning that we have performed in the base class about the behavior of the template method, including the trace-based specification of that method, may no longer be valid. For example, suppose a template method $t()$ first calls the hook method $h1()$; if the value returned by $h1()$ is positive, $t()$ then calls $h2()$, else it calls $h3()$. Suppose

[7]In fact, we would not only want to be assured about the identity of the hook methods called or the number of times they are called (which are the pieces of information provided by (4)) but also the parameter values passed in these calls as well as the state just before the calls, etc.; this is particularly important if the hook method in question is abstract. This additional information can be provided using our traces although the resulting specs are naturally much more involved [15].

also that the base class specification of $h1()$ asserts that it will return a positive value. When reasoning about the base class, we might then establish, on the basis of this specification of $h1()$, a specification for $t()$ which asserts that the identity of the first hook method that $t()$ calls (as recorded in the first element of the trace τ of $t()$) is $h1()$, and the identity of the second method called is $h2()$. Suppose now we redefine $h1()$ in the derived class so that it returns a negative value. Then, in the derived class, $t()$ will not satisfy its specification, and the problem is not with $t()$ but with the way that $h1()$ was redefined. The redefined $h1()$ does not satisfy its base class specification, i.e., it violates behavioral subtyping. Hence, when testing the behavior of the hook methods in the derived class, it may be useful not just to test against the derived class specification of the method, but also against its base class specification to ensure that the redefined hook method still satisfies that specification.

5. DISCUSSION

Let us briefly consider a class C that has a member variable acc of type Account. In reasoning about the behavior of the methods of C, we will of course depend upon the specifications of the Account class. Do we have to worry about the specifications of the NIAccount class? Yes, indeed. The point is that for a particular object that is an instance of C, the acc component may well be of type NIAccount[8]. In fact, one reason for defining classes such as NIAccount is precisely that client classes such as C can take advantage of the enrichment provided by this class. What are the issues that we have to consider in reasoning about and testing the behavior of C?

One possibility would be that in reasoning about C, we only take account of the specification of Account. And in testing C, we only create instances of C that have an acc component of type Account. But this is clearly insufficient. We need to test the behavior of C for instances that have an acc component of type NIAccount. In fact, whenever a new derived class of a base class such as Account is defined, the behavior of any client code of Account has to be re-tested [13]. While this may seem undesirable, it is to be expected. After all, by defining a new derived class of Account, we are enriching the behaviors that a client class, such as C, of Account can exhibit; so naturally we have to test for such richer behaviors. The techniques for reasoning about such richer behaviors of C, as well as the corresponding techniques for testing them, are topics for further work.

[8]Of course, in languages like $C++$ for this to happen, acc would have to be a pointer to Account but this is a language detail which we can ignore.

6. REFERENCES

[1] K. Arnold, J. Gosling, and D. Holmes. *Java Programming Language, Third Edition*, 2000.

[2] E. Berard. *Essays on object oriented software engineering*. Prentice-Hall, 1993.

[3] M. Buchi and W. Weck. The greybox approach: when blackbox specifications hide too much. Turku Centre for Computer Science TR No. 297, 1999, http://www.tucs.abo.fi/.

[4] K.K. Dhara and G.T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proc. of 18th Int. Conf. on Softw. Eng.*, pages 258–267. IEEE Computer Soc., 1996.

[5] S. Edwards, G. Shakir, M. Sitaraman, B. Weide, and J. Hollingsworth. A framework for detecting interface violations. In *Proc. of 5th Int. Conf. on Softw. Reuse*. IEEE, 1998.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable OO software*. Addison-Wesley, 1995.

[7] G. Kiczales and J. Lamping. Issues in the design and specification of class libraries. In *OOPSLA '92*, pages 435–451, 1992.

[8] P. Krishnamurthy and P. Sivilotti. The specification and testing of quantified progress properties in distributed systems. In *23rd Int. Conf. of Software Eng.* ACM, 2001.

[9] J. Lamping. Typing the specialization interface. In *OOPSLA*, pages 201–214, 1993.

[10] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. on Prog. Lang. and Systems*, 16:1811–1841, 1994.

[11] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[12] G. Myers. *The art of software testing*. John Wiley, 1979.

[13] D. Perry and G. Kaiser. Adequate testing and OO programming. *Journal of Object Oriented Programming*, 2:13–19, 1990.

[14] C. Ruby and G. Leavens. Safely creating correct subclasses without seeing superclass code. In *OOPSLA 2000*, pages 208–228. ACM, 2000.

[15] N. Soundarajan and S. Fridella. Framework-based applications: Incremental development to incremental reasoning. *Proc. of 6th Int. Conf. on Softw. Reuse*, pp. 100–116, Springer, 2000.

[16] N. Soundarajan and B. Tyler. Specification-based incremental testing of object-oriented systems. In *TOOLS 39*, pp. 35–44, IEEE CS Press, 2001.

Spying on Components: A Runtime Verification Technique

Mike Barnett and Wolfram Schulte
Microsoft Research
One Microsoft Way
Redmond WA, 98052-6399, USA
{mbarnett,schulte}@microsoft.com

ABSTRACT

A natural way to specify component-based systems is by an *interface specification*. Such a specification allows clients of a component to know not only its syntactic properties, as is current practice, but also its semantic properties. Any component implementation must be a behavioral refinement of its interface specification. We propose the use of executable specifications and a runtime monitor to check for behavioral equivalence between a component and its specification. Furthermore, we take advantage of the COM infrastructure to perform this kind of *runtime verification* without any instrumentation of the implementation, i.e., without any recompilation or re-linking.

1. INTRODUCTION

We believe that component-based programming needs formal specifications at the interface level. Currently there are standardized ways to formally specify the syntactic properties of a component, for example, by type libraries or IDL files for COM components [7]. However, the proper mechanism for specifying semantic properties is still an open research topic. Clearly, clients of a component, whether they are human or other software components, require some way of understanding the behavior of a component. Natural language descriptions, while valuable, are often incomplete or ambiguous and are in any case limited to human consumption.

Even if there was agreement on a particular specification technique, there is still the problem of ensuring that a particular component does indeed implement its specification. We propose an answer to the first problem and a technique that partially addresses the second problem.

Our approach for specifying components is to use AsmL to write an executable specification at the highest level of abstraction that defines the behavior of a component as seen through its interface by a client. AsmL is an industrial-strength specification language we have developed at Microsoft Research. Based on the theory of Abstract State Machines (ASMs) [16], it allows the writing of operational specifications at any given level of abstraction. Using it, we have built models of real-world components, like intelligent devices, internet protocols, debuggers and network components [2, 14]. Because ASMs have a formal semantics, an AsmL specification is itself a formal specification.

For the second problem, we use AsmL's native COM connectivity and the COM infrastructure to dynamically monitor the execution of a component. By checking for behavioral equivalence between the component and its concurrently executing specification we ensure that, during a particular run, the component is a behavioral refinement of its specification, i.e., a behavioral subtype [22].

There are two major issues that we do *not* address in this paper: non-deterministic specifications and callbacks. Both are crucial elements of component-based specification and we have developed solutions for both of them, but they are beyond the scope of this paper. Although our system is implemented for COM components, it applies to any component technology that uses dynamic linking.

The paper is organized as follows. Section 2 gives an overview of AsmL. Section 3 explains how to use AsmL to write an interface specification. Then in Section 4 we explain our technique for runtime verification. In Section 5 we describe some initial experiments we have conducted within Microsoft. An overview of similar approaches is discussed in Section 6; Section 7 summarizes, presents limitations, and describes future work.

2. AsmL

We write executable specifications of components in the Abstract State Machine Language (AsmL). The language is based on the theory of Abstract State Machines [16]. It is currently used within Microsoft for modeling, rapid prototyping, analyzing and checking of APIs, devices, and protocols.

The key aspects which distinguish AsmL from other related specification languages are:

- it is executable,
- it uses the ASM approach for dealing with state,
- it has a full-fledged object and component system,
- it supports writing non-deterministic specifications.

Our web site [13] contains a complete description as well as an implementation that is freely available for non-commercial purposes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA 2001 Workshop on Specification and Verification of Component-Based Systems Oct. 2001 Tampa, FL, USA
Copyright 2001 M. Barnett and W. Schulte.

Because AsmL has native COM connectivity (the next release will also be integrated into the .NET framework), one can not only specify components in AsmL and simulate them but also substitute low-level implementations by high-level specifications. This substition allows heterogeneous systems to be built, partly developed using standard programming languages and partly using executable specifications. It is also crucial for implementing runtime verification without the need for instrumenting the implementation.

Although not shown in this paper, non-determinism is one of the key features of AsmL. It allows designers to clearly mark those areas where an implementation must make a decision. In AsmL, non-determinsm is restricted; you can choose or quantify only over bounded sets [5].

AsmL specifications are *model programs*: they are operational specifications of the behavior expected of any implementation. Thus, they provide a *minimal model* by constraining implementations as little as possible. There are three main properties of AsmL that support this.

1. The ASM notion of *step* allows the specifier to choose an arbitrary granularity of sequentiality. Within a single step, all updates (assignment statements) are evaluated in parallel; locations (variables) are written in one atomic transaction at the end of the step. Using the maximal step size means that no unnecessary sequencing is forced on the implementer. An implementation is free to choose an evaluation order consistent with efficiency considerations.

2. Non-deterministic (bounded) choice is a basic construct in the language. Although non-determinism is undesirable in an implementation, non-deterministic specifications allow implementations to make the correct engineering decisions. For instance, a specification might say that any element satisfying certain conditions can be returned from a collection where the implementation might make a particular choice based on the efficiency of searching the data structures that are employed.

3. High-level data structures and programming constructs allow a specification to be expressed in ways that might not be acceptable when efficiency is the primary concern. For instance, a specification might not bother to normalize a data structure, but instead re-organize and manipulate it each time it is accessed.

In general, the primary goal of a specification is to be as clear and understandable as possible; the goal of an implementation is to meet engineering considerations such as execution time, storage efficiency, etc.

Compared to an implementation language such as C++, we have found AsmL specifications to be an order of magnitude more compact. While part of this is due to the advantages offered by any higher-level notation, some part is caused by the specific features of AsmL enumerated above.

3. INTERFACE SPECIFICATIONS

Figure 1 presents a small example that we use throughout. It is not COM-specific. Although written in AsmL, it corresponds exactly to an interface expressed in IDL. AsmL makes implicit the fact that COM methods also return a status value in addition to whatever other values they return; compiler-generated code handles that automatically.

```
interface ICanvas
  createFigure(...) as IFigure

interface IFigure
  getColor() as Color
  setColor(c as Color)
  getBorder() as IBorder

interface IBorder
  getWidth() as Integer
  setWidth(i as Integer)
```

Figure 1: Example Interfaces: Syntax Only

```
interface ICanvas
  createFigure(c as Color, ...) as IFigure =
    new IFigure(c,...)

interface IFigure
  var color as Color
  border as IBorder = new IBorder(3)
  getColor() as Color = color
  setColor(c as Color) = color := c
  getBorder() as IBorder = border

interface IBorder
  var w as Integer
  getWidth() as Integer = w
  setWidth(i as Integer) =
    if i < 0 then throw Exception(...) else w := i
```

Figure 2: Example Interfaces: Semantics

The example provides interfaces for a component-oriented drawing program: a client interacts with a root interface, *ICanvas*, to create and manipulate geometric figures, which support the interface *IFigure*. Each figure has a nested object, a border, which supports the interface *IBorder*. That is, a component supporting the *IFigure* interface also must be able to provide a reference to an *IBorder* interface. Whether this reference is actually to a separate component, or just a different interface on the same component is exactly the kind of underspecification that component-based programming encourages.

The method *createFigure* returns a reference to the *IFigure* interface on the figure that is created. A figure's border is created with some default attributes; the attributes can be changed later through calls to methods such as *setWidth*. Note the syntax of the interface definitions alone allows data values and interface references to be distinguished.

An example AsmL specification for this interface is shown in Figure 2. It is written as a model program, as opposed to a set of pre- and post-conditions (although AsmL does provide also that style of specification). It is a particularly trivial model; this is good — such a trivial component should not have a complicated specification.

Note that the method *setWidth* throws an exception if the argument is less than zero. The exception must belong to

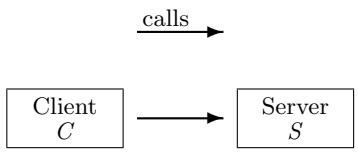


Figure 3: A client-server architecture

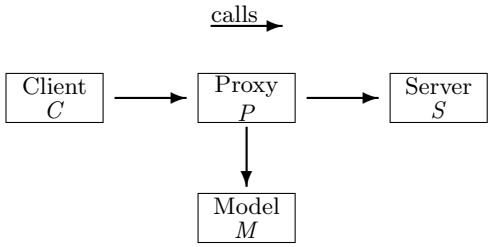


Figure 4: Proxy Architecture

some interface, but we do not show it.

4. RUNTIME VERIFICATION

A component that interacts with an implementation of *ICanvas* is a *client*, while the implementation is a *server*; together their architecture is shown in Figure 3. We refer to the client program as C and the implementation of the server as S. The important feature implied by such an architecture is that the client is completely unaware of the identity of the server component. C is aware solely of the functionality provided through whatever interfaces are supported by S. This is the crucial feature we rely on for implementing our runtime verification.

To enable the AsmL specification to spy on the interactions between C and S, we insert a component, P, which operates as a proxy, as shown in Figure 4. Using a proxy allows the interaction of the client C and the server S to be observed without having to instrument (i.e., modify) either component. The proxy forks all of the calls made from C to S so that they are delivered to the AsmL specification or model, M. From now on, we use the letters C, S, M, and P to refer to the client, server, model, and proxy, respectively.

Inserting a proxy is easily accomplished for COM components [7]. Clients can initiate access to a COM component only by making a request to the operating system. That request can be intercepted either with or without the client's cooperation. As long as the value returned to the client is a valid interface reference, the client is unable to distinguish whether the reference is to the actual implementation, S, or to our proxy, P. In fact, it is this property of COM that allows transparent access to COM components that are not local to the machine on which the client is executing.

Runtime verification means that from the client's point of view, the observed behavior of the model is indistinguishable from that of the server, i.e., they are *behaviorally equivalent*. Because this is a dynamic check, it means they are equivalent only on the observed behavior; ideally the specification allows more behaviors. An implementation restricts its behavior, usually for reasons of efficiency.

When runtime verification uncovers a difference in behavior between the specification and the implementation, there is no *a priori* way to know which is "wrong" (unless one assumes that the specification is always correct...). We are unaware of any method for creating perfect specifications; one can only hope to use a specification language that supports a layered approach. Engineering practice has shown the importance of separating unrelated concerns in order to focus on the proper details at issue. We have tried to ensure AsmL is such a language.

4.1 The Proxy P

All method calls between C and S are intercepted by P. As far as C is concerned, it is accessing the functionality provided by S and is unaware of either P or M. P manages the concurrent execution of M and S; it forks every call so that they are delivered to M as well as S. P compares the results from both components, checking at each interface call that they agree in terms of their success/failure codes as well as any return values. (In our examples, we do not explicitly show the checks for the success or failure of the methods.) As long as they are the same, the results are delivered to C. Otherwise S and M are not behaviorally equivalent; the discrepancy is made evident to an observer of the system.

We create P automatically from the definition of the interfaces that are used between C and S. The correct operation of P relies on two properties of object references in S that allow them to be used as identifiers.

1. They must be *stable*: an object reference returned to C maintains its identity in S. The client C can always use that reference to refer to the same object.

2. They can be tested for equality: a reflexive, symmetric operation allows P to distinguish different objects.

We believe both of these properties to be reasonable and easily met; we mention them only to be explicit about our dependencies.

4.2 Verifying Data

For methods that return atomic data values, runtime verification is comparatively simple. P maintains a global table *map*, which stores object references created in P to pairs of corresponding model and server object references:

map as *Map* of *Object* to (*Object* * *Object*)

The datatype *Map* in AsmL is an associative array, i.e., an array whose indices do not have to be integers. Initially this table contains just one entry: the reference of the root object of P along with the tuple containing the references to the roots of M and S. This entry is created when C first connects to P. As object references are returned to the client, the map is kept current, as explained in Section 4.3.

Thus when the client uses an interface reference of type *IFigure* to call *getColor*, it is really calling the method on an instance of a class, *PFigure*, defined in P. The behavior of *PFigure.getColor* is shown in Figure 5. The table *map* is consulted to retrieve the interface references to M and S. Each is an interface reference to an object implementing the interface *IFigure* in M and S, respectively. The method *getColor* is called in each of the components and their return

```
class PFigure implements IFigure
  getColor() as Color =
    let (M, S) = map(me)
    let m = M.getColor()
    let s = S.getColor()
    if s ≠ m then
      throw Exception(...)
    else
      return s
```

Figure 5: *PFigure.getColor*

values are then compared to guarantee that M and S remain equivalent, from the perspective of the client.

Consider the similar method *setWidth* that would be defined on an instance of a class *PBorder*. If the client called it with a negative argument, then M would throw an exception. In such a case, *S.setWidth* should also throw a subtype of the same exception type.

4.3 Verifying Objects

The simple scheme outlined in Section 4.2 breaks down when a method returns an interface reference. For instance, in our example, the methods *createFigure* and *getBorder* both return interface references. Consider the situation when a client calls *createFigure* (which is probably the first method the client will call). Our proxy, P, calls the method on both the implementation and the model. Both M and S will return to P a created object internal to the respective components; the objects must support the *IFigure* interface.

One problem is that there is no way for P to make an equality test between the references returned from M and S. That is, it cannot decide *at this time* whether or not the two figures are the same. That can be decided only as operations returning simple data (such as *getColor*) on those figures are invoked.

Another problem is that P needs to return a reference (to an object supporting the *IFigure* interface) to the client. If the interface reference from S is returned directly to C, then P will no longer be able to monitor the communication between C and S. C may use that interface reference to make further method calls and those calls would go directly to S.

To solve both problems, P creates a a new local object, p, from the class *PFigure*. P installs the pair of objects returned from M and S in the global table *map*, indexed by p. Instead of returning either the reference from M or S, it returns a reference to the local object p. Then, when the client calls *getColor* on p, it is executing the method shown in Figure 5.

In this way, all interface references from S are *spoofed*. (This is the standard way marshalling proxies are created for remote interfaces in COM [7].) Returning the interface reference to the local object means that all future calls can be monitored.

Now, consider the case when an interface reference is *not* new; say it is a reference that has been returned from S in some previous call. For instance, if *getBorder* is called more than once on the same figure, the same reference will be returned. So if S returns an interface reference, s_1, then M must also have returned an interface reference, m_1. Again, there is no way to know if the two interface references refer to two "equal" components: equality cannot be decided between them.

```
class PCanvas implements ICanvas
  createFigure(...) as IFigure =
    let (M, S) = map(me)
    let m = M.createFigure(...)
    let s = S.createFigure(...)
    if (m = nothing) and (s = nothing) then
      return nothing
    else
      let p = checkObjects(m, s)
      if p = nothing then
        let p' = new PFigure()
        map(p') := (m, s)
        return p'
      else
        return p
```

Figure 6: *PCanvas.createFigure*

```
checkObjects(m as Object, s as Object) as Object =
  if (m = nothing) or (s = nothing) then
    throw Exception(...)
  elseif ∃ p ∈ domain(map)
          where map(p) = (m, s) then
    return p
  elseif ∃ p ∈ domain(map)
          where first(map(p)) = m
                  or second(map(p)) = s then
    throw Exception(...)
  else
    return nothing
```

Figure 7: *checkObjects*

However, the references must have been seen together as a pair the previous time; this is where we assume the stability of the interface references. If the two returned references form such a pair, then there is a local object, p, such that $map(p)$ is the pair (m_1, s_1). Then p is the spoof for the pair and should be returned to C. Otherwise, there is some other pair in the map (m_2, s_1), indexed by another local object p'. (Remember the assumption is that s_1 has been seen before, i.e., returned from S at some earlier method invocation.) This is enough evidence to know that M and S are not behaviorally equivalent because they are not responding in the same way to the same method call. The symmetric argument handles the case when m_1 has been seen by P before.

All of these possibilities are illustrated in P's method for *createFigure* as shown in Figure 6. The logic that decides the correspondence (or lack thereof) between the returned interface references is enapsulated in the method *checkObjects*, which is defined in Figure 7. This explains how the entry in the table retrieved in Figure 5 was initially created.

5. EXPERIENCES

Within Microsoft, we have used AsmL for runtime verification in two case studies on existing product components. Since they already existed, we reverse-engineered an AsmL model from the available documentation, discussions with the responsible product group, and (self-imposed) limited access to the source code. We did not want to re-implement the current components, but wanted to have a true n-version system. Both components are of medium-size: between 50 and 100 thousand lines of code (LOC).

The first case study was partially described in [2]. We created a model of the Debug Services component for the .NET Runtime. The Debug Services control the execution of a .NET component in the runtime; a debugger is a client that requests the installation and removal of breakpoints, etc. (In turn, a person executing a debugger is a client of the debugger.) Our model was less than 4K LOC. The published case study is more concerned with describing the methodology for creating the specification. In the course of performing runtime verification, we encountered a violation of the Debug Services protocol. In conversations with the product team, it turned out that there was an unresolved ambiguity in the meaning of one method when used to respond to a callback. While it could not be considered a major bug in any sense, it did make them realize that they had never decided how to resolve the ambiguity even though they had held meetings about it. Had they been using runtime verification, the problem would not have been able to lie hidden for so long.

For our second case study we modeled the Network Configuration Engine that is part of the Windows operating system. The engine is responsible for maintaining a database of installed network drivers and the network paths that exist between them. We wrote the specification only from the documentation; it ended up being about 2K LOC. We performed runtime verification using an automated test suite provided by the product group and again found a discrepency between the model and the implementation. For one particular method, a flag is used to choose between two different behaviors. However in the real implementation there had originally been three different behaviors and the one that was removed was different from the one that was removed from the documentation. This demonstrated the usefulness of having a specification as documentation: had it been used during the development process, the documentation would have been guaranteed to be consistent with the implementation.

6. RELATED WORK

The need to specify and check components is widely recognized (cf. [26]). However there is neither a standard way to specify components nor any standard for checking an implementation's conformance with its specification.

In a recent book, Leavens and Sitaraman [19] summarize the current approaches for specifying components formally. In that book, Leavens and Dhara [20] use the specification language JML to specify Java components. As we do, JML uses model programs in addition to pre- and post-conditions. Our approaches are very similar, but JML is restricted to specifying Java, while AsmL can be used with any programming language. Müller and Poetzsch-Heffter's [25] article in the same volume also concerns the specification of interfaces, but with pre- and post-conditions. Their main concern is the verification of frame properties, i.e., controlling the modifications a method can make.

In Edwards et al. [9], an architecture is proposed for deriving wrappers for any class implementing an interface that is enriched with pre- and post-conditions. Human intervention is required to map the concrete state of the class to the abstract state used in the interface specification. The advantage of our approach is that the operations on the abstract state are independent of the concrete state, so an AsmL specification can check any implementation. However, the use of an abstraction function means that discrepencies can potentially be discovered earlier than by checking behavioral equivalence as we do.

Jonkers, working at Phillips, is also working on interface specifications [18]. In their work on Ispec, they use transition systems to provide the semantics for interface specifications. However they don't try to execute the model in isolation or run it in parallel with the implementation. Instead they want to generate black-box tests.

Besides JML, there has been a lot of work on using assertions to specify Java interfaces, e.g., Contract Java [11, 12], iContract [8], and Jass [4]. And of course, Eiffel [23, 24], uses pre- and post-conditions to specify components. However, these do not introduce model programs as we do.

Closer to our work on runtime verification is the work on program checking as proposed by Blum and Wasserman [6]. They argue that it is often much easier to write a program that checks whether a result is correct, than to prove the algorithm correct that produces the result. For example, it is difficult to factor an integer, but, given x and y, it is trivial to determine whether or not y is a factor of x. In our case the checker is the specification.

Using this idea, Antoy and Hamlet [1] propose the use of algebraic specifications to specify software. Algebraic specifications use high level data structures, thus solving one of the aforementioned problems of pre-/post-conditions. The price is that when checking the implementation against the specification one needs abstraction. Their system is able to run the executable specification (in fact it is a rewrite system) in parallel with the implementation in C; similar to our framework, they check the results on the method boundaries. They include a comprehensive review of similar work; we do not repeat it here. But due to the restricted nature of algebraic specifications, they cannot deal with state or with object identities (without a lot of coding).

Another similar project is the SLAM project by Herranz-Nieva and Moreno-Navarro [17]. They developed a new specification language and define class operations with pre-/post-conditions. The resulting specifications are translated to C++; part of the pre-/post-conditions are compiled to Prolog. Using a bridge between C++ and Prolog, the Prolog clauses are used as assertions during runtime. Results are speculative, since the project is in the early stages of development.

While not specifically relating to interface specification, Erlingsson and Schneider [10] have also developed a method for injecting a runtime monitor into programs to enforce security properties. In their examples, the monitors are derived from finite automata and so are consequently limited. The transitions of the automata must be triggered by events that are observable at the level of machine code. This is appropriate for the security properties they check, but are not

suited for checking interface properties.

Instead of performing checks at runtime, there has been much work using *static analysis* to prove general properties about a program. While it provides a more general result that is true of any execution of the program, the limitations of program analysis enforce a consequent weakening of the set of properties that can be checked. Perhaps the most well known static program checker is ESC/Java [21].

7. CONCLUSIONS

We have presented a specification method for interfaces that allows a component implementing the interface to be run concurrently with its specification with no need for recompiling, re-linking, or any sort of invasive instrumentation at all. While runtime verification does not prove that the component is correct (with respect to its specification), it does guarantee that, for that particular trace, the component is a behavioral subtype of its specification. For systems that are not amenable to current formal verification technology, this may be the highest degree of formal proof possible. To be useful in real-world applications, formal specifications must provide benefit within the existing development processes. Runtime verification can be used as part of current testing techniques, whether directed or ad-hoc.

We have used our methods to model two medium-sized components within Microsoft and performed runtime verification during user scenarios as well as in the context of testing using an automated test suite. Both times we have been able to find discrepencies between the actual component and its specification.

While this presentation has been restricted to deterministic specifications and systems that do not make callbacks, these burdensome qualifications are addressed in a more complicated scheme [3]. Unfortunately, this scheme is subject to exponential worst-case behavior. We are developing a new system that will be integrated into the .NET runtime, which does not suffer from this drawback.

Our specification language, AsmL, allows other opportunities which are beyond the scope of this paper. For instance, we have used it for early prototyping and test-case generation [15].

We believe that runtime verification shows promise in providing automated support for keeping a specification alive and for ensuring that an implementation correctly implements its specification.

8. REFERENCES

[1] Sergio Antoy and Richard G. Hamlet. Automatically checking an implementation against its formal specification. *Software Engineering*, 26(1):55–69, 2000.

[2] Mike Barnett, Egon Börger, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Using Abstract State Machines at Microsoft: A case study. In *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 367–379, Berlin, Germany, March 2000. Springer-Verlag.

[3] Mike Barnett, Lev Nachmanson, and Wolfram Schulte. Conformance checking of components against their non-deterministic specifications. Technical Report MSR-TR-2001-56, Microsoft Research, June 2001. Available from `http://research.microsoft.com/pubs`.

[4] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass — Java with Assertions. http://semantik.informatik.uni-oldenburg.de/~jass/doc/index.html.

[5] A. Blass, Y. Gurevich, and S. Shelah. Choiceless Polynomial Time. *Annals of Pure and Applied Logic*, 100:141–187, 1999.

[6] Manuel Blum and Hal Wasserman. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[7] Don Box. *Essential COM*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1998.

[8] A. Duncan and U. Hölze. Adding contracts to Java with handshake. Technical Report TRCS98-32, University of California at Santa Barbara, December 1998.

[9] Stephen H. Edwards, Gulam Shakir, Murali Sitaraman, Bruce W. Weide, and Joseph Hollingsworth. A framework for detecting interface violations in component-based software. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 46–55. IEEE Computer Society Press, 1998.

[10] Ulfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. Technical Report TR99-1758, Cornell University, Computer Science, July 19, 1999.

[11] Robert Bruce Findler and Matthias Felleisen. Behavioral interface contracts for java. Technical Report TR00-366, Department of Computer Science, Rice University, August 2000.

[12] Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. Object-oriented programming languages need well-founded contracts. Technical Report TR01-372, Department of Computer Science, Rice University, 6100 South Main Stree, Houston, Texas, 77005, 2001.

[13] Microsoft Research Foundations of Software Engineering, 2001. `http://research.microsoft.com/fse`.

[14] Uwe Glässer, Yuri Gurevich, and Margus Veanes. Universal plug and play machine models. Technical Report MSR-TR-2001-59, Microsoft Research, June 2001. Available from `http://research.microsoft.com/pubs/`.

[15] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Testing with Abstract State Machines. In *Formal Methods and Tools for Computer Science, Eurocast 2001*, pages 257–261. IUCTC Universidad de Las Palmas de Gran Canaria, February 2001. Submitted for inclusion in LNCS ASM 2001.

[16] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[17] Angel Herranz-Nieva and Juan Jose Moreno-Navarro. Generation of and debugging with logical pre and post-conditions. http://lml.ls.fi.upm.es/slam/.

[18] H.B. Jonker. Ispec: Towards practical and sound interface specifications. In *IFM'2000*, volume 1954 of *LNCS*, pages 116–135, Berlin, Germany, November 1999. Springer-Verlag.

[19] G. T. Leavens and M. Sitaraman (eds.). *Foundations*

of Component-Based Systems. Cambridge University Press, New York, NY, 2000.

[20] Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.

[21] K. Rustan M. Leino. Applications of extended static checking. In Patrick Cousot, editor, *Static Analysis: 8th International Symposium (SAS'01)*, Lecture Notes in Computer Science, pages 185–193. Springer, July 2001.

[22] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[23] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[24] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[25] P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In *Foundations of Component-Based Systems* [19], pages 137–160.

[26] Clemens Szyperski. *Component Software*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.

Toward Reflective Metadata Wrappers for Formally Specified Software Components

Stephen H. Edwards
Virginia Tech, Dept. of Computer Science
660 McBryde Hall
Blacksburg, VA 24061-0106 USA
+1 540 381 3020
edwards@cs.vt.edu

ABSTRACT

Abstract behavioral specifications for software components hold out the potential for significantly improving a software engineer's ability to understand, predict, and reason soundly about the behavior of component-based systems. Achieving these benefits, however, requires that specifications be delivered along with components to the consumer. This paper considers the question of what is the best way to package specification and verification information for delivery along with a component. Rather than distributing specifications in "source" form, an alternate solution based on reflection is presented. A reflective interface that supports program-level introspective access to behavioral descriptions is proposed. By embodying this interface in a wrapper component, it becomes possible for the reflective interface to also support services for contract violation checking, self-testing, and abstract value manipulation, even when the underlying component technology does not have built-in reflection capabilities.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, programming by contract, assertion checkers, class invariants*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*specification techniques, pre- and post-conditions, invariants, assertions*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object-oriented programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*.

General Terms

Design, Verification.

Keywords

Formal specification, reflection, design by contract, representation invariant, wrapper class, unit test, integration test.

1. INTRODUCTION

Component-based software development (CBSD) is becoming more prevalent every day, carrying with it the hope for greater productivity and software quality. Indeed, off-the-shelf components should be well-seasoned, well-tested, and more reliable than newly written code. An even greater benefit potentially can be provided by well-designed software components, however: they—or more correctly, the abstract specifications that explain their behavior—can help software engineers understand, predict, and reason soundly about the dynamic behavior of component-based software systems. Efforts at formally specifying the behavior of software components aim at maximizing this effect.

To achieve this benefit for a commercial component, naturally the component's (formal) specification must be distributed along with the component itself. For most commercial component technologies, including COM and its derivatives, CORBA, JavaBeans, ActiveX, and .NET, components are distributed in a binary form. Indeed, a central issue in reasoning about component-based software is the problem of correctly reasoning about composite behaviors when source code is unavailable. If the component provider is going to deliver a behavioral specification as well, in what form will it be delivered?

This position paper explores the question of how best to package and deliver a component's formal specification, as well as associated non-code information, along with the component's implementation. Section 2 outlines the problem, while Section 3 sketches a possible solution: delivering a wrapper component that provides access to a wide variety of information, including specification details, through a standardized, reflection-based interface. Section 4 explores the various kinds of metadata and services that may be appropriate to provide through such a reflective wrapper. Section 5 summarizes the limitations of the approach, Section 6 outlines relations with previous work, and Section 7 summarizes the issues covered.

2. PROBLEM AND SIGNIFICANCE

The problem under consideration, as introduced in Section 1, supposes that a component provider also wishes to provide a formal specification (perhaps along with verification information) when a component is delivered:

> *What is the best way to package specification and verification information for distribution to clients along with a component?*

One of the primary driving factors in binary packaging of commercial components is protection of proprietary information or trade secrets embodied in the component's implementation. This concern does not arise with specifications, of course. The client cannot receive many of the benefits of having a component's behavioral specification unless the specification is completely accessible. "Hiding" a specification clearly is at direct odds with the value added by distributing it with a component in the first place. This leads to the naïve view that a formal specification should be delivered in human-readable "source" form. Even something as

simple as a text file containing the specification in a suitable formal notation should suffice.

This simple approach to distributing specifications treats them the same as traditional documentation, which is distributed most frequently in printed form, as plain text, as HTML, or in a platform-specific help file format. Besides simplicity, source distribution of specifications has another strength: it highlights and reinforces the fact that specifications are designed for *communication* with software engineers—they are written for other people to read. In essence, isn't a formal specification the ultimate in rigorous documentation?

While the value of reading specifications cannot be underestimated, treating specifications as "plain old documentation" misses the opportunity to appropriately leverage those specifications in development tools, during component composition, and during automated reasoning or verification tasks. It is certainly possible to force every CBSD tool to pick some specification notation, support parsing/internalization of that notation, and maintain its own representation of the relations between specifications and components. Unfortunately, this strategy implies a huge duplication of effort among CBSD tool implementers, conflicting choices made in different tools, and a number of other inefficiencies. As a result, in asking what is the "best" way to package specifications for distribution, this paper is aiming to support both human and tool consumption and use of specifications.

The significance of this problem to researchers in formally specified components is apparent: a component's client cannot reap many of the benefits of a specification unless the specification is delivered with the component. Similarly, the specification should be delivered in a form that conveniently supports all of the activities the client may wish to perform, including both people-oriented and tool oriented tasks.

3. A POSSIBLE SOLUTION: REFLECTIVE METADATA WRAPPERS

If providing specifications (and other related information) as traditional documentation has disadvantages, what alternatives are available? Consider the history of software components. Providing specifications in source form is analogous in some ways to the "old days" when reusable components were subroutine libraries shared as source code files among programmers. While the client could always refer to the code, the critical interface information (the name, parameter profile, description, and usage of each subroutine) was typically provided in embedded comments or as separate documentation.

Component packaging and distribution has evolved enormously since subroutine libraries first came into use, however. While traditional documentation typically is provided for commercial components, most commercial component technologies, including COM, CORBA, JavaBeans, ActiveX, and .NET all provide some API for "inspecting" a component's interface. In effect, a component "knows" what it exports, and a client can use a well-defined interface to "ask" what operations are available, how many parameters of what type are needed, what properties are provided, and so on. Such an interface is a wonderful boon to component packaging. It naturally allows any development environment to immediately manage and support any newly installed component, it supports "plug-in"-style integration of new features in applications, it supports automatic checks for (syntactic) interface compatibility during composition, and it can even support general-purpose component-level scripting tools in some cases.

3.1 Reflect for a Moment

The ability of a component to respond to queries about its own structure, behavior, or implementation is the cornerstone of **reflection** [24, 11]. Reflective software is capable of representing (and thus operating on at run-time) some aspect(s) of itself. **Computational reflection** is the activity of a computational system when computing about or operating on (and thus potentially altering) its own computation [17]. This concept arose in the programming language arena, and has seriously impacted object-oriented programming language design.

A reflective component can provide two different forms of reflection services: **introspective** capabilities provide read-only access for inspecting component properties, while **intercessory** services allow one to modify a component or alter its behavior in some way [11]. While intercessory protocols are at the heart of computational reflection and metaprogramming [11, 16], the more restricted introspective protocols supported by most component technologies are still powerful tools. In effect, "interface inspection" APIs supported by most commercial components are simply scaled down introspective interfaces.

If a typical component (say, a JavaBean) already supports a standardized interface for reporting on the syntactic properties of its exported features, how difficult can it be to extend that interface to include access to specification-level descriptions? Perhaps the more interesting question is how far can this strategy be taken?

Many OO languages that support reflection, including Java, do so by associating each object with a metaobject that encapsulates information about how that object is structured and how it behaves. Often in class-based OO languages, an object's metaobject is a singleton object representing its class. This class object supports methods to determine the name and parameter profile of the methods supported by instances of the class, the name and type of instance and class-wide data members, the name and number of superclasses, and so on. A class object may also offer intercessory capabilities, such as changing the way method dispatch is supported. Normally, supporting intercessory capabilities requires the metaobject approach to be built-in to the language.

By analogy, it is possible to turn a behavioral specification into a stand-alone component that provides a standard interface for inspecting all facets of the specification. Whereas a traditional OOPL "class" object represents a class' structural or syntactic interface, a specification object instead represents a specification's structural *and behavioral* description. All of the normal reasoning and structuring techniques applied to abstract class collections and hierarchies could also be applied to specification objects. In effect, this strategy turns a behavioral specification into *another operational component* that can be delivered alongside the original component it describes.

Now providing specification information for components appears to be a simple matter: simply design a metaobject system similar to that used in a class-based OOPL, except that metaobjects model and allow access to formal behavioral descriptions instead of simply syntactic interfaces. If we wish to limit ourselves only to introspection, this approach may be satisfactory. However, intercessory services cannot easily be added through specification

objects alone if one is working using an existing component technology where metaobject-based reflection is not built-in to the component model.

3.2 Decorating With Wrappers

Specification objects are a powerful idea, particularly for providing introspective capabilities and for sharing specifications among behaviorally interchangeable components. Introducing them requires little more than adding some kind of `GetSpecification()` method to a component's interface. On the other hand, can intercessory services (that allow changes to component-level behavior) be added to components that are implemented using a non-reflective language or component technology?

It is possible to add some (but obviously not all) intercessory capabilities to any component with the correct design. The **decorator** pattern [7] suggests a simple approach that is suitable to the situation at hand: add the reflective interface by packaging the new operations in a wrapper component. This wrapper should conform to the original specification, but will delegate all of the work involved in the original operations to the component it wraps. We can call such a component a **reflective specification wrapper** (or simply reflective wrapper). At a minimum, this reflective wrapper provides the `GetSpecification()` access to a stand-alone specification object. Further, by interposing a separate processing layer between the client and the underlying component, it becomes possible to add or remove features before or after component operations. This supports a degree of intercessory customization—here, changing the behavior of the wrapper by turning some features on or off, rather than modifying the behavior of the underlying component. Appropriately exploiting this customization from the point of view of behavioral specifications is discussed in Section 4.

While neither the use of wrappers nor the use of reflection is a new idea, the novelty lies in combining the two to provide program-level access to specification information. First, the specification information is clearly turned into another component that can be delivered alongside the original. Further, rather than placing more operations and data inside the underlying component, such a wrapper isolates these features in a separate layer between the component and its client(s) (which now may include a host of development tools in addition to other application code). This approach, which is more in-line with object-oriented design, separates the added features from the underlying code in a way that can be made completely transparent to the remainder of the application, that supports easy insertion or removal of the added capabilities, and that naturally fits with conventional component distribution techniques.

Placing specification-oriented reflection features in a separate class or component is a simple idea, but it refocuses attention with dramatic results. It elevates the reflection features from the level of one or two methods in a component interface up to the level of a separately useful component abstraction. This elevation shifts attention to the question of exactly what "meta" data or services should be provided by a reflective specification wrapper.

3.3 Summarizing the Proposed Solution

The position espoused in this paper is that a component's specification (and other supporting information) should be provided *as another component* (or set of components), distributed in the normal fashion. This position is founded on three insights:

1. Reflection supports both human-readable and tool-based access to and application of the needed information. Reflection naturally supports standardized smart browsing tools and other document navigation aids for human understanding [6], while it also supports uniform automated services that rely on specification data without requiring the duplication of effort necessitated by source code distribution.
2. Wrappers can be used to transparently add features to a component without affecting the underlying entity. Further, they add the ability to support limited forms of intercessory reflection, even when such features are not directly supported in the underlying component technology.
3. If component instances are created using factories [7], client code is completely insulated from dependencies on the concrete implementation used for each instance. This can encapsulate and even parameterize wrapping decisions, so that reflection services can be employed when needed or stripped out when unnecessary without altering clients.

While the position presented here is founded on a wide-ranging collection of prior research, reflective specification wrappers in the form described here have not yet been implemented. Instead, this paper explores the issues and possibilities arising from the proposed approach, both to highlight the problem of packaging and distribution of specifications and to suggest a potential solution for exploration.

4. WHAT METADATA AND SERVICES ARE NEEDED?

If one wishes to provide specification (and verification) information through a reflection interface embodied in a wrapper component, the next issue to face is the question of what data and/or services to support through this interface. As is traditional with reflection, the component information we are concerned with here is truly *metadata*, in the sense that it describes the nature of the component and how it behaves, in contrast to the *data* that the component computes with or transforms. But exactly what metadata or intercessory services should be supported?

4.1 A Component's Formal Specification

The most obvious metadata to provide is some representation of a component's formal specification. Just as a conventional metaobject protocol provides introspective access to an object's class, its methods, and its fields, a reflective specification wrapper should provide introspective access to all aspects of a component's formally specified **exported interface**. This is the role of the specification objects introduced in Section 3.1.

For a model-based component specification [28], the specification object could provide access to the object's **abstract model**, to the **pre- and postcondition for each operation** or method, and to the object's **abstract invariant**. If an algebraic specification approach were used, access would instead be oriented toward axioms and algebras. Beyond the basics, access to publicly available fields or properties, exception behavior, and relationships to other specifications (such as inheritance, perhaps) also need be considered.

However, as Szyperski notes in his definition, there is more to a component than just an exported interface: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only" [25]. This perspective is also shared by the 3C model [15]. As a result, it is clear that in addition to the exported interface, a reflective wrapper should also provide introspective access to a component's **imported interface**: that is, the explicit context dependencies it places on its environment. While the exported interface captures the contract between a component and its client, the imported interface forms the contract(s) between a component and the other, lower-level components on which it is built.

Taken together, providing program-level access to a component's import and export interfaces seems like the bulk of the problem when it comes to packaging and distributing a component's formal specification. On the other hand, elevating the reflection interface to a separate component focuses attention on the other information and services that can be provided through such an object.

4.2 A Component's Verification History

While one's initial concern will necessarily be with distributing specification information, in the long term, consideration of verification information will also be useful. Was a component formally verified? By hand? With tool assistance? Was model-checking used instead? Or was a testing-based approach used? Is a proof or proof fragment available? Upon what assumptions is the verification based?

The extent and quality of verification performed on a component is clearly of interest to the client. In many cases, this information is most useful before making a component purchasing decision. Further, in the ideal situation where local certifiability (also called the modular reasoning property) [26] is ensured by all components in an application, there would be little need for verification details by the client after purchase. However, without local certifiability, verification details are important in supporting application-level verification of component compositions. Further, if formal verification is not used systematically throughout an application, component-level verification information may be useful in localizing defects during testing.

4.3 Violation Checking Services

Component-based development highlights the differing needs and perspectives of the component-provider and the component-user [8]. It is important to provide powerful capabilities for establishing a component's quality to the component-provider. The component-user, on the other hand, must also be provided with the services and information necessary to test her application in combination with the component.

One approach to addressing both concerns is checking interface contracts for violations. In addition to simply providing access to specification and verification information, a reflective wrapper can also provide contract checking features. Because of the way the wrapper is interposed between the client and the component, it is easy to add any or all of the following run-time checks:

- Precondition checks
- Postcondition checks
- Abstract invariant checks
- Representation invariant checks

This idea, originally proposed by the author and colleagues [3], has been used with some success [9, 2]. Postcondition and invariant checking are extremely useful to the component provider during development [1, 2, 19], while precondition checks are useful to the client during component integration. Postcondition and invariant checking can also be useful to the client in defect localization during application testing.

A carefully designed reflective wrapper could allow each category of checks to be enabled or disabled, perhaps on a per-operation basis. Similarly, a component might even offer different levels of checking for some conditions—fast, but less rigorous checks versus slow but tediously thorough checks, for example. An interface that provides a systematic way to query the wrapper for the checks it can provide as well as enable or disable them at designated levels would allow component composition environments to directly support such services in a uniform way.

4.4 Self-Testing Services

Component-based approaches to software construction highlight the need for detecting failures that arise as a result of miscommunication among components. In an invited paper at the 22nd International Conference on Software Engineering, Mary Jean Harrold laid out a roadmap for the future of software testing research and identified testing techniques for component-based systems as one of the fundamental research areas ripe for exploration [8]. Violation checking services address some aspects of testing-based component verification, but additional testing support can be critical in supporting an application develop in the process of verifying an application in combination with a component.

It is possible for a reflective wrapper to provide self-testing capabilities in addition to violation checking services. For example, a selection of component developer-provided test suites (from short and simple to long and thorough) could be embodied in the reflective wrapper. Self-testing can then be performed by executing a selected test suite on the wrapped component—perhaps while also enabling interface violation checking.

Such a testing approach provides a natural, incremental approach to application integration. If each component comes prepackaged with test data (and with violation checking services acting in the role of test oracle), a component's own self-test becomes an ideal "real world" test for the lower-level components on which it depends.

Such an approach could even be expanded to support the integration of client-written test suites into the self-testing scheme. Bruce Weide has also suggested that such a wrapper could potentially be augmented to provide operation call/parameter record and playback capabilities [27].

At this point, the benefit of intercessory services from the point of view of component specifications becomes clear. An appropriately structured reflective wrapper can provide for changes in its own behavior—it can allow one to enable or disable specific actions that occur immediately before or after it delegates calls to the wrapped implementation. Although this does not support intercessory actions on the underlying component, simply adding or removing certain actions before and after delegating to the

wrapped component supports many powerful capabilities oriented toward component composition and testing-based verification.

4.5 Abstract Value Manipulation

The component wrapping scheme previously proposed for interface violation checking [3] uses a novel approach to implementing checks before and after operations. Instead of implementing checks in terms of the concrete implementation values inside the underlying component (and thus violating encapsulation), the component is required to provide the computational equivalent of an abstraction function (or abstraction relation). Program-level classes that correspond to the various mathematical modeling types used in defining the state model and pre- and postconditions for the component are used to represent abstract (specification-level) values. The result is that the wrapper asks the component to "project" an abstract value of its current state as a separate object. All checking and analysis is then done on this object, which is designed to mimic the corresponding mathematical abstraction.

This abstraction relation approach can be co-opted for a reflective specification wrapper to provide additional intercessory capabilities. If a component were to provide both an abstract-relation-based projection function ("convert-to-abstract-model") and a corresponding injection function ("convert-from-abstract-model"), then it would be possible for a development environment or other tool to directly access and manipulate an abstract representation of the state of a component. Further, manipulations of that abstract representation could then be "pumped back down" into the component itself. This approach works naturally for components where the abstraction relation is a one-to-one mapping. For many-to-one or many-to-many mappings from representations to abstract values, practical convert-from-abstract-model injection functions are not always possible, and so such a feature cannot be required for all reflective wrappers.

While such an interface can be used for certain kinds of metaprogramming, in the context of component-based software, greater impact is likely to accrue from using such a capability within a development environment. All components would now have a standard interface for plugging into state visualization tools, for prototyping and testing use, and for interfacing with model-checking tools.

4.6 Documentation?

To come full circle, one can also consider incorporating program-level access to component documentation through a reflective interface, as opposed to providing specification information through traditional documentation. JavaDoc and other embedded documentation strategies push the documentation down into the source code in a way that allows tools to extract, format, package, and navigate it for human readability. In the same manner, one can imagine the specification object obtained from `GetSpecification()` providing component-level, per-operation-level, and per-parameter-level documentation strings in a form suitable for compilation into on-line documentation, use in a component property browser, or use in a documentation search database. As with current components, it is likely that printed documentation will be needed for some purposes, but providing program-accessible documentation through a standard interface may have unexplored benefits.

5. LIMITATIONS

There are a number of drawbacks to explore when considering the wrapper approach proposed here. One immediate concern is that this approach may lead to code bloat in the final application, since each component would now be accompanied by one or more supplementary classes to provide its wrapper, testing support, abstract value manipulation, and other services. The critical aspect of the wrapper approach is that all of these services are designed for use during *development*, when specification information about components is most valuable, rather than after delivery, when component specifications are of little or no use to the end user. Because these additional services wrap the underlying component during development, it is a simple matter to remove them for final release builds, without requiring access to the component's source code.

Another concern is whether or not this approach will demonstrably lead to better quality components. However, the position in this paper is that reflective wrappers are designed to provide a mechanism to deliver and later access a component's specification (particularly by development tools). Solving this problem is necessary to allow developers of component-based software to leverage the formal specifications created by component developers. One should not make the mistake of presuming that any solution to the specification packaging and distribution problem will, by itself, be sufficient to guarantee an increase in software quality.

A more significant concern is the question of how specifications will be communicated through the wrapper interface. A program-manipulable representation of specification features is necessary for this strategy to work. One possibility is to use the Extensible Markup Language (XML) [30] to represent specification information, which would require the development of one or more appropriate Document Type Definitions (DTDs). Such a representation must be generally acceptable in order for tools to support it. It is difficult to imagine that one representation could work for the myriad of specification approaches and notations available today. Instead, such a representation would most likely require difficult choices about the specification approach to be used.

Finally, it is clear that packaging and delivery of specifications using the wrapper approach will require additional work by component developers above and beyond simply creating the specifications. Generation of wrapper boilerplate and implementation of many wrapper services, including representing and accessing specification details, can be automated so that no additional work is required of component developers. However, fully supporting all of the features described in Section 4 will require some manual effort. The primary services that may require additional developer-supplied code are:

- Precondition, postcondition, and invariant checks for violation-checking services.
- Selection (and perhaps even construction) of test suites for self-testing services.
- Abstract model projection and injection functions to support abstract value manipulation.

To make this approach practical, it is clear that a "sliding scale" of wrapper functionality, where a given wrapper provides only some subset of the reflective services described in Section 4, is desirable. Component developers who wish to devote the resources necessary for implementing more comprehensive wrapper features

might then have an advantage in competing for more demanding customers.

6. RELATED WORK

The wrapper approach to providing access to specification information was initially inspired by a prior framework for run-time behavioral contract checking [3] and a larger strategy for end-to-end, automated, specification-based testing [2]. This prior research is also related to formal specification and to verification, as well as specification-based testing and parameterized programming. Because of the sweeping nature of the position advocated here, it is related to and has been influenced by a wide variety of existing work across a selection of topics in design, programming languages, formal specification, testing, and software reuse.

Reflection has a 20-year history in programming languages [24], and has been widely discussed at OOPSLA, at the Annual Workshops on Object-Oriented Reflection and Metalevel Architectures, and more recently at the International Conference on Meta-Level Architectures and Reflection. Kiczales has provided one of the most influential discussions of the subject in the context of CLOS [11]. Ferber described alternative approaches to supporting computational reflection in class-based OOPLs [5]. The proposal in this paper adds nothing new to the realm of reflection—instead it aims to take what has been learned about reflection in the design of object-oriented languages and reapply those insights to a new problem: packaging and providing access to specification information and related services. The primary difference in the approach proposed here is that many useful reflection capabilities can be provided within a framework that does not support reflection simply by using wrappers (although general computational reflection cannot, of course). Past work involving formal specification and reflection has primarily focused on reflection as a specification technique, or on how to specify reflective behavior [14, 23].

Within the reuse community, the issue of providing program-level access to specification features has received little attention. The question of how and what to describe in relation to a component's verification history, however, has been discussed widely under the topic of "component certification" [20, 4, 29, 21, 10]. Lessons from the reuse community provide much insight into what kinds of information may be useful to clients in this regard.

The interface violation checking approach described here [3] naturally meshes with Bertrand Meyer's view of design by contract [18]. A violation checking wrapper is intended to provide run-time checking of such contractual obligations while separating such checks from either of the parties involved. The value added by the wrapper approach results from separating the checking code from both the client and the base component and promoting it to a separately manageable class. This addresses concerns about clutter, expression of more complex conditions, and detracting from the focus of the underlying implementation, while allowing one to easily include or exclude checks on a per-component basis in a plug-and-play fashion.

Alternative approaches to run-time assertion checking include Eiffel [19], iContract [13], Rosenblum's Annotation Pre-Processor (APP) [22], and Kiczales' AspectJ [12]. Eiffel supports compiler-generated run-time checks based on user-provided Boolean assertions phrased in terms of publicly exported class features. iContract provides services similar to those of Eiffel, but for Java programs. APP allows separately defined checking operations to be compiled into or out of C code for assertion checking, but makes no distinction between values at the abstract, specification level and the concrete, implementation level. AspectJ allows one to create a separate aspect containing checking code and then choose whether or not to weave this cross-cutting decision into a non-checking implementation at build time. In many ways, AspectJ is philosophically closest to the approach advocated here.

The self-testing concepts folded into the reflective wrappers proposed here have been most heavily influenced by current research in automated, specification-based testing [2]. The idea of providing self-testing capabilities through a standard interface is orthogonal to the approach(es) used to generate test suites and the approach(es) used to assess correctness. As a result, virtually any testing approach could be integrated into the wrapper strategy.

7. SUMMARY

For clients to receive the benefits provided by formal specifications, those specifications must be distributed to clients along with the components they describe. The discussion presented here explores some of the issues surrounding the question of how best to package and deliver such specification information.

The position taken in this paper is that a component's specification (and other supporting information) should be provided *as another component* (or set of components), consisting of a reflective specification wrapper and associated specification objects. This position is based on three insights: reflection supports both human-oriented and tool-oriented access to specifications; a wrapper approach cleanly allows the addition of interface functionality and opens up powerful reflection capabilities, even when the underlying component technology does not support reflection; and the whole scheme can be implemented in a manner transparent to client code.

Given this position, potential introspective and intercessory services for reflective specification wrappers were explored. In addition to specification and verification information, reflective wrappers could potentially be used to provide services for violation checking, self-testing, and even abstract value manipulation. Even documentation could be accessed programmatically through a reflective wrapper. Although such reflective specification wrappers have not been implemented in the form proposed here, their potential deserves further investigation.

The position taken in this paper only outlines one possible mechanism for packaging and delivering specification information, however. While many of the services and capabilities suggested in this paper require significant research issues to be addressed before one can capitalize on the wrapper approach, the services are orthogonal enough that progress can be made incrementally. Nevertheless, the cornerstone of the approach involves capturing and representing "plain old specifications." First and foremost, this research issue must be solved for the position espoused here to be viable. A program-manipulable representation of specification features is necessary for this strategy to work—perhaps one based on XML. Such a representation must be generally acceptable in order for tools to support it. It is difficult to imagine that one representation could work for the myriad of specification approaches and notations available today. Instead,

such a representation would most likely require difficult choices about the specification approach to be used.

8. ACKNOWLEDGMENTS

Bruce Weide, Murali Sitaraman, and Joseph Hollingsworth have all contributed to the basic approach proposed in this paper; their contributions are greatly appreciated. In addition, we gratefully acknowledge financial support from Virginia Tech and from the National Science Foundation under grant CCR-0113181. Any opinions, conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of NSF or Virginia Tech.

9. REFERENCES

[1] Edwards, S.H. Black-box testing using flowgraphs: An experimental assessment of effectiveness and automation potential. *Software Testing, Verification and Reliability*, Dec. 2000; 10(4): 249-262.

[2] Edwards, S.H. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, June 2001; 11(2).

[3] Edwards, S., Shakir, G., Sitaraman, M., Weide, B.W., and Hollingsworth, J. A framework for detecting interface violations in component-based software. In *Proc. 5th Int'l Conf. Software Reuse*, IEEE CS Press: Los Alamitos, CA,1998, pp. 46-55.

[4] Edwards, S.H., and Weide, B.W. WISR8: 8th Annual Workshop on Software Reuse: Summary and working group reports. *ACM SIGSOFT Software Engineering Notes*, Sept./Oct. 1997; 22(5): 17-32.

[5] Ferber, J. Computational reflection in class based object-oriented languages. *ACM SIGPLAN Notices (Proc. OOPSLA'89)*, Oct. 1989; 24(10): 317-326.

[6] Foote, B., and Johnson, R.E. Reflective facilities in Smalltalk-80. *ACM SIGPLAN Notices (Proc. OOPSLA'89)*, Oct. 1989; 24(10): 327-335.

[7] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[8] M.J. Harrold. Testing: A road map. In *The Future of Software Engineering*, A. Finkelstein, ed., ACM Press, New York, NY, 2000, pp. 61–72.

[9] Hollingsworth, J.E., Blankenship, L., and Weide, B.W. Experience report: Using RESOLVE/C++ for commercial software. In *Proc. ACM SIGSOFT 8th Int'l Symposium on the Foundations of Software Engineering* (San Diego, CA, November 2000), ACM, pp. 11-19.

[10] IEEE. *Supplement to IEEE Standard for Information Technology—Software Reuse—Data Model for Reuse Library Interoperability: Asset Certification Framework*. IEEE Std 1420.1a-1996, Apr. 3, 1997.

[11] Kiczales, G., des Rivieres, J., Bobrow, D.G. *The Art of the Metaobject Protocol*. MIT Press, 1992.

[12] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W.G. Aspect-oriented programming with AspectJ. Available on-line at http://www.aspectj.org.

[13] Kramer, R. iContract—the Java design by contract tool. In *Proc. Technology of Object-Oriented Languages, TOOLS 26*, IEEE CS Press, 1998, pp. 295-307.

[14] Kurihara, M. and Ohuchi, A. An algebraic specification of a reflective language. In *Proc. 15th Annual Int'l Computer Software and Applications Conf., COMPSAC '91*, IEEE CS Press, 1991, pp. 231-236.

[15] Latour, L., Wheeler, T., and Frakes, B. Descriptive and predictive aspects of the 3Cs model, SETA1 working group summary. *Ada Letters (Proc. 1st Symp. Environments and Tools for Ada)*, Spring 1991; 11(3): 9-17.

[16] Lee, A.H. and Zachary, J.L. Reflections on metaprogramming. *IEEE Trans. Software Engineering*, Nov. 1995; 21(11): 883-893.

[17] Maes, P. Concepts and experiments in computational reflection. *ACM SIGPLAN Notices (Proc. OOPSLA '87)*, Dec. 1987; 22(12): 147-155.

[18] Meyer, B. Applying "design by contract." *Computer*, Oct. 1992; 25(10): 40-51.

[19] Meyer, B. *Object-Oriented Software Construction, 2nd Edition.* Prentice Hall PTR: Upper Saddle River, New Jersey, 1997.

[20] Poulin, J., and Tracz, W. WISR'93: 6th Annual Workshop on Software Reuse: Summary and working group reports. *ACM SIGSOFT Software Engineering Notes*, Jan./Feb. 1994; 19(1).

[21] Rohde, S.L., Dyson, K.A., Geriner, P.T., and Cerino, D.A. Certification of reusable software components: Summary of work in progress. In *Proc. 2nd IEEE Int'l Con. Engineering of Complex Computer Systems*. IEEE CS Press, 1996, pp.120-123.

[22] Rosenblum, D.S. A practical approach to programming with assertions. *IEEE Trans. Software Eng.,* Jan. 1995; 21(1): 19-31.

[23] Saeki, M., Hiroi, T., and Ugai, T. Reflective specification: Applying a reflective language to formal specification. In *Proc. 7th Int'l Workshop on Software Specification and Design*, IEEE CS Press, 1993, pp. 204-213.

[24] Smith, B. Reflection and semantics in a procedural language. Technical Report 272, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, 1982.

[25] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[26] Weide, B.W., Heym, W.D., and Hollingsworth, J.E. Reverse engineering of legacy code exposed. In *Proc. 17th Int'l Conf. Software Engineering*, ACM, Seattle, WA, Apr. 1995, pp. 327-331.

[27] Weide, B.W., Heym, W.D., and Ogden, W.F. "Modular regression testing": Connections to component-based software. In *Proc. 9th Annual Workshop on Software Reuse*, Jan., 1999.

[28] Wing, J.M. A specifier's introduction to formal methods. *IEEE Computer*, Sept. 1990; 29(9): 8-24.

[29] Wohlin, C., and Runeson, P. Certification of software components. *IEEE Trans. Software Engineering*, June 1994; 20(6): 494-499.

[30] www.xml.org.

Architectural Reasoning in ArchJava

Jonathan Aldrich Craig Chambers

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350 USA
+1 206 616-1846

{jonal, chambers}@cs.washington.edu

Abstract

Software architecture is a crucial part of the specification of component-based systems. Reasoning about software architecture can aid design, program understanding, and formal analysis. However, existing approaches decouple implementation code from architecture, allowing inconsistencies, causing confusion, violating architectural properties, and inhibiting software evolution. ArchJava is an extension to Java that seamlessly unifies a software architecture with its implementation. ArchJava's type system ensures that the implementation conforms to the architectural constraints. Therefore, programmers can visualize, analyze, reason about, and evolve architectures with confidence that architectural properties are preserved by the implementation.

1. Introduction

Software architecture [GS93][PW92] is the organization of a software system as a collection of interacting components. A typical architecture includes a set of components, connections between the components, and constraints on how components interact. Describing architecture in a formal architecture description language (ADL) [MT00] can make designs more precise and subject to analysis, as well as aid program understanding, implementation, evolution, and reuse.

Existing ADLs, however, are loosely coupled to implementation languages, causing problems in the analysis, implementation, understanding, and evolution of software systems. Some ADLs [SDK+95][LV95] connect components that are implemented in a separate language. However, these languages do not guarantee that the implementation code obeys architectural constraints, but instead rely on developers to follow style guidelines that prohibit common programming idioms such as data sharing. Architectures described with more abstract ADLs [AG97][MQR95] must be implemented in an entirely different language, making it difficult to trace architectural features to the implementation, and allowing the implementation to become inconsistent with the architecture as the program evolves. Thus, analysis in existing ADLs may reveal important architectural properties, but these properties are not guaranteed to hold in the implementation.

In order to enable architectural reasoning about an implementation, the implementation must obey a consistency property called *communication integrity* [MQR95][LV95]. A system has communication integrity if implementation components only communicate directly with the components they are connected to in the architecture.

This paper presents ArchJava, a small, backwards-compatible extension to Java that integrates software architecture smoothly with Java implementation code. Our design makes two novel contributions:

- ArchJava seamlessly unifies architectural structure and implementation in one language, allowing flexible implementation techniques, ensuring traceability between architecture and code, and supporting the co-evolution of architecture and implementation.
- ArchJava also guarantees communication integrity in an architecture's implementation, even in the presence of advanced architectural features like run time component creation and connection.

The rest of this paper is organized as follows. After the next section's discussion of related work, section 3 introduces the ArchJava language. Section 4 formalizes ArchJava's type system and outlines a proof of soundness and communication integrity in ArchJava. Section 5 briefly describes our initial experience with ArchJava. Finally, section 6 concludes with a discussion of future work.

2. Related Work

A number of architecture description languages have been defined to describe, model, check, and implement software architectures [MT00]. Many ADLs support sophisticated analysis, such as checking for protocol deadlock [AG97] or formal reasoning about correct refinement [MQR95]. Some ADLs allow programmers to fill in implementation code to make a complete system [LV95][SDK+95]. However, there is no guarantee that the implementation respects the software architecture unless programmers adhere to certain style guidelines.

Tools such as Reflexion Models [MNS01] have been developed to show an engineer where an implementation is and is not consistent with an architectural view of a software system. These tools are particularly effective for legacy systems, where rewriting the application in a language that supports architecture directly would be prohibitively expensive.

The UML is an example of specification languages that support various kinds of structural specification. UML's class diagrams can show the relationships between classes, and UML's object diagrams show relationships between object instances. However, in most UML tools, these diagrams are only intended to show *some* of the ways in which classes and objects can interact—they cannot be used to argue that no other kinds of interaction are possible, and thus do not support communication integrity. Object hierarchies can be expressed using composition

relationships, but this relationship does not enforce communication integrity either, because elements of the composition can still interact with outside objects.

A number of computer-aided software engineering tools allow programmers to define a software architecture in a design language such as UML, ROOM, or SDL, and fill in the architecture with code in the same language or in C++ or Java. While these tools have powerful capabilities, they either do not enforce communication integrity or enforce it in a restricted language that is only applicable to certain domains. For example, the SDL embedded system language prohibits all data sharing between components via object references. This restriction ensures communication integrity, but it also makes these languages very awkward for general-purpose programming. Many UML tools such as Rational Rose or I-Logix Rhapsody, in contrast, allow method implementations to be specified in a language like C++ or Java. This supports a great deal of flexibility, but since the C++ or Java code may communicate arbitrarily with other system components, there is no guarantee of communication integrity in the implementation code.

Component-based infrastructures such as COM, CORBA, and JavaBeans provide sophisticated services such as naming, transactions and distribution for component-based applications. Some commercial tools even provide graphical ways to connect components together, allowing simple architectures to be visualized. However, these systems have poor support for structural specification of dynamically changing systems, and have no concept of communication integrity. Communication integrity can only be enforced by programmer discipline following guidelines such as the Law of Demeter [LH89] that states, "only talk to your immediate friends" in a system.

Advanced module systems such as MzScheme's Units [FF98] and ML's functors [MTH90] can be used to encapsulate components and to describe the static architecture of a system. The FoxNet project [B95] shows how functors can be used to build up a network stack architecture out of statically connected components. However, these systems do not guarantee communication integrity in the language; instead, programmers must follow a careful methodology to ensure that each module communicates only with the modules it is connected to in the architecture.

More recently, the component-oriented programming languages ComponentJ [SC00] and ACOEL [Sre01] extend a Java-like base language to explicitly support component composition. These languages can be used to express components and static architectures. However, neither language makes dynamic architectures explicit, and neither enforces communication integrity.

3. The ArchJava Language

ArchJava is designed to investigate the benefits and drawbacks of a relatively unexplored part of the ADL design space. Our approach extends a practical implementation language to incorporate architectural features and enforce communication integrity. Key benefits we hope to realize with this approach include better program understanding, reliable architectural reasoning about code, keeping architecture and code consistent as they evolve, and encouraging more developers to take advantage of software architecture. ArchJava's design also has some limitations, discussed below in section 3.6.

```
public component class Parser {
  public port in {
    provides void setInfo(Token symbol,
                          SymTabEntry e);
    requires Token nextToken()
                 throws ScanException;
  }
  public port out {
    provides SymTabEntry getInfo(Token t);
    requires void compile(AST ast);
  }

  void parse(String file) {
    Token tok = in.nextToken();
    AST ast = parseFile(tok);
    out.compile(ast);
  }

  void parseFile(Token lookahead) { ... }
  void setInfo(Token t, SymTabEntry e) { ... }
  SymTabEntry getInfo(Token t) { ... }
  ...
}
```

Figure 1. A parser component in ArchJava. The `Parser` component class uses two ports to communicate with other components in a compiler. The parser's `in` port declares a required method that requests a token from the lexical analyzer, and a provided method that initializes tokens in the symbol table. The `out` port requires a method that compiles an AST to object code, and provides a method that looks up tokens in the symbol table.

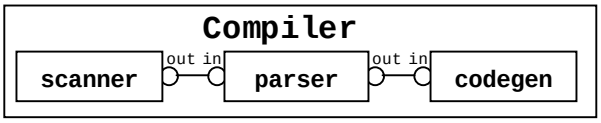
A prototype compiler for ArchJava is publicly available for download at the ArchJava web site [ACN01a]. Although in ArchJava the source code is the canonical representation of the architecture, visual representations are also important for conveying architectural structure. This paper uses hand-drawn diagrams to communicate architecture; however, we have also constructed a simple visualization tool that generates architectural diagrams automatically from ArchJava source code. In addition, we intend to provide an `archjavadoc` tool that would automatically construct graphical and textual web-based documentation for ArchJava architectures.

To allow programmers to describe software architecture, ArchJava adds new language constructs to support *components*, *connections*, and *ports*. The rest of this section describes by example how to use these constructs to express software architectures. Throughout the discussion, we show how the constructs work together to enforce communication integrity, culminating in a precise definition of communication integrity in ArchJava. Reports on the ArchJava web site [ACN01a] provide more information, including the complete language semantics and a formal proof of communication integrity in the core of ArchJava.

3.1 Components and Ports

A *component* is a special kind of object that communicates with other components in a structured way. Components are instances of *component classes*, such as the `Parser` component class in Figure 1. Component classes can inherit from other components.

A component instance communicates with external components through ports. A *port* represents a logical communication channel




```
public component class Compiler {
  component Scanner scanner;
  component Parser parser;
  component CodeGen codegen;

  connect scanner.out, parser.in;
  connect parser.out, codegen.in;

  public static void main(String args[]) {
    new Compiler().compile(args);
  }

  public void compile(String args[]) {
    // for each file in args do:
    ...parser.parse(file);...
  }
}
```

Figure 2. A graphical compiler architecture and its ArchJava representation. The `Compiler` component class contains three subcomponents—a `Scanner`, a `Parser`, and a `CodeGen`. This compiler architecture follows the well-known pipeline compiler design [GS93]. The `scanner`, `parser`, and `codegen` components are connected in a linear sequence, with the `out` port of one component connected to the `in` port of the next component.

between a component instance and one or more components that it is connected to.

Ports declare three sets of methods, specified using the **`requires`**, **`provides`**, and **`broadcasts`** keywords. *Provided* methods can be invoked by other components connected to the port. The component can invoke a disjoint set of *required* methods through the port. Each required method is implemented by a component that the port is connected to. *Broadcast* methods are just like required methods, except that they must return **`void`** and may be connected to an unbounded number of implementations.

A port specifies both the services implemented by a component and the services a component needs to do its job. Required interfaces make dependencies explicit, reducing coupling between components and promoting understanding of components in isolation. Ports also make it easier to reason about a component's communication patterns.

Each port is a first-class object that implements its required and broadcast methods, so a component can invoke these methods directly on its ports. For example, the `parse` method calls `nextToken` on the parser's `in` port. These calls will be bound to external components that implement the appropriate functionality.

3.2 Component Composition

In ArchJava, software architecture is expressed with *composite components*, which are made up of a number of subcomponents[1] connected together. Figure 2 shows how a compiler's architecture can be expressed in ArchJava. The example shows that the parser communicates with the scanner using one protocol, and with the code generator using another. The architecture also implies that the scanner does *not* communicate directly with the code generator. A primary goal of ArchJava is to ease program understanding tasks by supporting this kind of reasoning about program structure.

3.2.1 Subcomponents

A *subcomponent* is a component instance that is declared inside another component class. Components can invoke methods directly on their subcomponents. However, subcomponents cannot communicate with components external to their containing component. Thus, communication patterns among components are hierarchical.

Subcomponents are declared using a *component field*—a field of component type inside a component class, declared using the **`component`** keyword. For example, the compiler component class defines scanner, parser, and code generator subcomponents. To enable effective static reasoning about subcomponents, component fields are treated as **`protected`**, **`final`**, and not **`static`**. Subcomponents are automatically instantiated when the containing component is created—programmers can use a **`new`** expression in the field initializer in order to call a non-default constructor.

3.2.2 Connections

The **`connect`** primitive connects two or more subcomponent ports together, binding each required method to a provided method with the same name and signature. Connections are symmetric, and several connected components may require the same method. Required methods must be connected to exactly one provided method. However, invoking a broadcast method results in calls to each connected provided method with the same name and signature.

Provided methods can be implemented by forwarding invocations to subcomponents or to the required methods of another port. The semantics of method forwarding and broadcast methods are given in the language reference manual on the ArchJava web site [ACN01a]. Alternative connection semantics, such as asynchronous communication, can be implemented in ArchJava by writing custom "smart connector" components that take the place of ordinary connections in the architecture.

3.3 Communication Integrity

The compiler architecture in Figure 2 shows that while the parser communicates with the scanner and code generator, the scanner and code generator do not directly communicate with each other. If the diagram in Figure 2 represented an abstract architecture to be implemented in Java code, it might be difficult to verify the correctness of this reasoning in the implementation. For example, if the scanner obtained a reference to the code generator, it could invoke any of the code generator's methods, violating the intuition communicated by the architecture. In contrast, programmers can have confidence that an ArchJava architecture accurately represents communication between components, because the language semantics enforce communication integrity.

Communication integrity in ArchJava means that components in an architecture can only call each others' methods along declared

[1] Note: the term *subcomponent* indicates composition, whereas the term *component subclass* would indicate inheritance.

connections between ports. Each component in the architecture can use its ports to communicate with the components to which it is connected. However, a component may not directly invoke the methods of components other than its children, because this communication may not be declared in the architecture—a violation of communication integrity. We define communication integrity more precisely in section 3.5.

3.4 Dynamic Architectures

The constructs described above express architecture as a static hierarchy of interacting component instances, which is sufficient for a large class of systems. However, some system architectures require creating and connecting together a dynamically determined number of components. Furthermore, even in programs with a static architecture, the top-level component must be instantiated at the beginning of the application.

3.4.1 Dynamic Component Creation

Components can be dynamically instantiated using the same **`new`** syntax used to create ordinary objects. For example, Figure 2 shows the compiler's `main` method, which creates a `Compiler` component and calls its `invoke` method. At creation time, each component records the component instance that created it as its *parent component*. For components like `Compiler` that are instantiated outside the scope of any component instance, the parent component is **`null`**.

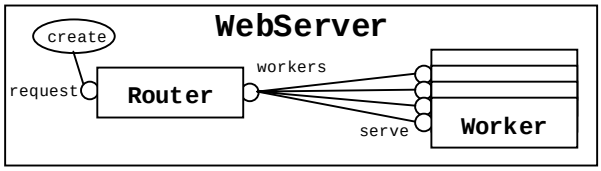
Communication integrity places restrictions on the ways in which component instances can be used. Because only a component's parent can invoke its methods directly, it is essential that typed references to subcomponents do not escape the scope of their parent component. This requirement is enforced by prohibiting component types in the ports and public interfaces of components, and prohibiting ordinary classes from declaring arrays or fields of component type. Since a component instance can still be freely passed between components as an expression of type `Object`, a `ComponentCastException` is thrown if an expression is downcast to a component type outside the scope of its parent component.

3.4.2 Connect expressions

Dynamically created components can be connected together at run time using a *connect expression*. For instance, Figure 3 shows a web server architecture where a `Router` component receives incoming HTTP requests and passes them through connections to `Worker` components that serve the request. The `requestWorker` method of the web server dynamically creates a `Worker` component and then connects its `serve` port to the `workers` port on the `Router`.

Communication integrity requires each component to explicitly document the kinds of architectural interactions that are permitted between its subcomponents. A *connection pattern* is used to describe a set of connections that can be instantiated at run time using connect expressions. For example, **`connect pattern`** `r.workers, Worker.serve` describes a set of connections between the component field `r` and dynamically created `Worker` components.

Each connect expression must match a connection pattern declared in the enclosing component. A connect expression *matches* a connection pattern if the connected ports are identical and each connected component instance is either the same component field specified in the pattern, or an instance of the type specified in the pattern. The connect expression in the web server example matches the corresponding connection pattern because




```
public component class WebServer {
  component Router r;
  connect r.request, create;
  connect pattern r.workers, Worker.serve;

  public void run() { r.listen(); }
  private port create {
    provides r.workers requestWorker() {
      Worker newWorker = new Worker();
      r.workers connection
        = connect(r.workers, newWorker.serve);
      return connection;
    }
  }
}

public component class Router {
  public port interface workers {
    requires void httpRequest(InputStream in,
                              OutputStream out);
  }
  public port request {
    requires this.workers requestWorker();
  }
  public void listen() {
    ServerSocket server = new ServerSocket(80);
    while (true) {
      Socket sock = server.accept();
      this.workers conn = main.requestWorker();
      conn.httpRequest(sock.getInputStream(),
                       sock.getOutputStream());
    }
  }
}

public component class Worker extends Thread {
  public port serve {
    provides void httpRequest(InputStream in,
                              OutputStream out) {
      this.in = in; this.out = out; start();
    }
  }
  public void run() {
    File f = getRequestedFile(in);
    sendHeaders(out);
    copyFile(f, out);
  }
  // more method & data declarations...
}
```

Figure 3. A web server architecture. The `Router` subcomponent accepts incoming HTTP requests, and pass them on to a set of `Worker` components that respond. When a request comes in, the `Router` requests a new worker connection on its `requestWorker` port. The `WebServer` then creates a new worker and connects it to the `Router`. The `Router` assigns requests to `Workers` through the `workers` port.

the `newWorker` component in the connect expression is of static type `Worker`, the same type declared in the pattern.

3.4.3 Port Interfaces

Often a single component participates in several connections using the same conceptual protocol. For example, the `Router` component in the web server communicates with several `Worker` components, each through a different connection. A *port interface* describes a port that can be instantiated several times to communicate through different connections at run time.

Each port interface defines a type that includes all of the required methods in that port. A *port interface type* combines a port's required interface with an *instance expression* that indicates which component instance the type allows access to. For example, in the `Router` component, the type **`this`**`.workers` refers to an instance of the `workers` port of the current `Router` component (in this case, **`this`** would be inferred automatically if it were omitted). The type `r.workers` refers to an instance of the `workers` port of the `r` subcomponent. This type can be used in method signatures such as `requestWorker` and local variable declarations such as `conn` in the `listen` method. Required methods can be invoked on expressions of port interface type, as shown by the call to `httpRequest` within `Router.listen`.

Port interfaces are instantiated by connect expressions. A connect expression returns a *connection object* that represents the connection. This connection object implements the port interfaces of all the connected ports. Thus, in Figure 3, the connection object `connection` implements the interfaces `Worker.serve` and `r.workers`, and can therefore be assigned to a variable of type `r.workers`.

Provided methods can obtain the connection object through which the method call was invoked using the **`sender`** keyword. The detailed semantics of **`sender`** and other language features are covered in the ArchJava language reference available on the ArchJava web site [ACN01a].

3.4.4 Removing Components and Connections

Just as Java does not provide a way to explicitly delete objects, ArchJava does not provide a way to explicitly remove components and connections. Instead, components are garbage-collected when they are no longer reachable through direct references or connections. For example, in Figure 3, a `Worker` component will be garbage collected when the reference to the original worker (`newWorker`) and the references to its connections (`connection` and `conn`) go out of scope, and the thread within `Worker` finishes execution.

3.5 Limitations of ArchJava

There are currently a number of limitations to the ArchJava approach. Our technique is presently only applicable to programs written in a single language and running on a single JVM, although the concepts may extend to a wider domain. Architectures in ArchJava are more concrete than architectures in ADLs such as Wright, restricting the ways in which a given architecture can be implemented—for example, inter-component connections must be implemented with method calls. Also, in order to focus on ensuring communication integrity, we do not yet support other types of architectural reasoning, such as reasoning about the temporal order of architectural events, or about component multiplicity.

ArchJava's definition of communication integrity supports reasoning about communication through method calls between components. Program objects can also communicate through data sharing via aliased objects, static fields, and the runtime system. However, existing ways to control communication through shared data often involve significant restrictions on programming style. Future work includes developing ways to reason about these additional communication channels while preserving expressiveness. Meanwhile, our experience (described below) suggests that rigorous reasoning about architectural control flow can aid in program understanding and evolution, even in the presence of shared data structures.

4. ArchJava Formalization

In this section, we discuss the formal definition of communication integrity and ArchJava's semantics. The next subsection defines communication integrity in ArchJava and intuitively explains how it is enforced. Subsection 5.2 gives the static and dynamic semantics of ArchFJ, a language incorporating the core features of ArchJava. Finally, subsection 5.3 outlines proofs of communication integrity, subject reduction, and progress for ArchFJ.

4.1 Definition of Communication Integrity

Communication integrity is the key property of ArchJava that ensures that the implementation does not communicate in ways that could violate reasoning about control flow in the architecture. Intuitively, communication integrity in ArchJava means that a component instance `A` may not call the methods of another component instance `B` unless `B` is `A`'s subcomponent, or `A` and `B` are sibling subcomponents of a common component instance that declares a connection or connection pattern between them.

We now precisely define communication integrity in ArchJava. Let the *execution scope* of component instance A on the run time stack, denoted *escope*(A), be any of A's executing methods and any of the object methods they transitively invoke, until another component's method is invoked.

Definition 1 [Dynamic Execution Scope]: Let `m` be an executing method with stack frame `mf`. If `m` is a component method, then `mf` ∈ *escope*(`this`). Otherwise, `mf` ∈ *escope*(*caller*(`mf`)).

Now we can define communication integrity:

Definition 2 [Communication Integrity in ArchJava]: Let :< be the subtyping relation over component classes. A program has communication integrity if, for all run time method calls to a method `m` of a component instance `b` in an executing stack frame `mf`, where `mf` ∈ *escope*(`a`), *either:*

1. `a` = `b`, *or*
2. `a` = *parent*(`b`), *or*
3. *parent*(`a`) = *parent*(`b`) ∧ "`connect [pattern] (f|t)`$_1$`.p`$_1$`,...,(f|t)`$_n$`.p`$_n$" ∈ *class*(*parent*(`a`))
   ∧ ∃i,j ∈ 1..n s.t. (*parent*(`a`).`f`$_i$ = `a` ∨ *type*(`a`)<:`t`$_i$) ∧
   (*parent*(`a`).`f`$_j$ = `b` ∨ *type*(`b`)<:`t`$_j$) ∧
   `m` ∈ *requiredmethods*(`p`$_i$) ∧
   `m` ∈ *providedmethods*(`p`$_j$)

Syntax:

```
CL      ::= class C extends C {C̄ f̄; K M̄}

CP      ::= component class P extends
            [P|Object] {C̄ f̄; K̄ M̄ R̄ X̄}

K       ::= E(C̄ f̄) {super(f̄); this.f̄ = f̄;}

M       ::= T m(T̄ x̄) { return e; }

R       ::= required T m(T̄ x̄)

X       ::= connect pattern (P̄)

e       ::= x
        |   e.f
        |   e.m(ē,_this)
        |   new C(ē)
        |   new P(ē, <fresh>, e_parent)
        |   (C)e
        |   (e.P_R)e
        |   cast(this, P, e)
        |   connect(ē, this)
        |   error
```

Figure 4. ArchFJ Syntax

4.2 Formalization as ArchFJ

We would like to use formal techniques to prove that the ArchJava language design guarantees communication integrity, and show that the language is type safe—that is, show that certain classes of errors cannot occur at run time. Unfortunately, proofs of type safety in a language like Java are extremely tedious due to the many cases involved, and to our knowledge the full Java language has never been formalized and proven type safe. Therefore, a standard technique, exemplified by Featherweight Java [IPW99], is to formalize a core language that captures the key typing issues while ignoring complicating language details.

We have modified Featherweight Java (FJ) to capture the essence of ArchJava in ArchFJ. ArchFJ makes a number of simplifications relative to ArchJava. ArchFJ leaves out ports; instead, each component class has a set of required and provided methods. Static connections and component fields are left out, as they are subsumed by dynamically created connections components. We also omit the **`sender`** keyword and broadcast methods. As in Featherweight Java (FJ), we omit interfaces. These changes make our type soundness proof shorter, but do not materially affect it otherwise.

4.2.1 Syntax

Figure 4 presents the syntax of ArchFJ. The metavariables `C` and `D` range over class names; `E` and `F` range over component and class names; `S`, `T`, and `V` range over types; `P` and `Q` range over component classes; `f` and `g` range over fields; `d` and `e` range over expressions; `l` ranges over labels generated by `<fresh>`; and `M` ranges over methods. As a shorthand, we use an overbar to represent a sequence. We assume a fixed class table *CT* mapping regular and component classes to their definitions. A program, then, is a pair (*CT*, `e`) of a class table and an expression.

ArchFJ includes the features of FJ plus a few extensions. Regular classes extend another class (which can be `Object`, a predefined class) and define a constructor `K` and a set of fields $\bar{f}$ and methods $\bar{M}$. Component classes can extend another component

Types:

```
T       ::= P
        | e.P_R
        | E
        | U(e.P̄_R)
```

Subtyping:

$$\frac{}{T <: T} \quad \text{(S-REFLEX)}$$

$$\frac{S <: T \qquad T <: V}{S <: V} \quad \text{(S-TRANS)}$$

$$\frac{P <: Q}{e_1.P_R <: e_2.Q_R} \quad \text{(S-REQUIRED)}$$

$$\frac{}{T <: \texttt{Object}} \quad \text{(S-OBJECT)}$$

$$\frac{e.P_R \in \overline{e.P_R}}{U(\overline{e.P_R}) <: e.P_R} \quad \text{(S-UNION)}$$

$$\frac{CT(E) = \texttt{[component] class E extends F \{ ... \}}}{E <: F} \quad \text{(S-EXTENDS)}$$

Figure 5. ArchFJ Types and Subtyping Rules

class, or `Object` (as in FJ, there are no interfaces). Component classes also declare a set of required methods $\bar{R}$ and a set of connection patterns $\bar{X}$ between their subcomponents.

Expressions include field lookup, method calls, object and component creation, various casts, a connect expression, and an error expression. These are extended from FJ in a few small ways:

- All method calls capture the current object `this` in an additional psuedo-argument which comes last and is not passed on to the callee.
- Components are labeled with a fresh label when they are created (labels in a method body are freshly generated when a method call is replaced with the method's body). This label allows us to reason about object identity in an otherwise functional language (assignment is not relevant to our type system or definition of communication integrity). Components also keep track of their parent, and which of their parent's component fields they were created with.
- In addition to regular casts to a class type, there are two new cast forms: one that allows casting to the required interface of a component (i.e., the set of methods the component requires), and another that allows casting to a component field type. The first cast accepts an instance expression type, while the latter cast includes an argument that captures the value of `this` in the current scope. Both arguments are used to verify the casts in the dynamic semantics.
- A connect expression conceptually creates a connection object on which components can invoke their required methods. The connect expression captures `this`, the parent object that created the connection.

Computation:

$$\frac{\mathit{fields}(\texttt{E}) = \overline{\texttt{C}}\ \overline{\texttt{f}}}{(\texttt{new E}(\overline{\texttt{e}},\ldots)).\texttt{f}_i \rightarrow \texttt{e}_i} \quad \text{(R-FIELD)}$$

$$\frac{\mathit{mbody}(\texttt{m},\texttt{C}) = (\overline{\texttt{x}}, \texttt{e}_0)}{(\texttt{new C}(\overline{\texttt{e}})).\texttt{m}(\overline{\texttt{d}},\ \texttt{d}_{\texttt{this}}) \rightarrow [\overline{\texttt{d}}/\overline{\texttt{x}},\ \texttt{new C}(\overline{\texttt{e}})/\texttt{this}]\texttt{e}_0} \quad \text{(R-INVK)}$$

$$\frac{\texttt{E} <: \texttt{C}}{(\texttt{C})(\texttt{new E}(\ldots)) \rightarrow \texttt{new E}(\ldots)} \quad \text{(R-CAST)}$$

$$\frac{\texttt{e} = \texttt{new E}(\ldots) \not<: \texttt{C} \ \vee\ \texttt{e} = \texttt{connect}(\ldots)}{(\texttt{C})(\texttt{e}) \rightarrow \texttt{error}} \quad \text{(E-CAST)}$$

$$\frac{\begin{array}{c}\texttt{e} = \texttt{new P}(\overline{\texttt{e}},\ \texttt{l},\ \texttt{e}_{\texttt{parent}}) \\ \mathit{mbody}(\texttt{m},\texttt{P}) = (\overline{\texttt{x}}, \texttt{e}_0) \\ \texttt{d}_{\texttt{this}} = \texttt{e} \vee \texttt{d}_{\texttt{this}} = \texttt{e}_{\texttt{parent}}\end{array}}{\texttt{e.m}(\overline{\texttt{d}},\ \texttt{d}_{\texttt{this}}) \rightarrow [\overline{\texttt{d}}/\overline{\texttt{x}},\ \texttt{e}/\texttt{this}]\texttt{e}_0} \quad \text{(R-PINVK)}$$

$$\frac{\texttt{e} = \texttt{new P}(\overline{\texttt{e}},\ \texttt{l},\ \texttt{e}_{\texttt{this}}) \qquad \texttt{P} <: \texttt{Q}}{\texttt{cast}(\texttt{e}_{\texttt{this}},\ \texttt{Q},\ \texttt{e}) \rightarrow \texttt{e}} \quad \text{(R-PCAST)}$$

$$\frac{\begin{array}{c}\texttt{e} = \texttt{connect}(\ldots) \ \vee \\ (\texttt{e} = \texttt{new P}(\overline{\texttt{e}},\ \texttt{l},\ \texttt{e}_{\texttt{parent}}) \\ \texttt{where } \texttt{e}_{\texttt{this}} \neq \texttt{e}_{\texttt{parent}} \vee \texttt{P} \not<: \texttt{Q})\end{array}}{\texttt{cast}(\texttt{e}_{\texttt{this}},\ \texttt{Q},\ \texttt{e}) \rightarrow \texttt{error}} \quad \text{(E-PCAST)}$$

$$\frac{\texttt{e}_{\texttt{cast}} = \texttt{new P}(\ldots) \qquad \texttt{e}_{\texttt{cast}} \in \overline{\texttt{e}} \qquad \texttt{P} <: \texttt{Q}}{\begin{array}{c}(\texttt{e}_{\texttt{cast}}.\texttt{Q}_{\texttt{R}})(\texttt{connect}(\overline{\texttt{e}},\texttt{e}_{\texttt{this}})) \\ \rightarrow \texttt{connect}(\overline{\texttt{e}},\texttt{e}_{\texttt{this}})\end{array}} \quad \text{(R-RCAST)}$$

$$\frac{\begin{array}{c}\texttt{e} = \texttt{new E}(\ldots) \ \vee \\ \texttt{e} = \texttt{connect}(\overline{\texttt{e}},\texttt{e}_{\texttt{this}}) \\ \texttt{where } \texttt{e}_{\texttt{cast}} \notin \overline{\texttt{e}} \ \vee\ \texttt{e}_{\texttt{cast}} = \texttt{new F}(\ldots) \not<: \texttt{Q}\end{array}}{(\texttt{e}_{\texttt{cast}}.\texttt{Q}_{\texttt{R}})(\texttt{e}) \rightarrow \texttt{error}} \quad \text{(E-RCAST)}$$

$$\frac{\begin{array}{c}\texttt{d}_{\texttt{this}} \in \overline{\texttt{e}} \qquad \mathit{legal}(\texttt{connect}(\overline{\texttt{e}},\texttt{e}_{\texttt{this}})) \\ \mathit{mbody}(\texttt{m},\texttt{connect}(\overline{\texttt{e}},\texttt{e}_{\texttt{this}})) = (\overline{\texttt{x}}, \texttt{e}_0, \texttt{e}_i)\end{array}}{\begin{array}{c}(\texttt{connect}(\overline{\texttt{e}},\texttt{e}_{\texttt{this}})).\texttt{m}(\overline{\texttt{d}},\ \texttt{d}_{\texttt{this}}) \\ \rightarrow [\overline{\texttt{d}}/\overline{\texttt{x}},\ \texttt{e}_i/\texttt{this}]\texttt{e}_0\end{array}} \quad \text{(R-XINVK)}$$

Figure 6. ArchFJ Reduction Rules

- We represent failed dynamic checks (such as casts) with an explicit `error` value, to make our progress theorem cleaner to state.

4.2.2 Types and Subtypes

ArchJava's types and subtyping rules are given in Figure 5. Types include class and component types (`E`), required interface types of components (`e.P`$_R$), and union types of multiple required interfaces. Subtyping of classes and components is defined by the reflexive, transitive closure of the immediate subclass relation given by the `extends` clauses in *CT*. We require that there are no cycles in the induced subtype relation. Required interface types follow the subtyping relation of components (ignoring the instance expressions, which are reasoned about separately from subtyping). Finally, every type is a subtype of `Object`, and a union is a subtype of all its member types.

4.2.3 Reduction Rules

The reduction relation, defined by the reduction rules given in Figure 6, is of the form `e`→`e′`, read "expression `e` reduces to expression `e′` in one step." We write →* for the reflexive, transitive closure of →. The only unusual reduction rule is R-XINVK, which allows method invocation on connection expressions. The *mbody* helper function does a lookup to determine the correct method body to invoke. Two error rules are defined representing casts that are not guaranteed to succeed by the type system presented below. The reduction rules can be applied at any point in an expression, so we also need appropriate congruence rules (such as if `e`→`e′` then `e.f`→`e′.f`), which we omit here. Furthermore, we assume an order of evaluation that follows Java's normal evaluation rules.

4.2.4 Typing Rules

Most of the typing rules given in Figure 7 are standard. Typing judgments are given in an environment Γ, a finite mapping from variables to types. Rule T-INVK places constraints on passing connection objects to an argument position declared with a required interface and instance expression of **`this`**, to ensure that the connection object does indeed connect the receiver object. Rule T-PNEW introduces qualified component types. Rule T-CONNECT introduces union types for connections. In addition, T-CONNECT verifies that some connection pattern in the current component matches the types of the connected objects; this will be important later for establishing that reduction cannot get stuck due to an illegal connection.

Class, method, and connection typing rules check for well-formed class definitions, and have the form "class declaration `E` is OK," and "method/connection `X` is OK in `E`." The rules for class and method typing are similar to those in FJ. In the case of component classes, the typing rule verifies that only subclasses of `Object` may define required methods—as in ArchJava, component subclasses may only inherit existing required methods from their component superclass. The connection typing rule verifies that each required method has a unique provided method with the right signature, and that every method name has only one signature across all the required methods.

We have made one significant simplification relative to FJ. We do not distinguish between upcasts, downcasts, and so-called "stupid casts" which cast one type to an unrelated one. This means that our type system does not check for "stupid casts" in the original typing derivation, as Java's type system does. However, the change shortens our presentation and proofs considerably, and the stupid casts technique from FJ can be easily applied to our system to get the same checks that are present in Java.

Expression Typing:

$$\Gamma \vdash x \in \Gamma(x) \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_0 \in C_0 \qquad \mathit{fields}(C_0) = \overline{C}\ \overline{f}}{\Gamma \vdash e_0.f_i \in C_i} \qquad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 \in T_0 \qquad \mathit{mtype}(m, T_0[, e_{this}]) = \overline{T} \to T \\ \Gamma \vdash \overline{e} \in \overline{S} \qquad \overline{S} <: \overline{T} \\ \Gamma \vdash e_{this} \in T_{this} \\ T_i = \texttt{this.}P_{Ri} \text{ implies } S_i = e_0.S_{Ri}\end{array}}{\Gamma \vdash e_0.m(\overline{e}, e_{this}) \in T[e_0/\texttt{this}]} \qquad \text{(T-INVK)}$$

$$\frac{\mathit{fields}(C) = \overline{D}\ \overline{f} \qquad \Gamma \vdash \overline{e} \in \overline{C} \qquad \overline{C} <: \overline{D}}{\Gamma \vdash \texttt{new } C(\overline{e}) \in C} \qquad \text{(T-NEW)}$$

$$\frac{\mathit{fields}(P) = \overline{D}\ \overline{f} \qquad \Gamma \vdash \overline{e} \in \overline{C} \qquad \overline{C} <: \overline{D} \qquad \Gamma \vdash e_p \in T_p}{\Gamma \vdash \texttt{new } P(\overline{e}, \texttt{<fresh>}, e_p) \in P} \qquad \text{(T-PNEW)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_{this} \in P_{this} \qquad \Gamma \vdash \overline{e} \in \overline{P} \qquad \overline{P} <: \overline{Q} \\ \texttt{connect pattern}(\overline{Q}) \in \mathit{connects}(P_{this})\end{array}}{\Gamma \vdash \texttt{connect}(\overline{e}, e_{this}) \in U(\overline{e.P_R})} \qquad \text{(T-CONNECT)}$$

$$\frac{\Gamma \vdash e \in T_0}{\Gamma \vdash ([e_{cast}.P_R | C])e \in [e_{cast}.P_R | C]} \qquad \text{(T-CAST)}$$

$$\frac{\Gamma \vdash e \in T \qquad \Gamma \vdash e_{this} \in P_{this}}{\Gamma \vdash \texttt{cast}(e_{this}, Q, e) \in Q} \qquad \text{(T-PCAST)}$$

Class Typing:

$$\frac{\begin{array}{c}K = F(\overline{D}\ \overline{g}, \overline{C}\ \overline{f})\{\texttt{super}(\overline{g}); \texttt{this.}\overline{f} = \overline{f};\} \\ \mathit{fields}(E) = \overline{D}\ \overline{g} \qquad \overline{M}[, \overline{X}] \text{ OK IN } P \\ E = \texttt{Object} \vee \#(\overline{R}) = 0 \\ [\texttt{component}]\ \texttt{class } F \texttt{ extends } E \\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\ [\overline{R}\ \overline{X}]\}\ \text{OK}\end{array}}{} \qquad \text{(T-CLASS)}$$

Method Typing:

$$\frac{\begin{array}{c}\overline{x}:\overline{T}, \texttt{this}:E \vdash e \in S \qquad S <: T \\ CT(E) = [\texttt{component}]\ \texttt{class } E \texttt{ extends } F\ \{..\} \\ \mathit{override}(m, F, \overline{T} \to T) \qquad T, \overline{T} \text{ not components}\end{array}}{T\ m(\overline{T}\ \overline{x})\{\ \texttt{return } e;\ \}\ \text{OK in } E} \qquad \text{(T-METH)}$$

Connection Typing:

$$\frac{\begin{array}{c}\forall i\ \ \mathit{mtype}(m, P_{Ri}) = \overline{T} \to T \text{ implies} \\ \exists j \neq i\ \ s.t.\ \mathit{mtype}(m, P_j) = \overline{T} \to T \\ \wedge\ \forall k \neq j\ \ \mathit{mtype}(m, P_k)\ \ not\ \ defined \\ \forall i, j\ \ \mathit{mtype}(m, P_{Ri}) = \overline{T} \to T \wedge \mathit{mtype}(m, P_{Rj}) = \overline{S} \to S \\ \text{implies } \overline{T} = \overline{S} \wedge T = S\end{array}}{\texttt{connect pattern}(\overline{P})\ \text{OK IN } Q} \qquad \text{(T-X)}$$

Figure 7. ArchFJ Static Semantics

Field lookup:

$$\mathit{fields}(\texttt{Object}) = \bullet$$

$$\frac{\begin{array}{c}CT(E) = [\texttt{component}]\ \texttt{class } E \texttt{ extends } F \\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\ [\overline{R}\ \overline{X}]\} \\ \mathit{fields}(F) = \overline{D}\ \overline{g}\end{array}}{\mathit{fields}(E) = \overline{D}\ \overline{g}, \overline{C}\ \overline{f}}$$

Connection lookup:

$$\mathit{connects}(\texttt{Object}) = \bullet$$

$$\frac{\begin{array}{c}CT(P) = \texttt{component class } P \texttt{ extends } E \\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\ \overline{R}\ \overline{X}\} \\ \mathit{connects}(E) = \overline{X_0}\end{array}}{\mathit{connects}(P) = \overline{X_0}, \overline{X}}$$

Method type lookup:

$$\frac{\begin{array}{c}CT(E) = [\texttt{component}]\ \texttt{class } E \texttt{ extends } F\ \{..\overline{M}..\} \\ T\ m\ (\overline{T}\ \overline{x})\ \{\texttt{return } e;\} \in \overline{M}\end{array}}{\mathit{mtype}(m, E) = \overline{T} \to T}$$

$$\frac{\begin{array}{c}CT(E) = [\texttt{component}]\ \texttt{class } E \texttt{ extends } F\ \{..\overline{M}..\} \\ m \text{ is not defined in } \overline{M}\end{array}}{\mathit{mtype}(m, E) = \mathit{mtype}(m, F)}$$

$$\frac{\begin{array}{c}CT(P) = \texttt{component class } P \texttt{ extends } E\ \{..\overline{R}..\} \\ \texttt{required } T\ m\ (\overline{T}\ \overline{x}) \in \overline{R}\end{array}}{\mathit{mtype}(m, P) = \overline{T} \to T}$$

$$\frac{\begin{array}{c}CT(P) = \texttt{component class } P \texttt{ extends } E\ \{..\overline{R}..\} \\ m \text{ is not declared in } \overline{R}\end{array}}{\mathit{mtype}(m, e.P_R) = \mathit{mtype}(m, e.E_R)}$$

$$\frac{e_{this} = e_i \qquad \mathit{mtype}(m, e_i.P_{Ri}) = \overline{T} \to T}{\mathit{mtype}(m, U(\overline{e.P_R}), e_{this}) = \overline{T} \to T}$$

Figure 8. ArchFJ Auxiliary Definitions

4.2.5 Auxiliary Definitions

Most of the auxiliary definitions shown in Figures 8 and 9 are straightforward and are taken from FJ. The connection typing rule verifies that the passed-in **`this`** expression is one of the instance expressions in the union type. The connection method lookup rule chooses the component *i* providing the method with *mtype*, based on the static types in the original connection declaration. It is guaranteed to choose a unique component because the connection typing rule implies that *mtype* is only defined for one of the types in the connection. It then picks the actual method body dynamically using the usual *mbody* rule. Finally, it returns the expression to be passed as **`this`** in the method call.

The *legal* rule checks that a connect expression corresponds to a connection pattern. It also verifies that the connect expression was created inside the parent component of each sibling.

Method body lookup:

$$\frac{CT(\mathtt{E}) = [\mathtt{component}]\ \mathtt{class\ E\ extends\ F}\ \{\ldots \overline{\mathtt{M}} \ldots\} \qquad \mathtt{C\ m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\mathtt{return\ e;}\} \in \overline{\mathtt{M}}}{mbody(\mathtt{m}, \mathtt{E}) = (\overline{\mathtt{x}}, \mathtt{e})}$$

$$\frac{CT(\mathtt{E}) = [\mathtt{component}]\ \mathtt{class\ E\ extends\ F}\ \{\ldots \overline{\mathtt{M}} \ldots\} \qquad \mathtt{m}\ \text{is not defined in}\ \overline{\mathtt{M}}}{mbody(\mathtt{m}, \mathtt{E}) = mbody(\mathtt{m}, \mathtt{F})}$$

$$\frac{\mathtt{e_{this}} = \mathtt{new}\ \mathtt{P_{this}}(\ldots) \qquad \overline{\mathtt{e}} = \mathtt{new}\ \overline{\mathtt{Q}}(\ldots) \qquad \mathtt{connect\ pattern}(\overline{\mathtt{P}}) \in connects(\mathtt{P_{this}}) \qquad \overline{\mathtt{Q}} <: \overline{\mathtt{P}} \qquad mtype(\mathtt{m}, \mathtt{P}_i) = \overline{\mathtt{T}} \to \mathtt{T} \qquad mbody(\mathtt{m}, \mathtt{Q}_i) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{mbody(\mathtt{m}, \mathtt{connect}(\overline{\mathtt{e}}, \mathtt{e_{this}})) = (\overline{\mathtt{x}}, \mathtt{e}_0, \mathtt{e}_i)}$$

Legal Connections:

$$\frac{\mathtt{e_{this}} = \mathtt{new}\ \mathtt{P_{this}}(\ldots) \qquad \mathtt{e}_i = \mathtt{new}\ \mathtt{Q}_i(\overline{\mathtt{d}_i}, \mathtt{l}_i, \mathtt{e}_{pi}) \qquad \mathtt{connect\ pattern}(\overline{\mathtt{P}}) \in connects(\mathtt{P_{this}}) \qquad \overline{\mathtt{Q}} <: \overline{\mathtt{P}} \qquad \forall \mathtt{i}\ \mathtt{e}_i = \mathtt{e_{this}} \vee \mathtt{e}_{pi} = \mathtt{e_{this}}}{legal(\mathtt{connect}(\overline{\mathtt{e}}, \mathtt{e_{this}}))}$$

Valid method overriding:

$$\frac{mtype(\mathtt{m},\ \mathtt{E}) = \overline{\mathtt{T}} \to \mathtt{T}_0,\ \text{implies}\ \overline{\mathtt{S}} = \overline{\mathtt{T}}\ \text{and}\ \mathtt{S}_0 = \mathtt{T}_0}{override(\mathtt{m},\ \mathtt{E},\ \overline{\mathtt{S}} \to \mathtt{S}_0)}$$

Figure 9. More Auxiliary Definitions

4.3 Theorems

We state three main theorems: communication integrity, subject reduction, and progress. Subject reduction and progress together imply that the ArchJava type system is sound. First, the reduction rules ensure communication integrity:

Theorem [Communication Integrity in ArchFJ]:

1. For all direct method invocations on a component `P` that succeed, either `P` or `P`'s parent component is the current component `this`.
2. For all method invocations on a connection that succeed, the current component `P` is part of the connection, `P` and the component `Q` being invoked either have the same parent or one is the parent of the other, and the parent `P'` declared a connection pattern between `P` and `Q`.

Proof: Part 1 of communication integrity is ensured by the precondition $\mathtt{d_{this}} = \mathtt{e} \vee \mathtt{d_{this}} = \mathtt{e_{parent}}$ of R-PINVK. Part 2 of communication integrity is ensured by the precondition $\mathtt{d_{this}} \in \overline{\mathtt{e}}$ of R-XINVK as well as the definition of *legal*.

The presentation of our Subject Reduction and Progress theorems is adapted from FJ [IPW99].

Theorem [Subject Reduction]: If $\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0$ and $\mathtt{e}_0 \to \mathtt{e}_1$, then $\Gamma \vdash \mathtt{e}_1 \in \mathtt{T}_1$ for some $\mathtt{T}_1 <: \mathtt{T}_0$.

Proof sketch: The main property required is the following term-substitution lemma:

Lemma 1 [Term Substitution]: If $\Gamma, \overline{\mathtt{x}}:\overline{\mathtt{S}_0} \vdash \mathtt{e} \in \mathtt{T}_0$ and $\Gamma \vdash \overline{\mathtt{d}} \in \overline{\mathtt{S}_1}$ where $\overline{\mathtt{S}_1} <: \overline{\mathtt{S}_0}$, then $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} \in \mathtt{T}_1$ for some $\mathtt{T}_1 <: \mathtt{T}_0$.

Lemma 1 is proved by induction on the derivation of $\Gamma, \overline{\mathtt{x}}:\overline{\mathtt{S}_0} \vdash \mathtt{e} \in \mathtt{T}_0$.

The theorem itself can then be proved by induction on the derivation of $\mathtt{e}_0 \to \mathtt{e}_1$, with a case analysis on the last rule used. Lemma 1 is useful in many of the steps, and especially for the congruence rules.

The only tricky case is to show that the preconditions of T-INVK still hold after a reduction step. This can be shown based on a case analysis on the introduction of required component types (T-INVK, T-CONNECT, and T-CAST), and a lemma that term substitution preserves the required relationships among instance expressions.

Theorem [Progress]: Suppose `e` is a well-typed expression. Then either `e` has an `error` subexpression, or `e` is a *value* made up of only new and connect expressions, or $\mathtt{e} \to \mathtt{e}'$.

Proof sketch: The theorem is proved by induction on the derivation of the reduction of `e`. For each reduction rule, we show that any valid typing for the subexpressions in the left-hand-side, together with the assumption of progress for the subexpression, implies the preconditions for the reduction rule. In most cases the implication is clear, but two interesting lemmas are necessary for rules R-PINVK and R-XINVK, respectively.

Lemma 2 [An expression of component type reduces to `this` or a direct child component of `this`]:

Consider an expression $\mathtt{e_t.m}(\overline{\mathtt{e}}, \ldots)$ where $\mathtt{e_t} = \mathtt{new\ E}(\ldots)$, $mbody(\mathtt{m}, \mathtt{E}) = (\mathtt{x}, \mathtt{e}_0)$, and $\mathtt{e}_0$ has a subexpression $\mathtt{e}_1\mathtt{.m}(\overline{\mathtt{e}_1}, \mathtt{this})$. If $\overline{\mathtt{x}}:\overline{\mathtt{T}}, \mathtt{this:E} \vdash \mathtt{e}_1 \in \mathtt{P}$ and $[\mathtt{d/x}, \mathtt{e_t/this}]\mathtt{e}_1 \to^* \mathtt{new\ Q}(\ldots, \mathtt{e_{parent}})$, then either $\mathtt{e}_1 = \mathtt{this}$ or $\mathtt{e_{parent}} \to^* \mathtt{e_t}$.

This lemma can be proved by a case analysis of the last typing rule used in the typing derivation of $\mathtt{e}_1$. There are only three rules that result in a component type: T-VAR, T-PNEW, and T-PCAST (methods cannot return component type, by the well-formed method rule). The T-VAR rule gives a component type to a variable `x`, but the only way a component type can be introduced into Γ is by the component method typing rule, with $\mathtt{x} = \mathtt{this}$. If the component type was introduced in T-PNEW, $\mathtt{e}_1 = \mathtt{new\ Q}(\ldots, \mathtt{this})$ and so $\mathtt{e_{parent}} = \mathtt{e_t}$. If the component type came from T-PCAST, $\mathtt{e}_1$ must be of the form $\mathtt{cast(this,\ P,\ new\ Q}(\ldots, \mathtt{e_{parent}}))$, and so the derivation of $[\mathtt{d/x}, \mathtt{e_t/this}]\mathtt{e}_1 \to^* \mathtt{new\ Q}(\ldots, \mathtt{e_{parent}})$ must include a reduction rule R-PCAST which verifies that $\mathtt{e_{parent}} = \mathtt{e_t}$ in the final expression.

Lemma 3 [Well-typed connection expressions are legal]: If $\Gamma \vdash \mathtt{connect(e, e_{this})} \in \mathtt{T}$ then $legal(\mathtt{connect(e, e_{this})})$.

The typing rule T-CONNECT, together with Lemma 2, demonstrates that all the required properties in *legal* hold.

5. Evaluation

We have written a prototype compiler for ArchJava, which is available for download from the ArchJava web site [ACN01a]. In order to determine whether the ArchJava language enables effective component-oriented programming, we undertook a case study applying ArchJava to Aphyds, a 12,000-line circuit design application written in Java.

Results from our case study [ACN01b] indicate that for this program, the developer's architecture can be expressed in ArchJava with relatively little effort (about 30 programmer hours). The resulting architecture yields insight into the program's communication patterns, and may be useful in eliminating software defects.

6. Conclusion and Future Work

ArchJava allows programmers to effectively express software architecture and then seamlessly fill in the implementation with Java code. This paper has motivated and outlined a language design integrating architecture and implementation, and proved type soundness and communication integrity in a formalization of ArchJava. At every stage of development and evolution, ArchJava enforces communication integrity, ensuring that the implementation conforms to the specified architecture. Thus, ArchJava helps to promote effective architecture-based design, implementation, program understanding, and evolution.

In future work, we intend to extend the case study to larger programs, to see if ArchJava can be successfully applied to programs of 100,000 lines and up. We will also investigate extending the language design to enable more advanced reasoning about component-based systems, including temporal ordering constraints on component method invocations and constraints on data sharing between components.

7. Acknowledgements

We would like to thank David Notkin, Todd Millstein, Vassily Litvinov, Vibha Sazawal, Matthai Philipose, and the anonymous reviewers for their comments and suggestions. This work was supported in part by NSF grant CCR-9970986, NSF Young Investigator Award CCR-945776, and gifts from Sun Microsystems and IBM.

8. References

[ACN01a] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava web site. http://www.cs.washington.edu/homes/jonal/archjava/

[ACN01b] Jonathan Aldrich, Craig Chambers, and David Notkin. Component-Oriented Programming in ArchJava. In Proceedings of the OOPSLA '01 Workshop on Language Mechanisms for Programming Software Components, July 2001. Available at http://www.cs.washington.edu/homes/jonal/archjava/

[AG97] Robert Allen and David Garlan. *A Formal Basis for Architectural Connection.* ACM Transactions on Software Engineering and Methodology, 6(3):213---249, July 1997.

[B95] Jeremy Buhler. The Fox Project. ACM Crossroads 2.1, September 1995.

[FF98] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In PLDI'98 - ACM Conf. on Programming Language Design and Implementation, pages 236--248, 1998.

[GS93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In Advances in Software Engineering and Knowledge Engineering, I (Ambriola V, Tortora G, Eds.) World Scientific Publishing Company, 1993.

[IPW99] Atsushi Igarishi, Benjamin Pierce, and Philip Wadler. *Featherweight Java: A minimal core calculus for Java and GJ.* In Proceedings of ACM Conference on Object Oriented Languages and Systems, November 1999.

[LH89] Karl Lieberherr and Ian Holland. *Assuring Good Style for Object-Oriented Programs*. IEEE Software, Sept 1989.

[LV95] D.C. Luckham, J. Vera. An Event Based Architecture Definition Language. IEEE Transactions on Software Engineering Vol. 21, No 9, September 1995.

[MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software Reflexion Models: Bridging the Gap Between Design and Implementation. To appear in *IEEE Transactions on Software Engineering*, 2001.

[MQR95] M. Moriconi, X. Qian, A.A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, Vol. 21, No 4, April 1995.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70-93, January 2000.

[MTH90] R. Milner, M. Tofte, and R. Harper. The Definition of Standard ML. The MIT Press, Cambridge, Massachussetts, 1990.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes, 17:40--52, October 1992.

[SC00] J. C. Seco and L. Caires. A Basic Model of Typed Components. Proc. European Conference on Object-Oriented Programming, 2000.

[SDK+95] M. Shaw, R. DeLine, V. Klein, T.L. Ross, D.M. Young, G. Zelesnik. *Abstractions for Software Architecture and Tools to Support Them.* IEEE Transactions on Software Engineering, Vol. 21, No 4, April 95.

[Sre01] V. C. Sreedhar. ACOEL: A Component-Oriented Extensional Language. Unpublished manuscript, July 2001.

Using Message Sequence Charts for Component-Based Formal Verification*

Bernd Finkbeiner
Computer Science Department
Stanford University
Stanford, CA 94305, USA
finkbein@cs.stanford.edu

Ingolf Krüger
Department of Informatics
Technical University of Munich
80290 Munich, Germany
kruegeri@in.tum.de

ABSTRACT

Message sequence charts (MSCs) are are a popular tool to informally explain the behavioral embedding of a component in its environment. In this paper we investigate if MSCs can also serve as a specification and reasoning technique for the composition of systems from components. We identify three challenges: (1) *Semantic Duality:* MSCs express global coordination properties as well as requirements on individual components for their correct participation in an interaction pattern. We show that the two semantics do not always agree and suggest syntactic constraints that ensure the represented property can be decomposed. (2) *Completeness:* we define a decompositional proof rule based on MSCs. We show that the rule is incomplete and discuss reasons and possible improvements. (3) *Compositionality:* in component-oriented system development, the different parts of the system are designed independently of each other. We suggest a composition operator for MSC specifications of such components and outline differences to operators used for the composition of scenarios.

1. INTRODUCTION

Component-based software development shortens the design process by allowing the software engineer to use black-box components. A prerequisite for the composition of systems from components is adequate information about their interface.

Here, with the notion of interface we associate not only the signatures of the operations a component offers to its environment; although popular, this interface notion offers much too little information to be of value in a more rigorous approach to software development. Instead, we add the component's observable behaviors as part of our interface notion. Approaches at component-oriented system development, such as ROOM [32] and UML-RT [33], have a similar but more informal interface notion. Our goal is to exploit the extra information our interface notion offers during system verification in the context of pragmatic and industrially accepted engineering approaches.

It is easy to describe the signature part of a component's interface. But how to capture, represent, and systematically develop the behavioral aspects of an interface? Message Sequence Charts (MSCs) have gained wide acceptance for scenario-based specifications of component interaction behavior (see, for instance, [20, 8, 31, 6, 29]). Due to their intuitive notation MSCs have proven useful as a communication tool between customers and developers, thus helping to reduce misunderstandings from the very early development stages.

In this paper we investigate if MSCs can also serve as a specification and reasoning technique for the composition of systems from components. When used in a formal setting, MSCs could provide the link between the verification of individual components and the correctness proof for the complete system.

MSCs capture the communication or collaboration among a set of components. Typically, an MSC consists of a set of axes, each labeled with the name of a component. An axis represents part of the existence of its corresponding component. Arrows in MSCs denote communication. An arrow starts at the axis of the sender or initiator of the communication; the axis at which the head of the arrow ends designates the communication's recipient or destination. Intuitively, the order in which the arrows occur within an MSC defines sequences of interaction among the depicted components. Figure 1 shows an MSC that displays a sequence of interactions among three components in a simple communication protocol.

The information that an MSC captures includes several structural and behavioral aspects. The separate axes indicate logical or physical component distribution. The presence of an arrow between two axes indicates the existence of a communication link between the corresponding components, as well as the occurrence of interaction itself. Finally, some MSC dialects, such as [31, 22, 10], allow the developer also to indicate state changes of individual components contained in an MSC. Composite MSCs (C-MSCs) extend the MSC language with additional structure, such as

*This research was supported in part by NSF grant CCR-99-00984-001, by ARO grants DAAG55-98-1-0471 and DAAD19-01-1-0723, by ARPA/AF contracts F33615-00-C-1693 and F33615-99-C-3014 and by the Deutsche Forschungsgemeinschaft within the priority program "SoftSpez" (SPP 1064) under project name *InTime*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA 2001 Workshop on Specification and Verification of Component-Based Systems Oct. 2001 Tampa, FL, USA
Copyright 2001 Bernd Finkbeiner and Ingolf Krüger.

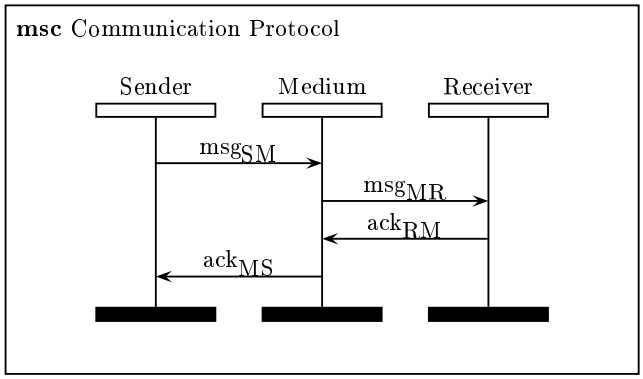


Figure 1: Basic MSC specifying a communication protocol.

loops, alternatives, or sequential composition. The example in Figure 2 shows a communication protocol with two alternative outcomes: the "Receiver" process may report success ("ack") or failure ("fail").

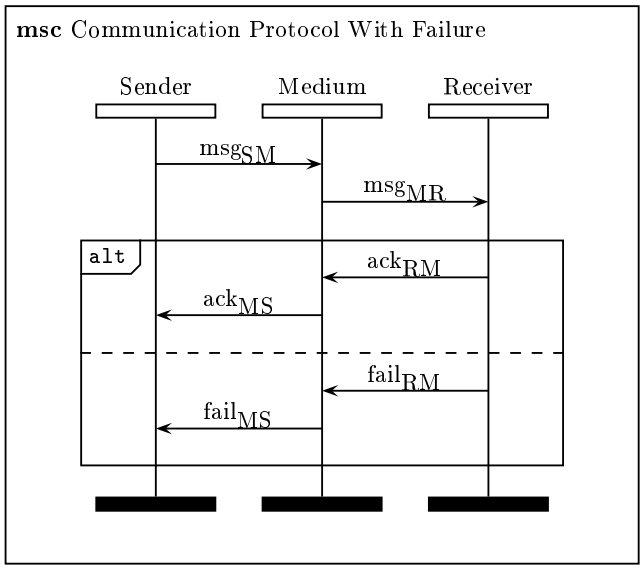


Figure 2: C-MSC specifying a communication protocol with failure.

MSCs describe the embedding of individual components into their environments, i.e., how components cooperate to achieve a certain task in a distributed system. They hide most of the details of local state changes of individual components and, instead, convey the "big picture" of the collaborations among the referenced components. This abstract, and integrated view on system behavior has resulted in the application of MSCs for use case specifications, particularly in object-oriented analysis and design, as well as for test-case specifications and simulation-run visualizations, especially in tools for telecommunication and embedded systems. In a sense, MSCs represent projections of the overall system behavior onto particular *services* or tasks of the system; automata, another popular description technique for behavioral aspects, typically represent projections of the overall system behavior onto individual components.

Of increasing importance is the use of MSCs as a description technique for complete behavior patterns, instead of for mere exemplary interaction scenarios, because this facilitates the MSC's seamless integration into an overall development process for distributed systems (cf. [24, 22], and the references contained therein). This is a particularly useful approach for the specification for component interfaces (as opposed to complete component behavior), because of the typically limited size of the corresponding interaction protocols.

An MSC describes both the global system behavior, and how each individual component should perform to establish the desired result. In this paper we explore how this duality can be used in the verification of distributed systems. Formally, MSCs here assume the role of a decompositional proof rule. They decompose a global specification into local specifications which are satisfied by the individual components. In the verification literature, this technique is known as the *assumption-commitment* paradigm (cf. [12, 7]): the environment of a component is specified only to the extent that is necessary so that the component can guarantee its correct operation; any implementation details about the environment are left unspecified at this point. Assumption-commitment reasoning shifts the burden of formal verification from the system-level down to the component-level.

While the motivation for MSC-based descriptions and assumption-commitment specifications is similar, there is a gap in the degree of formalization offered that must be closed before MSCs can be used as a formal reasoning tool. In the following sections we will discuss three challenges.

1. *Semantic Duality:* MSCs express global coordination properties as well as requirements on individual components for their correct participation in an interaction pattern. We say an MSC has the *decomposition property* if the two semantics agree. The decomposition property is necessary for MSCs used in component oriented proofs, since we infer the validity of a global property from the validity of local properties. Unfortunately, not all MSCs have the decomposition property. We suggest syntactic constraints that ensure the represented property can be decomposed.

2. *Completeness:* For MSCs that have the decomposition property, we can formulate a decompositional proof rule that reduces the proof of a global property to the verification of local component properties. The rule is incomplete: not all valid system properties can actually be proven with the rule. We discuss reasons for the incompleteness and possible improvements.

3. *Compositionality:* In component-oriented system development, the different parts of the system are designed independently of each other. Correspondingly, their MSC specifications are unlikely to be identical and we need a process to resolve differences. We suggest such a composition operator and discuss differences to operators used for the composition of scenarios.

In the following section we introduce MSCs formally, and give a simple semantics based on ω-automata; we address the three challenges in Sections 3, 4 and 5.

2. MESSAGE SEQUENCE CHARTS

MSCs have a wide spectrum of applications in the development process, ranging from analysis to implementation support. Correspondingly, many different interpretations have been proposed in the literature. An MSC language supporting requirements capture and analysis of interaction patterns requires a very liberal underlying semantics definition; it should, for instance, not exclude other possible interaction patterns too early in the development process.

In this paper, we focus on the verification of universal properties. Correspondingly, we are interested in the exclusion of undesired behaviors. In this section we describe a semantics that achieves this by identifying all possible interaction patterns: behaviors other than the ones that are explicitly depicted will be excluded.

We base our semantics on ω-automata. The ω-regular languages form a particularly useful class since it is closed under complementation and intersection, it is decidable and in fact well-supported by verification algorithms (cf. [13, 18]). We will work with a simplified definition of basic message sequence charts. Similar definitions appear in [2, 3]; a semantics for a richer dialect is given in [22].

DEFINITION 1 (MESSAGE SEQUENCE CHARTS). *A (basic)* message sequence chart *(MSC)* $M = \langle P, \mathcal{M}, E, C, O\rangle$ *is a labeled graph with the following components:*

- processes: *a finite set* P *of processes or components;*
- messages: *a finite set* $\mathcal{M}$ *of messages, we assume that the messages can be partitioned according to their sender* $\mathcal{M} = \bigcup_{p\in P} S_p$ *and according to their recipient* $\mathcal{M} = \bigcup_{p\in P} R_p$*; let* $\mathcal{M}_p$ *denote the union* $S_p \cup R_p$*;*
- events: *a finite set* E *of events, each process* $p \in P$ *has a single initial event* e_p*;*
- interprocess edges: *a set of directed edges connecting events, labeled by messages between processes* $C \subseteq E \times \mathcal{M} \times E$*; we assume that each event appears on exactly one edge;*
- intraprocess edges: *a function* $O : E \to E \cup \{\bot\}$ *connecting events in the order in which they are displayed.* $\bot$ *indicates that there is no subsequent event.*

The ω-regular languages are recognized by ω-automata. Different types of ω-automata are distinguished according to their acceptance conditions, in the following we will use the fairly simple Büchi acceptance condition on the transitions (for a survey on ω-automata see [36]).

DEFINITION 2 (BÜCHI AUTOMATON). *A Büchi automaton is a tuple* $\mathcal{A} = \langle N, \Sigma, I, T, \mathcal{F}\rangle$ *with*

- nodes: *a finite set* N *of nodes,*
- input alphabet: *a finite set* Σ *of input symbols,*
- initial nodes: *a subset* $I \subset N$*,*
- transitions: *a finite set* $T \subseteq N \times \Sigma \times N$ *of labeled edges connecting nodes,*
- acceptance condition: *a subset* $\mathcal{F} \subseteq T$*.*

Acceptance of an input sequence is determined as follows.

DEFINITION 3 (ACCEPTING PATHS). *For an infinite sequence of input symbols* $\sigma : s_0, s_1, s_2, \ldots$ *an infinite sequence of transitions* $\pi = (n_0, s_0, n_1), (n_1, s_1, n_2), \ldots$ *is a* path *of* $\mathcal{A}$ *on* σ *if* $n_0 \in I$*. A path* π *is accepting if some edge in* $\mathcal{F}$ *occurs infinitely often in* π*.*

DEFINITION 4 (LANGUAGE). *The language* $\mathrm{L}(\mathcal{A})$ *of an automaton* $\mathcal{A}$ *is the set of all infinite sequences* σ *of input symbols that have an accepting path in* $\mathcal{A}$*.*

We represent an MSC as an automaton by using sets of "simultaneously active" events as states. There is a transition for each interprocess edge and an additional τ-transition that simulates an internal computation step. We assume zero-delay communication: the transitions respect the partial order on the intraprocess edges as well as the synchronization introduced by the interprocess edges.

DEFINITION 5 (GLOBAL SEMANTICS). *Given a basic MSC* $M = \langle P, \mathcal{M}, E, C, O\rangle$ *the* global semantics *is given as the* associated global automaton $\mathcal{A} = \langle N, \Sigma, I, T, \mathcal{F}\rangle$ *with*

- $N = 2^{E\cup\{\bot\}}$,
- $\Sigma = \mathcal{M} \cup \{\tau\}$,
- $I = \{\ \{e_p \mid p \in P\}\ \}$,
- T *contains a set of self-loops* $\{(n, \tau, n) \mid n \in N\}$ *and a set of transitions reacting to messages:* $\{(n_1, s, n_2)\}$ *such that*
  - *for each* $e_1 \in n_1$ *one of the following holds:*
    - $e_1 \in n_2$ *and there is no event* e' *with* $(e_1, s, e') \in C$ *or* $(e', s, e_1) \in C$*,*
    - $O(e_1) \in n_2$ *and there is an event* $e' \in n_1$ *with* $(e_1, s, e') \in C$ *or* $(e', s, e_1) \in C$*, and*
  - *for each* $e_2 \in n_2$ *one of the following holds:*
    - $e_2 \in n_1$ *and there is no event* $e' \in E$ *with* $(e_1, s, e') \in C$ *or* $(e', s, e_1) \in C$
    - *there is an event* $e_1 \in n_1$ *such that* $e_2 = O(e_1)$ *and there is an event* e' *with* $(e_1, s, e') \in C$ *or* $(e', s, e_1) \in C$*,*
- $\mathcal{F} = \{(\{\bot\}, \tau, \{\bot\})\}$.

Figure 3a shows the automaton associated with the MSC from Figure 1. Accepting transitions are depicted with double edges. The semantics of C-MSC constructs can be described as the corresponding transformations on the automata. Here we restrict ourselves to sequential composition, nondeterministic alternatives, and finite as well as infinite loops; these language constructs suffice for the purposes of this paper. We refer the reader to [22] for similar constructions for almost all of the MSC-96 standard [20].

DEFINITION 6 (AUTOMATA TRANSFORMATIONS). *For two Büchi automata* $\mathcal{A}_1 = \langle N_1, \Sigma, I_1, T_1, \mathcal{F}_1\rangle$ *and* $\mathcal{A}_2 = \langle N_2, \Sigma, I_2, T_2, \mathcal{F}_2\rangle$ *we define the result* $\langle N', \Sigma, I', T', \mathcal{F}'\rangle$ *of the following transformations:*

- *sequential composition* $\mathcal{A}_1;\mathcal{A}_2$*:*
  - $N' = N_1 \cup N_2$,
  - $I' = I_1$,

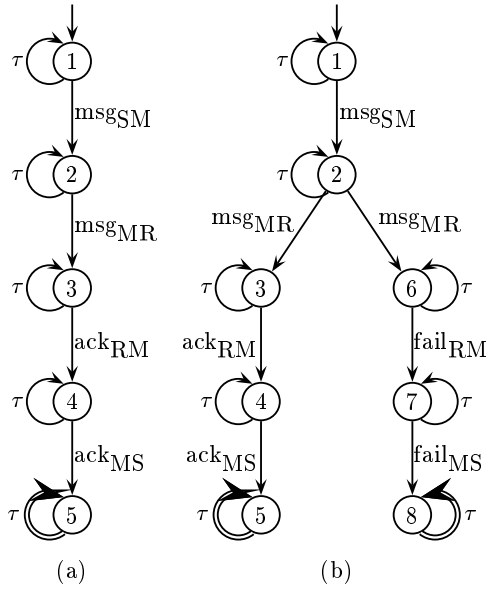


Figure 3: Automata associated with (a) the MSC from Figure 1 and (b) the C-MSC from Figure 2.

  - $T' = T_1 - \{(n_1, s, \{\bot\}) \in T_1\}$
    $\cup\ \{(n_1, s, n_2) \mid (n_1, s, \{\bot\}) \in T_1, n_2 \in I_2\}$
    $\cup\ T_2$
  - $\mathcal{F}' = \mathcal{F}_1 \cup \mathcal{F}_2$,

- *alternative* $\texttt{alt}(\mathcal{A}_1, \mathcal{A}_2)$*:*
  - $N' = N_1 \cup N_2$,
  - $I' = I_1 \cup I_2$,
  - $T' = T_1 \cup T_2$,
  - $\mathcal{F}' = \mathcal{F}_1 \cup \mathcal{F}_2$,

- *finite loop* $\texttt{loop}(\mathcal{A}_1^*)$*:*
  - $N' = N_1$,
  - $I' = I_1 \cup \{\{\bot\}\}$,
  - $T' = T_1 \cup \{(n, s, m) \mid (n, s, \{\bot\}) \in T_1, m \in I_1\}$,
  - $\mathcal{F}' = \mathcal{F}_1$,

- *infinite loop* $\texttt{loop}(\mathcal{A}_1^\omega)$*:*
  - $N' = N_1$,
  - $I' = I_1$,
  - $T' = \{(n, s, m) \mid (n, s, m) \in T_1, m \neq \{\bot\}\}$
    $\cup\ \{(n, s, m) \mid (n, s, \{\bot\}) \in T_1, m \in I_1\}$,
  - $\mathcal{F}' = \{(n, s, m) \mid (n, s, m) \in \mathcal{F}_1, m \neq \{\bot\}\}$
    $\cup\ \{(n, s, m) \mid (n, s, \{\bot\}) \in T_1, m \in I_1\}$.

As an example consider again the C-MSC from Figure 2: its associated automaton is shown in Figure 3b. Note that both the paths that stay in node 5, and the paths that stay in node 8 are accepting.

Definition 7 (Global Language). *A (finite or infinite) sequence of messages σ is accepted by an MSC M if there is an infinite sequence σ' of symbols in $\mathcal{M} \cup \{\tau\}$ such that σ' with all occurrences of τ removed is equal to σ and σ' is accepted by the automaton associated with M.*

3. CHALLENGE 1: SEMANTIC DUALITY

MSCs describe both the system behavior and how each individual component should perform to establish the desired result. A semantics reflecting this duality thus has both a global language and a *local language* for each process involved in the depicted collaboration.

In this section we study the relationship between the two languages. In a first step we distinguish the messages in whose sending or receipt a certain process is directly involved, and those that are sent and received in the process's environment. The local semantics reflects the fact that all messages that are not either sent or received by a given process are *hidden*: the process behavior is independent of hidden messages.

For each transition (n, s, m) in the global automaton with a hidden message s we add *all* transitions (n, s', m) with $s' \in (\mathcal{M} - \mathcal{M}_p) \cup \{\{\tau\}\}$: from the process's point of view it is indistinguishable if it was message s that was sent, or some other hidden message, or even no message at all.

Definition 8 (Local semantics). *For an MSC with processes P and global automaton $\mathcal{A} = \langle N, \Sigma, I, T, \mathcal{F}\rangle$, the* local semantics *for a process $p \in P$ is given as the associated* local automaton *$\mathcal{A}_p = \langle N, \Sigma, I, T', \mathcal{F}'\rangle$ with*

- $T' = \{(n, s, n') \mid (n, s, n') \in T$ *and* $s \in \mathcal{M}_p \cup \{\tau\}\}$
  $\cup\ \{(n, s', n') \mid (n, s, n') \in T$ *and*
  $s \in \Sigma - \mathcal{M}_p - \{\tau\}$ *and* $s' \in \Sigma - \mathcal{M}_p\}$
- $\mathcal{F}' = \{(n, s, n') \mid (n, s, n') \in \mathcal{F}$ *and* $s \in \mathcal{M}_p \cup \{\tau\}\}$
  $\cup\ \{(n, s', n') \mid (n, s, n') \in \mathcal{F}$ *and*
  $s \in \Sigma - \mathcal{M}_p - \{\tau\}$ *and* $s' \in \Sigma - \mathcal{M}_p\}$

In component-oriented proofs, we infer the validity of a global property from the validity of local properties. Hence, we require that the global and local semantics are in agreement. However, not all MSCs have this property.

More formally, we say that an MSC has the *decomposition property* if the following equation holds for the global automaton $\mathcal{A}$, processes P and the local automata $\mathcal{A}_p$, $p \in P$:

$$\bigcap_{p \in P} \mathrm{L}(\mathcal{A}_p) = \mathrm{L}(\mathcal{A})$$

Figure 4 shows an MSC that does not have the decomposition property: consider an implementation in which process "A" first sends message "A1" and process "C" then sends message "C2": this interaction is not allowed by the global semantics. It is, however, accepted by all local automata.

Since equivalence between Büchi automata can be checked with standard verification techniques (cf. [13]), a practical solution is to check the decomposition property whenever the MSC is intended to be used in a decompositional proof.

An alternative solution is to restrict the MSC syntax so that the decomposition property is guaranteed. *Causality* is such a restriction. Consider again the example in Figure 4. There is an implicit causal relationship between messages "A1" and "C1," and "A2" and "C2," respectively. If the causalities were made explicit (for example with an extra message between process "A" and and process "C" in one of the alternatives), the decomposition property would hold.

We now give a syntactic characterization of a class of *causal* MSCs. We introduce a few auxiliary notions: the

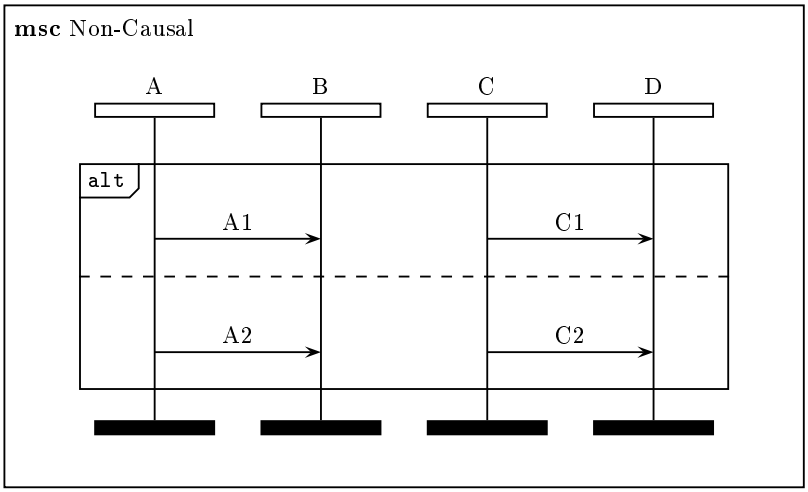


Figure 4: Non-causal MSC.

initial events $\text{init}(M)$ of an MSC M are those events that do not causally depend on any other event in M; dually, the *terminal* events $\text{term}(M)$ are those events that do not cause any other events in M. These sets of events serve as the basis for determining whether all message sequences expressed by an MSC are causally connected. Moreover, our aim is to distinguish clearly between different alternatives within a C-MSC by considering only the first message occurring within such an alternative; therefore, we also introduce a formal characterization for the set of first messages exchanged between two processes of an MSC.

We start by defining a causal order for the messages depicted in an MSC M. This serves as the basis for defining the sets $\text{init}(M)$ and $\text{term}(M)$, below.

Definition 9 (Causal Ancestor). *Let an MSC $M = \langle P, \mathcal{M}, E, C, O\rangle$ be given. We define an order $\leq \subseteq E \times E$ on M's events as follows. Let $e, f \in E$, then*

$$e \leq f \equiv (e = f) \vee (\exists m \in \mathcal{M} : (e, m, f) \in C) \vee (f = O(e))$$

If we have $e \leq f$, we call e direct causal ancestor *of f. By $\leq^*$ we denote the reflexive, transitive closure of $\leq$. If we have $e \leq^* f$, we call e* causal ancestor *of f.*

Thus, $e \in E$ is a direct causal ancestor of $f \in E$, if either e and f coincide, or e and f are the send and corresponding receive event of the same message transmission, or e occurs immediately before f on the axis of the same process in the corresponding MSC $M = \langle P, \mathcal{M}, E, C, O\rangle$. The causal order $\leq^*$ captures indirect causal dependencies. This allows us to define initial and terminal events by structural induction on the MSC syntax.

Definition 10 (Initial and terminal events). *For a basic MSC $M = \langle P, \mathcal{M}, E, C, O\rangle$ and its associated causal order $\leq^*$ we call an event $e \in E$* initial, *if $\forall f \in E : e \leq^* f$ holds; similarly, we call e* terminal, *if we have $\forall f \in E : f \leq^* e$. If e is M's initial event, we set $\text{init}(M) = \{e\}$; if M has no initial event we define $\text{init}(M) = \emptyset$. Similarly, we set $\text{term}(M) = \{e\}$ if e is M's terminal event, and $\text{term}(M) = \emptyset$ if no terminal event exists in M.*

For a C-MSC M the initial and terminal events are given as follows:

- $\text{init}(M_1;M_2) = \text{init}(M_1)$;
  $\text{term}(M_1;M_2) = \text{term}(M_2)$;
- $\text{init}(\texttt{alt}(M_1, M_2)) = \text{init}(M_1) \cup \text{init}(M_2)$;
  $\text{term}(\texttt{alt}(M_1, M_2)) = \text{term}(M_1) \cup \text{term}(M_2)$;
- $\text{init}(\texttt{loop}(M_1^*);M_2) = \text{init}(M_1) \cup \text{init}(M_2)$;
  $\text{term}(M_1;\texttt{loop}(M_2^*)) = \text{term}(M_1) \cup \text{term}(M_2)$;
- $\text{init}(\texttt{loop}(M_1^*)) = \text{init}(M_1)$;
  $\text{term}(\texttt{loop}(M_1^*)) = \text{term}(M_1)$
- $\text{init}(\texttt{loop}(M_1^\omega)) = \text{init}(M_1)$;
  $\text{term}(\texttt{loop}(M_1^\omega)) = \emptyset$.

In the definition of causal MSCs we will also constrain what messages may occur as a first message between two processes. We denote the set of first messages between process p and process q in the MSC M as $\text{fm}(M, p, q)$. Formally, let $\text{edges}(p, q)$ denote the set of interprocess edges between two processes p and q in a basic MSC:

$$\text{edges}(p, q) = \{(e_1, s, e_2) \in C, e_1, e_2 \in E_p \cup E_q\}.$$

The set of first messages is then defined as follows.

Definition 11 (First messages). *For a basic MSC M and two processes p, q with no interprocess edges between p and q, $\text{edges}(p, q) = \emptyset$, the set of first messages is empty: $\text{fm}(M, p, q) = \emptyset$. For non-empty $\text{edges}(p, q)$, we call the edge $(e_1, s, e_2) \in \text{edges}(p, q)$ where e_1 is a causal ancestor to all other send events e_1' with $(e_1', s', e_2') \in \text{edges}(p, q)$ the* first interprocess edge *and the message s the* first message *between p and q: $\text{fm}(M, p, q) = \{s\}$. For C-MSCs the first messages are the following sets:*

- $\text{fm}(M_1;M_2, p, q) = \text{fm}(M_1, p, q)$ *if* $\text{fm}(M_1, p, q) \neq \emptyset$ *and* $\text{fm}(M_2, p, q)$ *otherwise;*
- $\text{fm}(\texttt{alt}(M_1, M_2), p, q) = \text{fm}(M_1, p, q) \cup \text{fm}(M_1, p, q)$
- $\text{fm}(\texttt{loop}(M_1^*);M_2, p, q) = \text{fm}(M_1, p, q) \cup \text{fm}(M_1, p, q)$

- $\mathrm{fm}(\mathtt{loop}(M_1^*), p, q) = \mathrm{fm}(M_1, p, q)$
- $\mathrm{fm}(\mathtt{loop}(M_1^\omega), p, q) = \mathrm{fm}(M_1, p, q)$

Intuitively, each process in a causal MSC should always be able to infer which branch of the MSC is currently executed. This is ensured with the following syntactic constraints.

DEFINITION 12 (CAUSAL MSC). *An MSC M is a* causal MSC *if one of the following conditions holds.*

- *M is a basic MSC and has an initial event;*
- *M is a sequential composition $M = M_1;M_2$, M_1 has a terminal event e_1, M_2 has an initial event e_2 and e_1 and e_2 belong to the same process;*
- *M is an alternative between two causal MSCs, $M = \mathtt{alt}(M_1, M_2)$, and for all processes p and q, $\mathrm{fm}(M_1, p, q) \cap \mathrm{fm}(M_2, p, q) = \emptyset$*
- *M is a finite loop $\mathtt{loop}(M_1^*)$ or an infinite loop $\mathtt{loop}(M_1^\omega)$ of a causal MSC M_1.*

The MSC in Figure 4 is not causal, because the send-events for messages "A1" and "C1" do not have a common causal ancestor. In fact, the MSC would remain non-causal if were to remove the second alternative, even though the decomposition property holds for the resulting MSC. Causality is hence a sufficient but not necessary condition for the decomposition property.

Related work. The difficulty in mapping global properties to responsibilities of individual components has been considered in the literature (cf. [25, 26, 1] among others), sometimes under the keyword "nonlocal choice." Besides syntactic constraints as done for causal MSCs here, the problem can also be solved by partial or total distribution of an automaton representing the global property to all or part of the component implementation [19]; this ensures that all components synchronize their actions via the global automaton. This comes at the cost of increasing the complexity of the individual components considerably.

4. CHALLENGE 2: COMPLETENESS

In formal verification, we prove that a system satisfies its specification. If the system is the composition of a set of components $S = \{C_p \mid p \in P\}$ and the specification is given as an MSC M, verifying $S \models M$ corresponds to checking the language inclusion

$$\bigcap_{p \in P} \mathrm{L}(C_p) \subseteq \mathrm{L}(\mathcal{A})$$

where $\mathrm{L}(C_p)$ is the language accepted by the component implementing process p and $\mathcal{A}$ is the global automaton associated with M.

Analysis techniques for this problem are computationally expensive; the complexity of model checking [9], for instance, is exponential in n. It has therefore long been recognized that verification must be based on the decomposition of the system into its components.

We now discuss a decompositional proof rule for MSCs. Following the *assumption-commitment* paradigm, such a rule supplies two automata for each component: the *assumption* on the component's environment, represented by

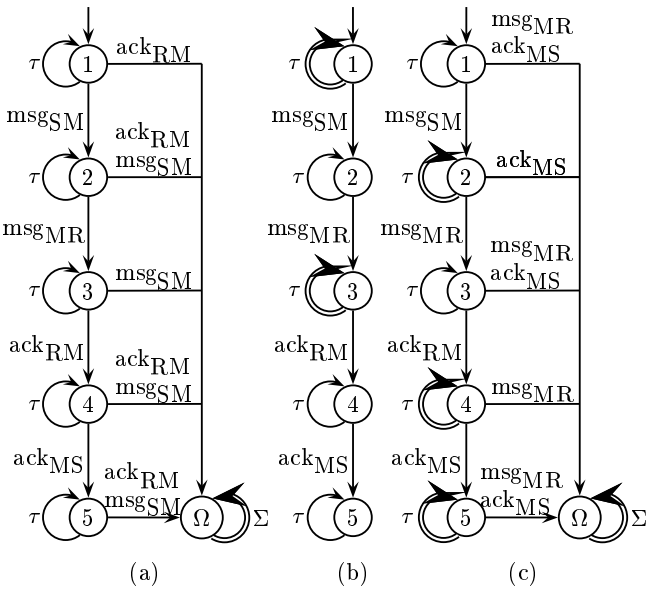


Figure 5: (a) environment-safety automaton, (b) environment-liveness automaton, (c) environment automaton for the "Medium" process from Figure 1.

an environment automaton $\mathcal{E}_p$, and the *commitment*, represented by the associated local automaton from the previous section. To prove $S \models M$ for a system $S = \{C_p \mid p \in P\}$ and an MSC M with global automaton $\mathcal{A}$ we find a second MSC M' that has the decomposition property. Let $\mathcal{A}'$ be the global automaton associated with M', and $\mathcal{A}'_p$ and $\mathcal{E}'_p$, the local automata and environment automata, respectively, for processes $p \in P$. The following rule reduces the global property to local proof obligations for each component:

DECOMPOSITIONAL PROOF RULE

if	(1) for all $p \in P$,	$\mathrm{L}(\mathcal{E}_p) \cap \mathrm{L}(C_p)$	$\subseteq$	$\mathrm{L}(\mathcal{A}'_p)$
and	(2)	$\mathrm{L}(\mathcal{A}')$	$\subseteq$	$\mathrm{L}(\mathcal{A})$
then		$\bigcap_{p \in P} \mathrm{L}(C_p)$	$\subseteq$	$\mathrm{L}(\mathcal{A})$

Since the component can *rely* on the environment to cooperate, we can exclude behaviors from the environment automaton in which the environment either illegaly sends a message (*safety* violation) or in which the component is kept waiting for the next message infinitely (*liveness* violation). We construct two automata, the environment-safety automaton $\mathcal{S}$ that recognizes all behaviors where the environment violates safety, and the environment-liveness automaton $\mathcal{L}$, that recognizes all behaviors where the environment violates liveness. Behaviors accepted by either automaton need not be considered in the verification of the component.

Safety violations can be recognized by considering finite prefixes of input sequences. Let $\mathcal{A} = \langle N, \Sigma, I, T, \mathcal{F} \rangle$ be the global automaton associated with an MSC M. The set of all finite prefixes of sequences in $\mathrm{L}(\mathcal{A})$, the *prefix language* of M, is accepted by the automaton $\langle N, \Sigma, I, T, T \rangle$. Because of the trivial acceptance condition it is possible to construct a deterministic Büchi automaton $\mathcal{P}_\mathcal{A}$ that accepts the prefix language (cf. [36]).

DEFINITION 13 (ENVIRONMENT-SAFETY). *Let the automaton* $\mathcal{P}_\mathcal{A} = \langle N, \Sigma, I, T, \mathcal{F} \rangle$ *be a deterministic Büchi automaton that accepts the prefix language of an MSC. The* environment-safety automaton *for process* p *is the automaton* $\mathcal{S}_p = \langle N', \Sigma, I, T', \mathcal{F}' \rangle$ *with*

- $N' = N \cup \{\Omega\}$
- $T' = T \cup \{(n, s, \Omega) \mid s \in \mathcal{M} - S_p \text{ and } \nexists n' \in N \,.\, (n, s, n') \in T\} \cup \{(\Omega, s, \Omega) \mid s \in \Sigma\}$
- $\mathcal{F}' = \{(\Omega, s, \Omega) \mid s \in \Sigma\}$

Figure 5a shows the environment-safety automaton for the "Medium" process from the communication protocol example. Note that the accepting paths stay in node Ω; a transition to Ω occurs whenever the environment illegally sends a message.

Figure 5b shows the environment-liveness automaton for the "Medium" process. The accepting paths stay in nodes 1 and 3: in node 1, the "Medium" process can count on the "Sender" process to eventually send a message; in node 3, the "Medium" process awaits the acknowledgement from the "Receiver" process.

DEFINITION 14 (ENVIRONMENT-LIVENESS). *Let the automaton* $\mathcal{P}_\mathcal{A} = \langle N, \Sigma, I, T, \mathcal{F} \rangle$ *be a deterministic Büchi automaton that accepts the prefix language of an MSC. The* environment-liveness automaton *for process* p *is the automaton* $\mathcal{L}_p = \langle N, \Sigma, I, T, \mathcal{F}' \rangle$ *with*

$$\mathcal{F}' = \{(n, \tau, n) \mid \nexists n' \in N, s \in S_p \,.\, (n, s, n') \in T \text{ and } \exists n' \in N, s \in (\mathcal{M} - S_p) \,.\, (n, s, n') \in T\}$$

Finally, the environment automaton $\mathcal{E}_p$ contains all behaviors in which the environment commits neither a safety nor a liveness violation. In the example, this combination results in the environment automaton shown in Figure 5c.

DEFINITION 15 (ENVIRONMENT AUTOMATON). *Let* $\mathcal{S}_p$ *be the environment-safety automaton and* $\mathcal{L}_p$ *the environment-liveness automaton for a process* p. *The* environment automaton $\mathcal{E}_p$ *accepts the language*

$$\mathrm{L}(\mathcal{E}_p) = \overline{\mathrm{L}(\mathcal{S}_p) \cup \mathrm{L}(\mathcal{L}_p)}$$

We can now analyze the completeness of our rule. The decompositional proof rule is *complete* if for *any* system S and MSC M with $S \models M$, there is an MSC M' such that the conditions of the rule hold. So far, we have made no assumptions about the components allowed in the system composition. In this generality, the decompositional proof rule is clearly incomplete.

In Figure 6, the specification is satisfied if the two processes "A" and "B" exchange exactly one message. Now consider the following implementation: component "A" chooses at each point nondeterministically whether or not to send its message to "B" (unless it receives a message from "B" first). "B," on the other hand, applies a timeout-mechanism that guarantees that eventually a message is sent. There is no MSC M' such that the conditions of the decompositional proof rule hold: none of the two alternatives in Figure 6 can be removed since either message may occur.

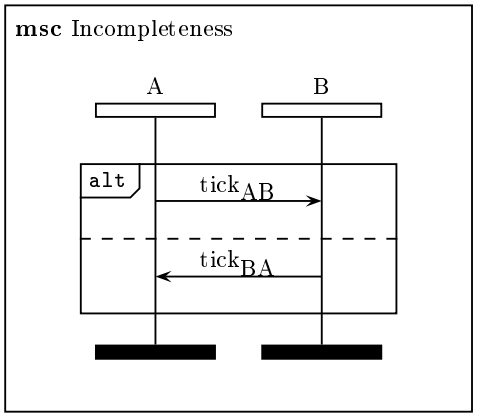


Figure 6: Incompleteness example.

The incompleteness of the proof rule stems from the difference in expressiveness of the MSCs we have considered so far, and the components implementing individual processes. This leaves us with two options for achieving completeness: one is to *restrict* our attention to a smaller class of systems, the other is to *add* to the expressiveness of MSCs.

Examples for restrictions are *regularity*: the language accepted by each component is ω-regular, *reactivity*: every component exchanges infinitely many messages with its environment, and *I/O directedness*: a component has control only over its output messages. If such restrictions are inadequate, it is certainly possible to make MSCs more expressive, for example with an explicit assignment of progress responsibilities: the property in Figure 6 could be proven for the described implementation by indicating in Figure 6 that process "B" is responsible for the progress beyond the interprocess edge (resulting in an appropriately modified environment automaton for process "A").

In practice, components often accept a non-regular language. Suggestions in the literature to extend MSCs to non-regular languages include extensions with data states [6, 22], as well as performance and real-time constraints (cf. [16, 17, 31]). However, any extension to the MSC's expressiveness comes at the price of increased complexity: many extended MSCs are undecidable. Care is also required to avoid syntactic clutter and to maintain the MSCs' intuitive appearance.

Related work. Decompositional proofs have been studied for a long time, starting with the *rely-guarantee* formalism [21] and proofs for networks of processes [27]. Since then, many assumption-commitment rules have been proposed, see [12] for an overview and [30] for a discussion of their completeness. Our decomposition of MSC properties into an assumption-commitment specification for individual components is similar to the one in [7]; the semantic framework used there includes the reactivity and I/O directedness requirements mentioned above. We are not aware of any work that formally analyzes the completeness of MSC languages. A closely related topic, however, is the "reverse engineering" of MSCs from systems; this is studied in [28].

5. CHALLENGE 3: COMPOSITIONALITY

In the preceding sections we have addressed the properties expressed by individual MSCs with respect to a certain system under consideration. Now we turn our attention to the composition of specifications from several, possibly *non-orthogonal* MSCs. Intuitively, two MSCs are non-orthogonal, if one contains a segment of an interaction pattern depicted by the other. We deal with this problem from two perspectives. First, we study the composition of "off-the-shelf" components specifically in the context of verification; here, the basic problem is to relate the already fixed interface specifications of already existing components. Second, we consider MSC composition in the more general context of scenario specifications.

In component-oriented system development, the different parts of a system are designed independently of each other; components may be retrieved from a database that was put together long before the system's conception. It is therefore unrealistic to expect that the MSCs documenting the different components will agree, and we need a process to resolve any differences.

In our communication protocol example, assume the "Sender" component is described by the simple MSC from Figure 1, and the "Medium" component has the richer functionality depicted in Figure 2. Which MSC describes the embedding of the composition of the two components in the system? Or should this combination of components be rejected altogether?

In assumption-commitment reasoning, the environment of a component is expected to show *at most* the behavior allowed by the environment assumption (cf. [12]). Hence, the combination of "Sender" and "Medium" component in our communication protocol example would be rejected, since the "Medium" component may send a "fail" message which is not allowed in the MSC of the "Sender" component. Semantically, this analysis corresponds to a *pessimistic* view of the environment [11]: The combination of two components is rejected because an environment exists that would violate the specification of one of the components. In this example, there is an implementation of the (so far not analyzed) "Receiver" process that corresponds to the MSC of the "Medium" component, but that would cause a violation of the MSC of the "Sender" component. A more liberal *optimistic* view allows the combination of two components as long as an implementation for the remaining environment exists that would allow all specifications to be satisfied.

The optimistic point of view can be implemented in a process for the composition of component MSCs. Given a system S and an MSC M_A specifying the behavior of a subset of the components $A \subseteq S$, and an MSC M_B specifying the components $B \subseteq S$, we construct an MSC $M_{A \cup B}$ specifying $A \cup B$. In this chart only those behaviors of $S - (A \cup B)$ are allowed that do not cause the components in A to violate environment assumptions of the components in B, or, vice versa, cause the components in B to violate environment assumptions of components in A.

There are automata-based solutions for optimistic composition (cf. [11]). It would be desirable to have purely syntactic combination operations for MSCs that implement this semantic construction and combinations for more expressive MSC languages. This would constitute a first step towards a thorough, seamless usage of MSCs as a specification and verification aid in the context of component composition.

We now turn to questions of MSC composition in a more general setting including analysis and design in addition to verification. As we have argued in the preceding sections we need a very strict MSC interpretation for promising MSC application in the verification task. The well-established usage of MSCs for capturing *scenarios*, on the other hand, is an example of a very liberal MSC interpretation. A scenario captures one possible segment of an overall system execution, projected onto the components referenced in the MSC. Because scenarios describe usually very specific instances of behavior, a corresponding composition operator must be very permissive; it cannot exclude alternative or even interleaved behaviors prematurely. [22] contains a composition operator, called "join", which matches the messages shared by the two operand MSCs; the resulting MSC's semantics contains only behaviors where this match is possible. This form of composition explicitly supports the combination of overlapping specifications; it is easily transferred into the semantic framework we have established in this paper.

Related work. The distinction between "optimistic" and "pessimistic" compositionality has been made in the verification literature, for example in *lazy compositional verification* [34] and, more recently, within the formalism of *interface automata* [11]. In the MSC literature certain dialects can be seen as closer to the pessimistic or optimistic point of view. [22] discusses MSC interpretations in the range from scenarios to exact component behavior; the latter excludes behaviors other than the explicitly depicted ones.

6. CONCLUSIONS AND OUTLOOK

Message sequence charts have been used for quite some time to *informally* describe the embedding of a component in its environment. In this paper we have formulated criteria the MSC language should satisfy so that the embedding furthermore qualifies as a *formal proof:* if this is achieved, then the correctness of the system is guaranteed once each individual component is verified.

Certain compromises must be made when choosing an MSC language. The simple MSCs described in Section 2 are attractive because their semantics is well-supported by verification methods; however, they do not provide a complete proof technique as discussed in Section 4. More expressive languages, such as the ones mentioned at the end of Section 4, on the other hand, are hard to analyze or even undecidable. For a given system and component model, a good compromise would be to first select a language on the basis of its completeness and then identify fragments according to their expressiveness.

Using MSCs as a verification tool as suggested in this paper should feel natural to designers familiar with MSC-based scenario descriptions. There is also a close resemblance to verification tools such as generalized verification diagrams [4, 5]: verification diagrams are similarly based on ω-automata, and they can also be used for component-oriented proofs [14]. MSCs and verification diagrams work, however, on different levels: verification diagrams are complete proofs of a certain property. MSCs, on the other hand, do not constitute complete proofs by themselves, since they are constructed independently of implementation details. Instead, they integrate the verification of individual properties in the correctness proof of the overall system.

Compositionality may be the hardest remaining challenge for a practical application of MSCs in component-based verification. In this paper we have addressed the composition of MSC specifications referencing concrete components of the system under consideration. Often, however, similar interaction patterns occur over and over again within the same system among different sets of components, and also within other systems. We can also identify and describe these interaction patterns by means of MSCs: we only have to interpret the axes of the MSCs more liberally. By parameterizing MSCs with respect to their axis labelings, i.e., the components they reference, we obtain a flexible language for such recurring interaction patterns. Instead of a single concrete component of a particular system under consideration, an axis then represents the "role" of a participant in the interaction pattern. The resulting MSCs describe interaction patterns abstractly, without references to concrete participants of a collaboration. We also speak of "connectors", when referencing abstract interaction protocols (cf. also [37, 33, 35, 7, 6]).

To use MSCs successfully in describing connectors (cf. [6, 7, 23, 15]) we need a way to relate abstract connectors and concrete component interfaces. One way to do so is to instantiate the roles in connectors by concrete components, whose interfaces are also specified by MSCs; in a second step we then have to match the behaviors allowed by the connector with those of the instantiating components.

Exploiting the information contained in a connector during component-oriented verification displays much potential for reducing the overall verification complexity, and is a promising area of future research.

Acknowledgments

The authors are grateful to Manfred Broy, César Sánchez, Bernhard Schätz and Henny Sipma for helpful discussions and comments on causality and MSCs in general.

7. REFERENCES

[1] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. In *Proceedings of 22nd International Conference on Software Engineering*, pages 304–313, 2000.

[2] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *28th International Colloquium on Automata, Languages and Programming*, LNCS. Springer-Verlag, 2001.

[3] R. Alur, G. J. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software — Concepts and Tools*, 17:70 – 77, 1996.

[4] A. Browne, Z. Manna, and H. B. Sipma. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 484–498. Springer-Verlag, 1995.

[5] A. Browne, Z. Manna, and H. B. Sipma. Hierarchical verification using verification diagrams. In 2^{nd} *Asian Computing Science Conf.*, volume 1179 of *LNCS*, pages 276–286. Springer-Verlag, Dec. 1996.

[6] M. Broy, C. Hofmann, I. Krüger, and M. Schmidt. A graphical description technique for communication in software architectures. Technical Report TUM-I9705, Technische Universität München, 1997.

[7] M. Broy and I. Krüger. Interaction Interfaces – Towards a scientific foundation of a methodological usage of Message Sequence Charts. In J. Staples, M. G. Hinchey, and S. Liu, editors, *Formal Engineering Methods (ICFEM'98)*, pages 2–15. IEEE Computer Society, 1998.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns. Pattern-Oriented Software Architecture*. Wiley, 1996.

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[10] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *FMOODS'99 IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, 1999.

[11] L. de Alfaro and T. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*. ACM Press, 2001.

[12] W.-P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference. COMPOS'97*, volume 1536 of *LNCS*. Springer-Verlag, 1998.

[13] B. Finkbeiner. Language containment checking using nondeterministic bdds. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*. Springer-Verlag, 2001.

[14] B. Finkbeiner, Z. Manna, and H. B. Sipma. Deductive verification of modular systems. In de Roever et al. [12], pages 239–275.

[15] J. Grabowski, P. Graubmann, and E. Rudolph. HyperMSCs with Connectors for Advanced Visual System Modelling and Testing. In *SDL Forum 2001*, pages 129–147. Springer, 2001.

[16] R. Grosu, I. Krüger, and T. Stauner. Hybrid sequence charts. Technical Report TUM-I9914, Technische Univerität München, 1999.

[17] R. Grosu, I. Krüger, and T. Stauner. Requirements Specification of an Automotive System with Hybrid Sequence Charts. In *WORDS'99F, Fifth International Workshop on Object-oriented Real-time Dependable Systems*. IEEE, 1999.

[18] R. Hardin, Z. Har'El, and R. Kurshan. COSPAN. In R. Alur and T. A. Henzinger, editors, *Proc.* 8^{th} *Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 423–427. Springer-Verlag, July 1996.

[19] D. Harel and H. Kugler. Synthesizing object systems from lcs specifications, 1999. (submitted).

[20] ITU-TS. Recommendation Z.120 : Message Sequence Chart (MSC). Geneva, 1996.

[21] C. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[22] I. Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Universität München, 2000.

[23] I. Krüger. Notational and Methodical Issues in Forward Engineering with MSCs. In T. Systä, editor, *Proceedings of OOPSLA 2000 Workshop:*

Scenario-based round trip engineering. Tampere University of Technology, Software Systems Laboratory, Report 20, 2000.

[24] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In *DIPES'98*. Kluwer, 1999.

[25] P. B. Ladkin and S. Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, (5):473–509, 1995.

[26] S. Leue. *Methods and Semantics for Telecommunications Systems Engineering*. PhD thesis, Universität Bern, 1995.

[27] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.

[28] A. Muscholl and D. Peled. From finite state communication protocols to high level message sequence charts. In *28th Int. Col. on Automata Languages and Programming (ICALP'2001)*, volume 2076 of *LNCS*, pages 720–731. Springer-Verlag, 2001.

[29] R. Nahm. Designing and documenting componentware with message sequence charts. In T. Jell, editor, *Component-based Software Engineering*, pages 111–116. Cambridge University Press, 1998.

[30] K. S. Namjoshi and R. J. Trefler. On the completeness of compositional reasoning. In *12th International Conference on Computer Aided Verification*, volume 1855 of *LNCS*, pages 139–153. Springer-Verlag, 2000.

[31] Unified modeling language, version 1.1. Rational Software Corporation, 1997.

[32] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[33] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. `http://www.objectime.com/otl/technical`, April 1998.

[34] N. Shankar. Lazy compositional verification. In de Roever et al. [12].

[35] M. Shaw and D. Garlan. Software architectures. perspectives on an emerging discipline, 1996.

[36] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier Science Publishers (North-Holland), 1990.

[37] A. C. Wills and D. D'Souza. *Objects, Components, and Frameworks with UML– The Catalysis Approach*. Addison Wesley, 1998.

Reasoning about Composition: A Predicate Transformer Approach

[Position Paper]

Michel Charpentier
Department of Computer Science
University of New Hampshire
charpov@cs.unh.edu

ABSTRACT

As interest in components and composition-related methods is growing rapidly, it is not always clear what the goals (and the corresponding difficulties) actually are. If composition is to become central in the future of software engineering, we need to better identify the fundamental issues that are related to it, before we attempt to solve them as they occur in object-oriented systems or in concurrent and reactive systems. In this paper, we present our formulation of some of the composition problems in a context of formal methods and program specification and verification. This formalization is based on predicate calculus and predicate transformers and aims at remaining as general as possible. This way, we hope to better understand some of the fundamental issues of composition and component-based reasoning.

1. INTRODUCTION

Composition is receiving a lot of attention these days: Components are everywhere and everything is (or ought to be) "*compositional*". What is meant by that, though, is far from being clear, and there is a wide range of opinions on what is still to be done. Some might argue that the composition problem is now solved at a fundamental level and that actual techniques and tools just need to be put in place. At the other end of the spectrum, some might believe that composition, as we understand it today, cannot be achieved in software engineering and that other approaches must be sought. And, between these two extremes, are research projects, mostly independent from one another, that focus on specific instances of this composition problem, be it a type system in an object-oriented context or a temporal logic for reactive systems.

A possible reason for this apparent contradiction and confusion is that composition is a broad concept and that the composition problem might not be unique. There are many issues related to composition, some are easier to tackle than others, and many must be dealt with before the problem can be considered solved (or unsolvable). In this paper, we advocate the idea that an important step today is to identify those composition problems and to understand how they relate to each other.

We restrict our attention to composition in a formal methods context. Other contexts, such as for instance programming languages, lead to other composition problems and all have to be solved in order to make composition viable as a whole. Furthermore, we choose a static point of view: we reason about properties of systems and components whereas a dynamic point of view would focus on the *process* of building systems from components.

These choices, however, leave us in a broad background where fundamental questions related to many forms of composition can be explored: What are components? How are they composed? How are they described and specified? What do we expect from such specifications? What is the relationship between systems and components specifications? What does it mean to be "*compositional*"? How can we obtain compositional specifications? Can composition lead to simpler correctness proofs? How does composition relate to reuse? How does it relate to abstraction?

Our current effort focuses on addressing these questions without specializing the chosen context any further. This way, we hope to better understand what problems are common to different forms of composition and what problems are specific to families of components or laws of composition. As a guideline for this general exploration, we also consider the special case of concurrent composition of processes specified in temporal logic. This familiar but complex background, in which the composition problem is far from being solved, is both a source of inspiration and a test-bench for our abstract study of composition. Our approach to studying composition as well as some of our results are informally introduced in the remaining of this paper. Technical details can be found in cited references.

2. SPECIFICATIONS AND PROOFS IN COMPOSITIONAL DESIGNS

2.1 Compositional Design versus Compositional Verification

Composition has often been advocated as a necessary step in the proof of large systems. While this is certainly true, we do not want to restrict composition to that role.

For instance, it is possible to build a system from components, generate correctness proof obligations from the *complete* system, and then apply composition at the proof level (split the global proof obligation into several independent proofs). This approach is suggested, for instance, in [24]. Compositional model-checking also follows this philosophy to some degree.

While the previous technique is relatively simple and allows verification techniques to handle large systems, we have in mind a more ambitious role for composition, namely the "open system" approach. In this approach, we want to verify the correctness of components in isolation, *before* they become part of any system. In the previous case, the complete knowledge of the system can be used to verify one component. For open systems, this is not true anymore. All that is known are specific assumptions on possible environments, which are part of a component specification. This tends to make proofs harder since these assumptions describe a set of possible environments instead of a completely specified context, and they have to be abstract and generic enough to allow a large number of environments to use the component.

However, the open system approach also has benefits that make its study worthwhile. Firstly, since components are already proved correct with respect to their specifications, the correctness proof of a complete system can rely on these specifications instead of the components' implementations. This allows designers not to take into account the many details of the internal structure of each component. Compositionality of designs breaks down when reasoning about a system requires managing too many details from each part of that system.

Secondly, and this is probably the main benefit, the open system approach allows designers to embed parts of a correctness proof into components, making these parts available each time a component is used to build a system. Indeed, when a component is proved correct with respect to its specification, relevant facts about this component are extracted from the details of its implementation and become part of the component specification. When this component is composed with a larger system, these facts can be used in the system correctness proof without the need for proving them again. Each time a component is reused, a (possibly difficult) proof is reused too, as well as any other correctness argument available such as tests or behavior in other systems.

2.2 Abstract Specifications

In order to be able to achieve such reuse, we need specifications to remain abstract enough to describe what is required from a component, all that is required and only what is required. When designing a system and looking for a suitable component, the specification used by the designer cannot include too many details about this component, because any component with the right functionalities should be usable, whatever its implementation details are. Such a specification must also be able to express that some aspects are irrelevant in order to avoid an overspecification of requirements. If requirements are overspecified, then designers might end up not finding any suitable component while actually some existing component would fit their needs perfectly.

A second reason why we want specifications to be abstract is to keep composition worthwhile and cost effective in spite of the natural overhead it generates. A key idea of component technology is that the same component can be used in many systems, and thus the effort that goes into specifying, proving and implementing components can be exploited many times. As explained before, each time a component is reused, a proof, the correctness proof of that component, is reused too. If a component specification contains abstract, relevant, hard-to-prove facts about the component, a possibly difficult and large proof is reused. However, if a component specification is too close to its implementation and not abstract enough, very little proof can be reused. Therefore, greater productivity is achieved by using components that embody substantial effort by containing proofs of *abstract* specifications.

This situation is illustrated in figure 1. Proofs labeled with 'T' are those component-correctness proofs that are left unchanged through composition and that can be reused in the design of several systems. Proofs labeled with 'C' are proofs of composition, i.e., proofs of system properties from component properties. The level of abstraction of component specifications clearly influences the amount of effort that has to be put in T-proofs and in C-proofs. A good framework for composition should allow us to put most of the effort in T-proofs and keep C-proofs as simple as possible. Even if the sum of C and T-proofs is larger and more complex than a direct (noncompositional) proof for the same system, composition is still worthwhile because existing T-proofs can be reused.

Part of the problem is that specifications that are too abstract do not contain enough information to be composed. Therefore, the right balance between abstraction and ability to be composed must be found.

3. SPECIFICITY OF OUR RESEARCH

3.1 Shortcomings of Current Approaches

When deterministic components are composed sequentially, the problem reduces to composition of functions and remains tractable. Developers use libraries of procedures every day and rely on their specifications without having to consider implementation details.

However, effective compositional design often involves nondeterministic components and concurrent composition. For instance, the different parts of a reactive system cannot be specified in terms of precondition and postcondition because of their possibly infinite behavior, which leads to tremendous difficulties in terms of composition.

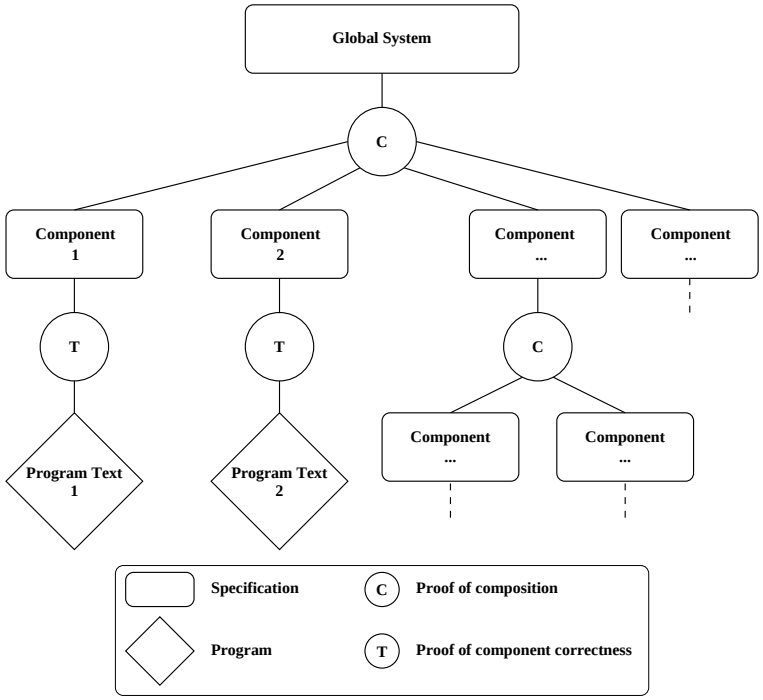


Figure 1: A compositional design

Composition of such systems, which interact at the level of their behavior, not at the level of their initial/final state, has been extensively studied. Very schematically, two distinct families emerge.

On the one hand, process algebras, such as CSP, CCS or π-calculus to name a few, integrate composition as a central part of their design. Systems are compositions of processes and processes compose quite naturally. The resulting formally well-defined notation, however, often looks like a programming language more than a specification language. In this context, it is quite difficult to express abstract properties on the expected behavior of these components and systems. As a consequence, it is difficult to obtain reusable generic specifications, as well as specifications easily related to informal requirements.

Temporal logics, on the other hand, such as LTL, CTL, TLA, or UNITY, are well-suited to express nonoperational, abstract specifications. They provide us with specification languages that are closer to informal descriptions, which makes specifications more easily readable and checkable with respect to informal requirements. However, the starting point of these notations is the specification of a system, globally. Composition is viewed as an additional issue, which requires a specific treatment. Work has been done to manage composition issues with specific logics [25, 19, 18, 2, 20, 23, 34, 21], but little work has been done to study composition in itself, independently of the underlying logic [1, 4, 3].

3.2 Composition in the Abstract

The specificity of our approach is to study composition independently from what components and the laws of composition actually are. We are not focusing on a specific domain, nor do we want to design specification languages tailored to certain forms of composition. Our view is one of a component-based software industry, where composition is involved in almost every design. We want to deal with this composition, whether it works well or not, whether it is easy or not. This departs from the works on temporal logic cited above, where usually the whole language is made "composable" by restricting up front the type of interaction under consideration. For instance, composition works fine in TLA (it reduces to conjunction) [2], but component assumptions are made at the transition level and cannot be at the computation level, as in the case of liveness assumptions (see example in section 4.4).

The context of our work is therefore independent of the nature of systems. It is not a context of "variables", "states", "computations", "interleaving", "safety" or "liveness", but rather one of "systems", "components", "specifications" and "composition laws". No specific logic or process model is used and few hypotheses are made on composition laws. This way, it is hoped that we can understand aspects that are common to many forms of composition and many types of systems. Later, that knowledge can be applied to the concurrent composition of reactive systems, for instance.

This approach inherits from both the process algebra and the temporal logic families mentioned above. On the one hand, we consider that systems are specified logically (without choosing a specific logic), which provides us with a rich specification language and allows us potentially to apply results to temporal logics for reactive systems. On the other hand, we define an algebra of composition with the goal of obtaining a calculus that would allow us to calculate (instead of guess and then prove) properties of systems and components. In this respect, our approach relates to process calculus based approaches.

4. CURRENT WORK

4.1 Existential and Universal Specifications

The starting point of our exploration is the definition of a simple model of components, systems and specifications. Because of our concern with generality, we use a monoid-like structure of components and specifications are boolean functions (predicates) on components and systems. In other words, we assume that components are composed with a single law of composition for which we assume associativity but no other property such as symmetry or idempotency. As a consequence, the model can be instantiated with transformational programs (specified in terms of preconditions and postconditions and composed sequentially) or with reactive processes (specified with temporal logics and composed concurrently), among other things.

In this context, we first focused on two particular families of specifications called *existential* and *universal* [7, 14]. We say that a specification is existential exactly when, for all systems, the specification holds in a system if it holds in at least one component of that system. Similarly, a specification is universal if it holds in a system when it holds in all components of that system. Existential and universal are characteristics of specifications, independent of a particular set of components. Some specifications are existential, some are universal and, of course, some are neither. However, when existential and universal specifications are used, they naturally lead to simple proofs of composition (C-proofs), properties being inherited by a system from its components.

4.2 A "*guarantees*" Operator for Assumption-Commitment Specifications

If we only allow existential and universal specifications to appear in component descriptions, this is a restriction on how components can be described. This is the price to pay for simple proofs of composition. However, we have found these two classes to be surprisingly rich. For instance, the work on temporal logics in [19, 18, 2, 20, 34, 21] relies almost exclusively on existential-like composition.

One reason why existential specifications appear to be so convenient is the existence of the *guarantees* operator defined in [7]. The *guarantees* operator can be used to express existential assumption-commitment specifications. Its main originality is that it is not defined in terms of component *environments*, as assumption-commitment specifications usually are (components are making assumptions on their possible environments). In the case of *guarantees*, the commitment part of the specification *as well as the assumption part* apply to a complete system (environment + component): X *guarantees* Y holds in a component F if and only if Y holds in $G \circ F \circ H$ (where $\circ$ denotes the law of composition under consideration) when X holds in $G \circ F \circ H$, for all systems G and H that can be composed with F. The fundamental property of *guarantees* is that X *guarantees* Y is existential regardless of what the specifications X and Y are. Therefore, proofs of composition are simplified when components are specified in terms of *guarantees*.

4.3 Predicate Transformers for Composition

By studying the *guarantees* operator carefully, we made the observation that it is merely the application to logical implication of a more general operator which we called WE [6]. This allows a separation of concerns: WE actually represents composition while logical implication represents the assumption-commitment mechanism. WE is a predicate transformer, in other words, a function from specifications to specifications. Formally, for a specification X, WE.X is defined as the weakest existential specification stronger than X (which exists regardless of X).

It can be proved that WE.X characterizes those components F such that specification X holds in any system that contains F as a component [14]. As a consequence, the specification X *guarantees* Y is actually equivalent to WE.$(X \Rightarrow Y)$. In other words, *guarantees* is the weakest (the most abstract) strengthening of logical implication that makes it composable (for the existential form of composition). This, in some sense, is a theoretical argument to claim that *guarantees* can provide us with abstract, reusable specifications.

WE is the first of a series of predicate transformers that we have started to study. Indeed, we can define SE.X as the strongest existential property weaker than X. The corresponding theorem states that SE.X characterizes those systems that contain at least one component that satisfies X. In other words, when a component that satisfies X is used in a system, this system satisfies SE.X. In the best case (when X is an existential specification), the system satisfies X (SE.X is equivalent to X); in the worst case (where all of X is lost through composition), SE.X reduces to *true*. In some sense, SE.X represents the part of specification X that composes (existentially). Equivalently, SE.X characterizes those systems that are (or can be) built using a component that satisfies specification X [15].

Things are different in the case of universal composition. A transformer SU can be defined (as the strongest universal specification weaker than a given specification), but we are still looking for a suitable WU. Such a transformer would be useful to characterize what has to be proved on a component instead of a (nonuniversal) specification X in order to inherit the simplicity of universal composition. However, it cannot be defined as the weakest universal specification stronger than a given specification because such a weakest element does not always exist, depending on the nonuniversal specification that is considered. We have started to study several possible candidates for a WU operator but we do not have a strong argument in favor of one of them yet. As a guideline for that search of WU, we have also studied the question of strengthening nonuniversal properties in a more restricted context, namely a linear temporal logic (see section 4.4).

Furthermore, by describing composition in terms of predicate transformers, for which a large amount of literature exists [22], we are able to reuse classic techniques such as *conjugates*. Every predicate transformer $\mathcal{T}$ has a unique conjugate $\mathcal{T}^*$ such that $\mathcal{T}^*.X = \neg\mathcal{T}.(\neg X)$. The transformers we have defined for existential and universal composition also have conjugates, namely WE*, SE* and SU*. It should be noted that, while WE, SE and SU describe composition from components to systems (what has to be proved on components, what can be deduced on systems), WE*, SE* and SU* describe composition from systems to components

(what should be proved on systems, what can be deduced on components). For instance, $\mathbb{WE}^*.X$ is true of any component that is used to build a system that satisfies specification X. This form of reasoning, from systems to components, is sometimes neglected. We believe it to be extremely important because it is the kind of reasoning that is involved when system designers are looking for components. A designer who is building a system to satisfy specification X knows that only components that satisfy $\mathbb{WE}^*.X$ can be used and that other components need not be considered. We find conjugates to be a powerful and elegant way to switch from bottom-up to top-down views on composition [13]. In particular, many properties of predicate transformers, such as junctivity and monotonicity, are inherited from transformers to conjugates. This allows us to avoid duplicating proofs.

4.4 Application to UNITY logic

In parallel with our work on predicate transformers and composition, we have started to apply our ideas to specifications and proofs of concurrent and distributed systems. Theoretical investigation is one way to claim the usefulness of operators (for instance, by proving that they are the weakest solution to some set of equations). Practical attempts at writing specifications and proofs based on these operators are another.

Two of these examples were fully developed and published. One focuses on shared memory systems [11], while the other deals with distributed systems [12, 5].

In the first example, universal specifications are used instead of *guarantees*, which does not seem to fit this example well enough. In this case, the correctness argument relies on the fact that some dependency graph among processes remains acyclic. Since each process only modifies the dependency graph locally (by interacting with its neighbors), no single process can guarantee that the graph remains acyclic, using an existential property. However, there can be a property that states that no process will ever create a cycle in the graph. Such a property can be formulated in a universal way so that, when it is satisfied by all processes, the global system also satisfies it and cycles cannot be introduced in the graph.

This raises a number of interesting questions. In this example, it appears that universal specifications are required to describe the behavior of shared variables (variables that are written by several processes). However, there are other examples with shared variables that can successfully be specified in terms of *guarantees*. There are also systems without shared variables (distributed systems) but where a shared virtual data structure (such as a graph among processes) is used in the correctness proof. Should such a system be specified in terms of *guarantees* (it usually can, from the absence of shared variables) or in terms of universal specifications of the shared virtual data structure? And if *guarantees* is used, should the correctness proof rely directly on it or can we obtain a simpler proof by using an intermediate (universal) specification that is deduced from the original (existential) specification? These are the kind of fundamental questions we plan to explore through the development of other examples.

Using universal specifications gives rise to other interesting issues. For instance, the UNITY logic (which was used in our examples) exists in two forms: a weak form and a strong form [31, 28, 27]. The UNITY operator *invariant* leads to universal specifications in its strong form but not in its weak form. For the sake of simplicity, we used the strong form of UNITY logic in our example. However, this is not realistic from a practical point of view (the strong form of the logic is much too strong for a specification) and we have to find ways of strengthening the weak form to make it universal. We have defined such a strengthening based on $\mathbb{WE}$ [9] (the resulting universal form of the weak invariant resembles a similar operator from [34]), but we cannot tell if this is an optimal solution. In other words, we do not know if the resulting operator is the weakest universal specification stronger than the weak invariant (we do not even know if such a weakest solution exists). Besides its practical interest, this question also relates to the problem of finding a suitable transformer $\mathbb{WU}$, as explained earlier in section 4.3.

Our second example involves distributed systems. It makes use of *guarantees*, mixed with techniques for abstract communication description that were previously developed [16, 26, 8, 33, 17]. This abstract description of communication is made possible by the ability of *guarantees* to involve liveness specifications in its assumption part. Basically, a network component guarantees that the sequence of received messages is always a prefix of the sequence of sent messages (safety) and that any message that is sent is eventually received (liveness).

There are other places in this example where our use of liveness specifications combined with *guarantees* leads to simpler proofs of composition by embedding larger proofs in components verification (see the discussion in 2.2). For instance, this example involves a resource allocator component that satisfies a property of the form: *clients return resources in finite time (and other conditions)* guarantees *any request for resources is eventually satisfied*. The proof of composition remains simple because the corresponding client component property that states that clients actually return resources in finite time can be plugged (through network specifications) into the left-hand side of this *guarantees* property to deduce that all requests are eventually granted.

If liveness properties cannot be used in the assumption part of a composition operator $\mapsto$ (as in [1, 2, 18, 19, 20, 21]), the resource allocator specification has to be of the form: *enough resources are available to satisfy the first pending request* $\mapsto$ *the first pending request is eventually granted*. In this case, the fact that clients return resources in finite time cannot be used directly as before. Instead, a first proof of composition is required to show that enough resources will eventually be available to satisfy the first pending request and then a second proof to show that other requests are eventually satisfied. When *guarantees* is used, these two proofs (by induction) are inside the correctness proof of the allocator component and can be reused when the allocator component is reused. In the other case, they are in the proof of composition and have to be redone every time a new system is built from these components.

5. OUTLINE OF FUTURE RESEARCH

The work described above represents a first step towards our exploration of composition issues in system design. Starting with *guarantees* as a middle point, the research is now developing both upstream (towards predicate transformers and other fundamental composition-related operators) and downstream (towards practical application to concurrent systems).

One of our goals is the definition of a formal calculus in which specifications can be transformed to fit specific composition constraints. In other words, starting from requirements that are not compositional, we want to calculate a suitable compositional specification. In the case of existential composition, for example, it is not enough to know that WE.X is what needs to be proved on a component to ensure that systems which use that component will satisfy specification X. We need to know *how* to prove WE.X given a component description.

This can be achieved at different levels. At the most abstract level, we can exhibit theorems about WE that allow us to reduce the calculation of WE.X using known WE.Y, where Y is a part of X (for instance, using existential Y specifications). When this is possible, we can *calculate* WE.X inside the logic in which X is expressed, which gives us the corresponding component specification. We were able to achieve such calculations on toy examples [14], but we need more theorems and rules related to WE and our other transformers to be able to conduct such calculations on examples from more interesting domains. One difficulty when seeking such properties of the transformers is to free ourselves from implicit assumptions regarding the law of composition. For instance, we sometimes use concurrent composition of processes as a guideline to find general rules about the transformers, but we must be careful not to use an hypothesis such as symmetry or idempotency which we decided not to include systematically in our model.

Another way to deal with the transformers is to first instantiate our framework with a specification language and then to derive rules about WE.X, when X is expressed in the chosen logical language (instead of using general theorems about WE). We have started this process with UNITY logic in order to build the necessary correctness proofs in our examples with concurrent and distributed systems [9]. Furthermore, we also need to apply our approach to other frameworks for the specification and verification of concurrent systems. This effort has already started, for instance with CTL [32], but we want to consider other frameworks, such as TLA or I/O-automata.

Recently, we have started to generalize our approach to systems in which several laws of composition are used at the same time. An example of such a system is a software system in which components are composed sequentially *and* in parallel. According to preliminary results, it seems that our approach can still be applied. In other words, we are still able to define weakest and strongest transformers that represent specific views on composition (independently, this time, from existential and universal specifications). Furthermore, the resulting predicate transformers bear strong similarities with Dijkstra's *wlp* and *sp* transformers for program semantics, from which we can draw new inspirations [10]. This new set of transformers has now to be explored carefully. Especially, relationships between transformer properties and assumptions on the different laws of composition have to be found.

6. SUMMARY

The lack of composition-based methods is a major factor in the limited use of formal methods in actual designs. We believe our project adopts a novel view on an old and important problem. Most work on composition has focused on a specific form of composition (sequential, parallel with shared variables, parallel with message passing, etc.) and a specific type of component (namely, programs, either with states or with so-called "open system computations"). By choosing a much more general view, we hope to understand fundamental aspects of composition that are independent from the types of components and the way they interact.

Our ultimate goal is to build a calculus for composition. It would be a formal framework that can be instantiated with many form of compositions and many types of systems and components. We hope this framework will include generic rules and theorems about composition and logical specifications. The search for such fundamental rules, common to any kind of composition, is an exciting problem. Then, each instantiation enriches the framework with additional rules that are specific to this instantiation, making it more complete and more practically usable.

Besides this theoretical part of the project, we are experimenting with several notations for the specification and verification of concurrent systems to see how they can be extended through our approach into compositional notations. We hope, by modifying and extending existing notations, to develop an interesting framework to reason about concurrent composition of reactive systems. Another aspect of the problem is related to mechanization. We are investigating the question of the mechanization of *guarantees* through a collaboration with Larry Paulson from the University of Cambridge. Larry is currently working on a mechanization of UNITY [29] extended with *guarantees* [30] in the higher-order generic theorem prover *Isabelle*. His work is guided by his attempts at mechanizing hand proofs from our example involving distributed systems.

We are convinced that the future of software engineering is tied to composition. Component-based designs and reuse of generic components will be at the core of future software systems. Composition involves a number of practical issues, but also raises fundamental questions regarding component specifications and compositional reasoning. We need to improve our understanding of composition if we want to be able to devise the tools and principles that will allow us to use components reliably and efficiently in software engineering. Our project has started an exploration of some of the fundamental questions inherent in compositional design.

7. REFERENCES

[1] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[2] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[3] Martín Abadi and Stephan Merz. An abstract account of composition. In Jivrí Wiedermann and Petr Hajek, editors, *Mathematical Foundations of Computer Science*, volume 969 of *Lecture Notes in Computer Science*, pages 499–508. Springer-Verlag, September 1995.

[4] Martín Abadi and Gordon Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, June 1993.

[5] K. Mani Chandy and Michel Charpentier. An experiment in program composition and proof. *Formal Methods in System Design*, April 1999. Accepted for publication.

[6] K. Mani Chandy and Michel Charpentier. Predicate transformers for composition. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science: proccedings of the 1999 Oxford-Microsoft symposium in honour of Sir Tony Hoare*, Cornerstones of Computing, pages 81–90. Palgrave, 2000.

[7] K. Mani Chandy and Beverly Sanders. Reasoning about program composition. http://www.cise.ufl.edu/∼sanders/pubs/composition.ps.

[8] Michel Charpentier. *Assistance à la Répartition de Systèmes Réactifs*. PhD thesis, Institut National Polytechnique de Toulouse, France, November 1997.

[9] Michel Charpentier. Making UNITY properties compositional. Unpublished report, California Institute of Technology, 1999.

[10] Michel Charpentier. A theory of composition motivated by wp. Submitted for publication, August 2001.

[11] Michel Charpentier and K. Mani Chandy. Examples of program composition illustrating the use of universal properties. In J. Rolim, editor, *International workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'99)*, volume 1586 of *Lecture Notes in Computer Science*, pages 1215–1227. Springer-Verlag, April 1999.

[12] Michel Charpentier and K. Mani Chandy. Towards a compositional approach to the design and verification of distributed systems. In J. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems (FM'99), (Vol. I)*, volume 1708 of *Lecture Notes in Computer Science*, pages 570–589. Springer-Verlag, September 1999.

[13] Michel Charpentier and K. Mani Chandy. Reasoning about composition using property transformers and their conjugates. In J. van Leeuwen, O. Watanabe, M. Hagiya, P.D. Mosses, and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics (IFIP-TCS'2000)*, volume 1872 of *Lecture Notes in Computer Science*, pages 580–595. Springer-Verlag, August 2000.

[14] Michel Charpentier and K. Mani Chandy. Theorems about composition. In R. Backhouse and J. Nuno Oliveira, editors, *International Conference on Mathematics of Program Construction (MPC'2000)*, volume 1837 of *Lecture Notes in Computer Science*, pages 167–186. Springer-Verlag, July 2000.

[15] Michel Charpentier and K. Mani Chandy. Specification transformers: A predicate transformer approach to composition. Submitted for publication, July 2001.

[16] Michel Charpentier, Mamoun Filali, Philippe Mauran, Gérard Padiou, and Philippe Quéinnec. Abstracting communication to reason about distributed algorithms. In Ö. Babaoğlu and K. Marzullo, editors, *Tenth International Workshop on Distributed Algorithms (WDAG'96)*, volume 1151 of *Lecture Notes in Computer Science*, pages 89–104. Springer-Verlag, October 1996.

[17] Michel Charpentier, Mamoun Filali, Philippe Mauran, Gérard Padiou, and Philippe Quéinnec. The observation: an abstract communication mechanism. *Parallel Processing Letters*, 9(3):437–450, September 1999.

[18] Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications. Application to* UNITY. Doctoral thesis, Faculté des Sciences Appliquées, Université Catholique de Louvain, June 1994.

[19] Pierre Collette. An explanatory presentation of composition rules for assumption-commitment specifications. *Information Processing Letters*, 50:31–35, 1994.

[20] Pierre Collette and Edgar Knapp. Logical foundations for compositional verification and development of concurrent programs in UNITY. In *International Conference on Algebraic Methodology and Software Technology*, volume 936 of *Lecture Notes in Computer Science*, pages 353–367. Springer-Verlag, 1995.

[21] Pierre Collette and Edgar Knapp. A foundation for modular reasoning about safety and progress properties of state-based concurrent programs. *Theoretical Computer Science*, 183:253–279, 1997.

[22] Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Texts and monographs in computer science. Springer-Verlag, 1990.

[23] J.L. Fiadeiro and T. Maibaum. Verifying for reuse: foundations of object-oriented system verification. In

I. Makie C. Hankin and R. Nagarajan, editors, *Theory and Formal Methods*, pages 235–257. World Scientific Publishing Company, 1995.

[24] Leslie Lamport. Composition: A way to make proofs harder. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference (COMPOS'97)*, volume 1536 of *Lecture Notes in Computer Science*, pages 402–423. Springer-Verlag, September 1997.

[25] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[26] R. Manohar and Paul Sivilotti. Composing processes using modified rely-guarantee specifications. Technical Report CS-TR-96-22, California Institute of Technology, 1996.

[27] Jayadev Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[28] Jayadev Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[29] Lawrence C. Paulson. Mechanizing UNITY in Isabelle. *ACM Transactions on Computational Logic*, 1(1), July 2000.

[30] Lawrence C. Paulson. Mechanizing a theory of program composition for UNITY. *ACM Transactions on Computational Logic*, 2001. To appear.

[31] Beverly A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, April–June 1991.

[32] Beverly A. Sanders and Hector Andrade. Model checking for open systems. Submitted for publication, 2000.

[33] Paolo A. G. Sivilotti. *A Method for the Specification, Composition, and Testing of Distributed Object Systems*. PhD thesis, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125, December 1997.

[34] Rob T. Udink. *Program Refinement in* UNITY*-like Environments*. PhD thesis, Utrecht University, September 1995.

Specification and Verification with References

Bruce W. Weide and Wayne D. Heym
Computer and Information Science
The Ohio State University
Columbus, OH 43210
+1-614-292-1517
{weide,heym}@cis.ohio-state.edu

ABSTRACT

Modern object-oriented programming languages demand that component designers, specifiers, and clients deal with references. This is true despite the fact that some programming language and formal methods researchers have been announcing for decades, in effect, that pointers/references are harmful to the reasoning process. Their wise counsel to bury pointers/references as deeply as possible, or to eliminate them entirely, hasn't been heeded. What can be done to reconcile the practical need to program in the languages provided to us by the commercial powers-that-be, with the need to reason soundly about the behavior of component-based software systems? By directly comparing specifications for value and reference types, it is possible to assess the impact of visible pointers/references. The issues involved are the added difficulty for clients in understanding component specifications, and in reasoning about client program behavior. The conclusion is that making pointers/references visible to component clients needlessly complicates specification and verification.

Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: Languages, Methodologies.

D.2.4 [**Software/Program Verification**]: Correctness proofs, Formal methods, Programming by contract, Reliability.

General Terms

Design, Reliability, Languages, Verification.

Keywords

Java, Pointers, References, Specification, Verification.

1. INTRODUCTION

A well-known "folk theorem" in computing circles is that nearly every problem can be solved with one more level of indirection. Like most folklore, this claim is partially true—a fact not lost on programming language designers, who have consistently delivered not only computational models, but a variety of language constructs, to help us more easily write programs that use indirection.

The belief is a dangerous one, however, which has been noted many times over the past few decades. Writing programs more easily is one thing. Reasoning more easily about their behavior is quite another. As early as 1973, Tony Hoare remarked of pointers that "their introduction into high-level languages has been a step backward from which we may never recover" [10]. In 1976, Dick Kieburtz explained why we should be "programming without pointer variables" [15]. And in 1978, Steve Cook's seminal paper on the soundness and relative completeness of Hoare logic [3] identified aliasing (of arguments to calls, i.e., even in a language without pointer variables) as the key technical impediment to modular verification. There have been recent papers (e.g., [19, 23]) showing how it is *technically* possible to overcome such problems, but apparently only at the cost of even further complicating the programming model that a language presents to a software engineer.

Why do we need another paper about this issue? The consequences of programming with pointers have been examined so far primarily in the context of programming language design and formal methods. We take a position in the context of the human element of specification and verification:

> *Making pointers/references visible to component clients needlessly complicates specification and verification.*

In supporting this position, we rely in part on another, and far older, bit of folklore: "Occam's Razor", a.k.a. the Law of Parsimony. It holds that simpler explanations of phenomena are better than more complex ones. The phenomena of software behavior are entirely of our own making, giving us ample opportunity to control the intellectual complexity and comprehensibility of specifications and reasoning based on them.

Throughout the paper (and with apologies to C++ gurus, as noted in Section 5.1) the terms "pointer" and "reference" are used interchangeably. The point, so to speak, is that from the standpoint of specification and verification difficulties they amount to the same thing. Code examples use Java notation. The reader is also assumed to be familiar with the basis for standard model-based specifications but not with any particular specification language; RESOLVE [27] is used for specification examples, but the notation is explained right here.

Section 2 discusses the difference between value and reference variables, which might seem so well known as to go without saying. (The reason for saying it anyway is detailed in Section 5.2.) Section 3 describes the serious impact of this distinction on the complexity of behavioral specifications, and Section 4 describes the impact on modular verification. Section 5 discusses related work. Section 6 presents our conclusions.

2. VALUES VS. REFERENCES

Popular object-oriented languages, including C++, Eiffel, and Java, share a bizarre feature. They create a dichotomy between two kinds of types and, therefore, two kinds of variables:

- ***Value variables***, which stand for values of the built-in types (***value types***) such as *boolean*, *char*, and *int*.
- ***Reference variables***, which stand for references to objects whose values are of types (***reference types***) introduced through interfaces and classes.

Why is this dichotomy "bizarre"? It clearly is not intuitive, which is obvious if you have ever tried to explain and justify

it to students. Parsimony certainly suggests having only value variables or only reference variables, not both.

Knowing Hoare's hints on programming language design and recognizing the elegance of some purely functional programming languages, the C++, Eiffel, and Java designers must have preferred to have only value variables, all other things being equal. But all other things are not equal. For one thing, there is the folk theorem about indirection. In fact, the use of indirection is a little like the use of tobacco: an addictive bad habit. Modern programming languages have contributed to the problem by making indirection harder and harder to avoid and programs using indirection easier and easier to write. Reference variables are everywhere in Java yet carry no syntactic baggage at all! So surely it would be considered sacrilege to remove easy indirection from any modern imperative language—even though the effect of indirection, when truly appropriate as it is occasionally, could be provided by a small set of library components offering comparable power and performance profiles to language-provided pointers [12, 14].

Of course, tradition is not the reason these popular languages distinguish between values and references. Language designers simply failed to discover another way to make programs *efficient* in terms of execution time and storage usage [11]. Value variables can be represented with small chunks of storage that can easily be copied, leaving *x* and *y* completely independent in code following the assignment statement here:

```
int x;
int y;
...
y = x;
```

If user-defined types were value types that behaved like *int*s, then this kind of code could be terribly inefficient. For example, suppose *x* and *y* were value variables in the following Java code—remember they are not—and so would remain independent in code following the assignment statement:

```
SetOfInt x = new SetOfInt ();
SetOfInt y = new SetOfInt ();
...
y = x;
```

The assignment would then entail deep copying of a *SetOfInt* object representation, which presumably would take time linear in the size of the set *x*. Overriding the assignment operator to make a deep copy is recommended practice for C++ programmers who use the Standard Template Library [24], precisely because this leaves *x* and *y* independent of each other following the assignment. The Java assignment operator, on the other hand, cannot be overridden. An optional *clone* method is supposed to make a deep copy (but it doesn't, in fact, even for the collections in the popular java.util package).

Having reference variables directly addresses the performance problems involved in copying large data structures because:

- the representations of all variables remain small, i.e., the size of one pointer each, although every reference variable still refers to an object whose representation is a potentially large data structure; and
- the assignment statement is fast for both value and reference variables.

Most of the rest of this paper discusses the price paid for following this road to efficiency: complications in specification and verification, and therefore in understanding and reasoning about program behavior. The appendix (adapted from [14]) briefly explains the *swapping paradigm*, an alternative approach that permits the same efficiency to be achieved without introducing references into the language model, specifications, or programmer reasoning. The purpose of the appendix is to suggest that there are other solutions to the apparent reasoning vs. efficiency trade-off, i.e., that the choice is not limited to a pure functional programming paradigm (reasoning over efficiency) or the standard object-oriented programming paradigm (efficiency over reasoning).

3. IMPACT ON SPECIFICATION

Simply introducing reference types into a language model makes it harder for clients to understand the specified behavior of components—if such behavior were carefully specified, which in practice (e.g., Java component libraries) it is not. This section illustrates the additional complication by describing a reference type in a model-based specification language, RESOLVE, that is designed for specifying value types [27]. That is, there should be no syntactic sugar through which the specification language might mask the fact that there is a reference type. This approach allows an apples-to-apples comparison of the underlying "intellectual load" introduced by value vs. reference types, both on component specifiers and on clients of those specifications.

Wouldn't it be fair to (also?) select a specification language that is designed to handle reference types, and use it to try to specify value types? Not really. Variables in traditional mathematics stand for values; they do *not* stand for references to objects that have values. In other words, a hypothetical specification language that is designed to hide references behind syntactic sugar must still, in the final analysis, "mean" (i.e., have its semantics) in the domain of traditional mathematics. The verification conditions arising in correctness proofs must be stated in traditional mathematics in order that proofs can be carried out. "Desugaring" from references to values is, therefore, ultimately required. It is only in the desugared version of this hypothetical specification language that we could really compare the relative difficulties values and references pose for specification writers and readers.

3.1 Defining Mathematical Models for Types

Let's start with a simple case: specifying the mathematical model for a built-in value type. For example, for type *int* in Java the obvious mathematical model is a mathematical integer constrained to be within some bounds. In RESOLVE notation, this is expressed as follows:

```
type int is modeled by integer
  exemplar i
  constraint
    -2147483648 <= i <= 2147483647
```

The *exemplar* clause introduces an arbitrary name for a prototypical variable of the new type, and the *constraint* clause is an assertion that describes such a variable's value space. So, the meaning of this specification is that in reasoning about Java code such as that shown earlier using *int* variables, you should think of the values of *x* and *y* as being mathematical integers like 1372 and –49 (i.e., not as strings of 32 bits).

3.1.1 Value Type Specification

A similar scenario arises for a type such as *SetOfInt* whose mathematical model is more complex and whose representation is potentially large. For example, if *SetOfInt* were a value

type in the earlier Java code—remember it is not—then you would want to think of the values of *x* and *y* as being sets of numbers like {1, 34, 16, 13} and {2, –9, 45, 67, 15, 16, 942, 0}. The mathematical model specification would look like this:

```
type SetOfInt is modeled by
    finite set of integer
  exemplar s
  constraint
    for all k: integer where (k is in s)
      (-2147483648 <= k <= 2147483647)
  initialization ensures
    s = {}
```

The *initialization* clause says that when a new *SetOfInt* variable is declared, its value is the empty set.

3.1.2 Reference Type Specification

Unfortunately, life is not so simple in Java: *SetOfInt* is a reference type. In order to reason soundly about what your programs do, you must think of the values of *x* and *y* as being references to objects whose values are sets of numbers like {1, 34, 16, 13} and {2, –9, 45, 67, 15, 16, 942, 0}. That is, the fact that this is a reference type must be made explicit in the type's mathematical model specification. How can this be done?

Without syntactic sugar to hide references, the obvious approach (known to many others) is to model the mapping of references to sets of integers as a mathematical function whose scope is global to all *SetOfInt* variables. In RESOLVE, you can say this using abstract *state variables* that may be accessed and updated in (the specification of) any method associated with any variable of the type being specified. An appropriate mathematical model can be expressed as follows; there are other ways to do it but this is the simplest one we know:

```
state variables
    last: integer
    objval: function from integer to
              finite set of integer
  constraint
    for all r: integer
      (for all k: integer
          where (k is in objval(r))
        (-2147483648 <= k <= 2147483647))
  initialization ensures
    for all r: integer (objval(r) = {})

type SetOfInt is modeled by integer
  exemplar s
  initialization ensures
    last = #last + 1  and
    objval = #objval  and
    s = last
```

The state variable *last* is an abstraction of the address held in a *SetOfInt* variable. Its purpose is to ensure that a newly constructed *SetOfInt* object is independent of all others. The starting value of *last* does not matter because, each time a new *SetOfInt* object is constructed, the value of *last* is incremented ("#" before a variable name denotes the old value). Since *last* is an abstract variable, there is no need to worry about eventual overflow.

The value *null*, however, is an annoying problem: there must be some way to tell it apart from other values. This can be handled in the above model with a minor change:

```
  initialization ensures
    last = 0  and  ...
```

in which case null is modeled by 0. Throughout the rest of this paper, however, we ignore the possibility of null references. There are two reasons. First, we are trying to evaluate how little additional trouble is necessarily entailed by having reference types. Allowing null references only makes method specifications messier, i.e., what happens for null and what happens for non-null values of all the method parameters that are of reference types. Second, it might be possible in principle to have a language in which there were reference types but no null references. Java is used here for illustration, but we don't want to limit observations about reference types to Java.

The state variable *objval* is an abstraction of the mapping between references and the values of the objects they refer to. Again, *objval* is an abstract mathematical variable, so there is no problem that it (or, for that matter, *last*) has a value from a mathematical domain that is manifestly too large to represent. In this specification, we decided to initialize *objval* so every possible reference is mapped to an empty set. The illusion is that there is an infinite pool of objects whose values are empty sets of integers, and that every time a new *SetOfInt* object is constructed, one of these pre-formed objects is selected from that pool. There are other ways to model the situation, of course, but none is any simpler or cleaner when written out.

It is already evident that the mathematical machinery involved in modeling the reference type is significantly more complex than that needed to model the corresponding value type. But this is only part of the problem; there remains the issue of specifying the behavior of methods.

3.2 Defining Method Behavior

Let's consider a method to add an *int* to a *SetOfInt*:

```
public void addInt (int i);
```

3.2.1 Value Type Specification

If *SetOfInt* were a value type in Java—remember it is not—then the specification for *addInt* might look like this:

```
evaluates i
updates self
requires
  i is not in self
ensures
  self = #self union {i}
```

Before the precondition (*requires* clause) and the postcondition (*ensures* clause), the lists of variables classify each variable in scope as either unchanged (*restores* or *evaluates* list) or potentially modified (*updates* or *replaces* list). "Restores" means that the abstract value of the parameter undergoes no net change from call to return, but it might be modified temporarily while the method is operating. Because *i* is passed by value in Java, and the corresponding actual parameter is treated as an expression, *i* is listed as having *evaluates* mode.

3.2.2 Reference Type Specification

Here is what happens because *SetOfInt* is really a reference type:

```
evaluates i
restores self, last
updates objval
requires
  i is not in objval(self)
ensures
  objval(self) = #objval(self) union {i} and
  for all r: integer where (r /= self)
    (objval(r) = #objval(r))
```

Note that *self* is not changed because it is a reference. But the *SetOfInt* object it refers to has its value (i.e., *objval*(*self*)) changed. The last clause of the postcondition says that no other *SetOfInt* object has its value changed.

All the other public methods for *SetOfInt* have specifications with the same flavor as *addInt*. So, all of this is "boilerplate":

```
restores self, last
updates objval
ensures
  for all r: integer where (r /= self)
    (objval(r) = #objval(r))
```

By making these oft-repeated specification clauses implicit with a wave of the hand, it is possible to create a specification language with enough syntactic sugar to simplify the *look* of a specification for a reference type. In ESC/Modula-3 [19], for example, variables not in a "modifies" list are preserved, and the value of a referenced object (e.g., *objval*(*self*)) can be listed as though it were a variable name, so the short version of the above statements is (in RESOLVE-like syntax) just:

```
updates objval(self)
```

This does not materially change the intellectual task of understanding the *meaning* of the specification, however. And as noted in Section 4, the underlying additional complication of references reveals itself once you start relying on that specification to try to reason about client code that uses *SetOfInt*.

3.3 Assignment

It is instructive to specify the behavior of the Java assignment operator, prototypically of the following form:

```
lhs = rhs;
```

3.3.1 Value Type Specification

If *SetOfInt* were a value type in Java—remember it is not—then the specification would be:

```
evaluates rhs
replaces lhs
ensures
  rhs = lhs
```

Note that the "=" in the specification is not itself an assignment operator, but denotes the assertion of ordinary mathematical equality between the mathematical models of *lhs* and *rhs*. We have written "rhs = lhs" rather than the equivalent "lhs = rhs" to emphasize this, any ambiguity being removed by the specification that *rhs* is merely evaluated. The confusing use of "=" as an assignment operator is an unfortunate design choice that crept from Fortran back into C after having been nearly eradicated by ":=" in Algol-like languages.

3.3.2 Reference Type Specification

Interestingly, the assignment specification looks virtually identical for *SetOfInt* as a reference type, the only difference being that *last* and *objval* are also listed as being unchanged:

```
evaluates rhs
restores last, objval
replaces lhs
ensures
  rhs = lhs
```

Maybe there is some comfort in knowing that the assignment operator does "the same thing" for value and reference variables. Of course, the only reason it does "the same thing" is that the mathematical model for a reference type makes the value of a reference variable explicit and distinct from the value of the object it refers to. The assignment operator simply copies the value of the (value or reference) variable on the right-hand side to that on the left-hand side.

4. IMPACT ON MODULAR VERIFICATION

It is widely acknowledged that practical verification must be modular, a.k.a. compositional. Factoring of the verification task cuts along the lines of programming-by-contract [22]. That is, a component implementation is verified against its specification once and for all, out of the context of the client programs that might use it. The legitimacy of client use of a component implementation is gauged during verification of the client, based on knowledge of only the component specification, i.e., without "peeking inside" the separately-verified component implementation and without reverifying any part of it on a per-use basis.

The primary verification issue for software with references stems from the possibility of aliasing: having two or more references to the same object. Aliasing can arise either from reference assignment (the case considered here) or from parameter-passing anomalies (the case Cook considered in his study of Hoare logic [3]; see also [13, 17]). The challenge here is to discover how the specification of *SetOfInt* in Section 3 might be used in modular verification of a client of *SetOfInt*, if the client program could execute a reference assignment.

Let's consider a relatively simple situation where the client program is a main program having two "helper" operations *P* and *Q* with specifications not shown:

```
import Section3.SetOfInt;
class Client {
  private static void P (SetOfInt si) {
    ...
  }
  private static int Q (int i) {
    ...
  }
  public static void main (...) {
    int j, k;
    SetOfInt s1 = new SetOfInt();
    SetOfInt s2 = new SetOfInt();
    ...
    P(s1);
    ...
    k = Q(j);
    ...                              // point A
    P(s2);                           // point B
    ...
  }
}
```

Suppose this program uses no other classes or constructs that might cause modular verification problems, so the focus is entirely on the impact of using *SetOfInt*. In other words, suppose *main*, *P*, and *Q* could be verified independently except for any effects introduced by using *SetOfInt*.

4.1 Value Type Verification

If *SetOfInt* were a value type in Java—remember it is not—then variables of this type could be passed from *main* to *P* without fear that modularity might be compromised. The specification of *SetOfInt* as a value type makes this clear. There are no state variables in that specification and, consequently, no shared

state would be introduced among *main*, *P*, and *Q* as a result of their common visibility over the *SetOfInt* class. For example, suppose the intended behavior of *P* were this:

```
updates si
ensures
  si = #si union {13}
```

You would be able to reason about the correctness of the body of *P* independently of the bodies of *main* and *Q* because there would be nothing *P*'s body could do to the values of any variables in the program other than the argument passed for the formal *si* in a given call. The same would be true of the bodies of *main* and *Q*. Reasoning would remain modular even with this user-defined type in the picture—*if* it were a value type.

4.2 Reference Type Verification

In truth, *SetOfInt* is a reference type. But suppose, in a fit of wishful thinking, you decided that it didn't matter that much and made the simplification of thinking of *SetOfInt* as a value type. Given the specification above, you might expect *P* to have the following body:

```
if (! si.contains (13)) {
  si.addInt (13);
}
```

The problem with your thinking would be that *P* has visibility over the reference type *SetOfInt*, including the abstract state variables *last* and *objval*. Through them *P* might do other things. For example, *P* might copy and save the reference *s1* that *main* passes in the first call, and then quietly change *objval*(*s1*) through that alias during the next call. If you erroneously thought of *SetOfInt* as a value type, then it would seem that the value of *s1* changed spontaneously between the points labeled "A" and "B" in *main* even though the variable *s1* was not even mentioned in the statement executed between them. In reality, of course, what was changing was *objval*(*s1*); but by hypothesis you were oblivious to the abstract state variable *objval* and were thinking of *si* as a value variable— a "no-no".

So, the following might be the body of *P*. It also seems to satisfy the specification above in terms of its effect on *si*, if you treat *si* as a value variable and thereby ignore *objval*. Here, *Alias* is a simple class with two static methods, *saveTheAlias* and *theAlias*, which copy an *Object* reference and return the copy, respectively. The point is that nowhere outside the body of *P* is there even a hint that an alias is being kept inside it.

```
if (Alias.theAlias () != null) {
  ((SetOfInt) Alias.theAlias ()).clear ();
}
Alias.saveTheAlias (si);
if (! si.contains (13)) {
  si.addInt (13);
}
```

In reasoning about the body of *main*, how could you predict the strange behavior resulting from this code without examining the body of *P*—and thereby giving up modular reasoning? The key to salvaging modularity is to realize that the specification of *SetOfInt* as a reference type involves two abstract state variables, *last* and *objval*, that are visible throughout *main*, *P*, and *Q*. From the reasoning standpoint, there are variables in this program that are global to *main*, *P*, and *Q*, although the syntax of Java does a great job of hiding them.

Now *main* still can be verified independently of *P* and *Q* despite sharing *last* and *objval* with them. The specifications of *P* and *Q* simply must describe their effects on the abstract state variables *last* and *objval* as well as on their explicit parameters. *P*'s specification should be changed to this:

```
evaluates si
restores last
updates objval
ensures
  objval(si) = objval(#si) union {13}  and
  for all r: integer where (r /= si)
    (objval(r) = #objval(r))
```

Knowing only that *P* preserves *last* does not allow the verifier of *main* to be sure that *P* cannot create an alias by copying *si* and then changing the object value later. But the "nothing else changes" clause in the postcondition prevents a correct body for *P* from doing anything funny with an alias (like the second body above) even if it saves one.

Another possibility is that maybe the above specification isn't really what is wanted! Perhaps the weird implementation of *P* is correct according to the programmer's intent, and the problem is specifying what *P* is supposed to do. Such a situation also can be handled in this specification framework.

This example shows why it is critical for sound reasoning that a programmer not imagine and/or hope that reference variables are sort of like value variables. They aren't.

Can *Q* be verified independently of *main* and *P* despite sharing *last* and *objval* with them? Here, *main* and *P* can manipulate *last* and *objval* by executing any series of *SetOfInt* method calls. It turns out that *Q* cannot see the effects of those manipulations even if it declares and uses *SetOfInt* variables of its own—and vice versa. But the basis for this claim is not clearly evident from the specification of *SetOfInt*. It is a consequence of a special "non-interference" property that arises from the way the *SetOfInt* specification uses the abstract state variables: Neither of two methods declaring their own *SetOfInt* variables but otherwise not communicating with each other can detect changes that are made by the other to the abstract state variables. So, curiously, *Q* can be verified independently of *main* and *P* in this case even if its specification does not include a "nothing else changes" clause.

5. RELATED WORK

Following the early papers cited in Section 1, there have been some interesting recent episodes in the literature on programming language design, specification, and verification. They suggest a fundamental struggle between acknowledging the folklore about the importance and power of indirection, and the reasoning problems arising from its use. We briefly review two language designs, the cases of C++ and Java, in Sections 5.1 and 5.2, respectively. Other researchers have investigated some of the specification and verification difficulties arising from pointers. We briefly discuss their work in Section 5.3.

5.1 C++

C++ makes a distinction between pointers and references, as explained by Bjarne Stroustrup, the creator of C++ [30]:

> A *reference* is an alternative name for an object. The main use of references is for specifying arguments and return values for functions in general and for overloaded operators... [T]he value of a reference cannot be changed after initialization; it always refers to the object it was initialized to denote.

That is, references were introduced into C++ primarily to simplify parameter passing and overload resolution. These programming language concerns had nothing to do with trying to address the reasoning problems that arise from using pointers. Indeed, C++ still has pointers, too.

The decision to complicate C++ by not only introducing references, but making them different from pointers in a rather subtle way, might seem to be another "step backward". But other language features combine with references to give the C++ programmer the flexibility to change the default programming model from reference-oriented to value-oriented. That is, it turns out it is quite possible in C++ to keep pointers and references from bubbling up through component (class) interfaces where they must be faced by clients reading specifications and verifying client code. One of these extra features is the ability to override the assignment operator and copy constructor so they make deep copies, not merely copies of references.

The problem is that there is a performance penalty for making deep copies, as discussed earlier. Luckily, the flexibility of C++ does not stop there. It is also possible to make both the assignment operator and copy constructor private, so they are simply unavailable to clients of a class.

We have taken advantage of the latter feature (and several others) to create a disciplined style of programming in C++, the RESOLVE/C++ discipline [14, 33], in which adherence to many rules of the discipline is compiler-checked by C++ itself. The bottom line is that you can program in C++ using what are technically reference variables yet maintain the *illusion* that you have only value variables. To achieve this, we introduced the swap operator [9] to replace the private assignment operator and copy constructor. Then we designed a large library of class templates [25] whose formal specifications allow clients to reason modularly about client code [14, 33]. The RESOLVE/C++ discipline has been shown to be rather easily understandable and usable by introductory CS students [20, 28, 29] and has been shown to result in dramatically good code quality when used to build a commercial software system [14]. See the appendix for a brief discussion of the key idea behind the discipline, i.e., the swapping paradigm.

5.2 Java

By the time Java was born, Sun Microsystems apparently sensed that people were worried about the "safety" of their programming languages. Thus, the conservatism of Java's design was heavily stressed. In the first paragraph of *The Java Language Specification*, James Gosling, Bill Joy, and Guy Steele wrote [7]:

> Java is intended to be a production language, not a research language, and so, as C. A. R. Hoare suggested in his classic paper on language design, the design of Java has avoided including new and untested features.

Some of the early literature about Java also argued that it did not have certain old and well tested but known-to-be-dangerous features—like pointers. For example, consider this passage written by Gosling and Henry McGilton in their 1996 white paper on *The Java Language Environment* [8]:

> [P]ointers are one of the primary features that enable programmers to put bugs into their code. Given that structures are gone, and arrays and strings are objects, the need for pointers to these constructs goes away. Thus the Java language has no pointers.

Later, it became clear that this claim was a bit of an overstatement, or at least that it could be considered correct only in the legalistic sense that Java does not have pointer *syntax*. Of course, it has pointers almost everywhere, but it calls them references. The potential for confusion was addressed by Sun Microsystems itself in its on-line Java FAQ [31]:

> How can I program linked lists if there are no pointers?
>
> [Answer:] Of all the misconceptions about the Java programming language, this is the most egregious. Far from not having pointers, object-oriented programming is conducted in the Java programming language exclusively with pointers. In other words, objects are only ever accessed through pointers, never directly. The pointers are termed "references" and they are automatically dereferenced for you.
>
> > "An object is a class instance or an array. The reference values (often just references) are *pointers* to these objects." Java Language Specification, section 4.3.1. [emphasis is in the original text]
>
> Any book that claims Java does not have pointers is inconsistent with the Java reference specification.

Interestingly, then, some of Hoare's general advice about programming language design was heeded by the Java designers. But his specific warning about pointers was ignored, early claims to the contrary notwithstanding. By the way, what is the correct answer to the FAQ question, "How do I program linked lists?" You don't; you use java.util.List, or similar.

5.3 Specification and Verification

In the 1970s, several researchers addressed pointer specification and verification in the context of the precursors to object-oriented languages, notably Pascal. The culmination of this effort was reported in a 1979 paper by David Luckham and Nori Suzuki [21], where the modeling of the state of memory was made explicit in specifications and verification conditions in a slightly different way than we have done it. They introduced a mapping from the reference variable's textual name, not its mathematical model value (integer in our case), to the data value it pointed to. They would write the type-specific state variable we call *objval* in our example as *P#SetOfInt*, for "pointer to *SetOfInt*". Special notation also was introduced for dereferencing a pointer-to-*SetOfInt* variable *s* when writing assertions, i.e., *P#SetOfInt s* .

An important missing ingredient in this early work—apparently because Pascal lacked user-defined types with hidden representations—was any use of abstraction in explaining the behavior of new types. For example, in our *SetOfInt* specification as a reference type, as a client you may think of *objval*(*s*) as being a mathematical set of integers. In the Luckham/Suzuki style of specification, you would see not only the top-level reference complication but the pointers to the nodes in the (unhidden) data structure that represented the set. In other words, in 1979 and in Pascal, client component specifications for user-defined types exhibited all the complexity of specifications of reference types in Java, and then some. This was technically acceptable from the formal standpoint of verification but could not be used to give a fair comparison between specifying reference types and specifying value types because specifying reference types this way was even uglier than it needed to be, with no simplifying abstractions.

In 1980, George Ernst and Bill Ogden [5] considered similar specification and verification issues in Modula, which had a module construct with hidden exported types. They, therefore, needed to consider the question of how it was possible to hide reference types behind abstract specifications. They showed it was technically possible to hide references in module specifications through the use of some syntactic sugar in the specification language and an appropriate abstraction function in the module implementation. But the complexity moved over the horizon and into the proof rules:

> The only conceptual difficulty with the verification rules presented in this paper is that they do not prevent a procedure from side-effecting certain instances of abstract types which are not parameters to a call on it... [T]o verify a module, we must verify everything prescribed by the rule ..., but we must also verify that the side-effecting ... cannot occur. Developing such a rule is a non-trivial task ... beyond the scope of this paper.

One problem with showing that "the side-effecting ... cannot occur" is that it *can* occur according to the Modula language definition by assignment of a reference variable; even worse, some programmers *want* it to occur and write programs that way, and these programs might be correct, as noted in the example of Section 4.2. This means that hiding the complexity of references in a proof obligation stating that there is no aliasing causes a completeness problem.

In 1994, Ernst and Ogden, along with Ray Hookway, published a verification method for ADT realizations that handled "shared realizations" [6], including heap storage. Their approach to modeling references was essentially identical to our approach for reference types, with syntactic sugar hiding the abstract state variables that recorded the "serial number" of the last object constructed and the mapping from reference values to data values. A value-type specification was possible, with reference details arising only within the proof of the realization, because the only source of possible aliasing in the language was within realization code, i.e., not from client assignment of references. Moreover, the paper contained another caveat about the example used for specification and verification with a shared realization:

> The example does not use heap memory, because it would require extensive use of pointers, which would unnecessarily complicate both specification and verification...

So, the fundamental problem of how to verify programs with reference types seems technically solvable by making sure that the abstract state variables associated with reference variables "follow them around" throughout the proof. Everyone seems to agree that the introduction of references seriously complicates both specification and reasoning, though.

Other work that is directly related involves specification and verification of standard object-oriented software, where reference types are considered something we just have to learn to live with. The primary group in this area includes Gary Leavens, K. Rustan M. Leino, Peter Müller, and Arnd Poetzsch-Heffter, who have worked on similar issues both separately and in combination. They have tackled the problem of specifying behaviors of components involving pointers and references, and have dealt with potential aliasing both from copying of references and from parameter passing anomalies [4, 16, 17, 18, 19, 23]. Others (e.g., [2]) also have proposed ways to limit yet not eliminate aliasing through clever linguistic mechanisms. But these approaches have yet to be shown understandable by the real programmers they are supposed to enable to deal with references, and there is little evidence so far that this will be easily achieved. In other words, despite impressive technical advances that could contribute to the survival of reference types in our languages, these ideas still need to be validated against their ultimate objective: to become practically useful and comprehensible by real programmers.

Of course, we also have published some prior work in this area [13, 26, 32, 33, 34], including evidence of the comprehensibility and practical effectiveness of the RESOLVE discipline, which simply eliminates reference types [14, 28]. And Manfred Broy, like us, has generally suggested designing components for ease of specification, as opposed to writing "post-mortem" specifications for previously-designed components [1]. This would suggest simply avoiding reference types: the advice we'd get from Occam, too, were he still around.

To summarize, we were unable to find any work directly comparing the intellectual loads involved in specifying value types vs. reference types, or using such analysis to compare the difficulty in reasoning about client programs using them.

6. CONCLUSIONS

Adding references types to value types, as in Java, significantly and needlessly complicates standard model-based specifications and the modular verification they help enable. Technically, everything can be made to "work" with reference types. Reasonably concise and even plausibly comprehensible model-based specifications can be designed that account for the behavioral peculiarities arising from references.

By comparing the complexity of mathematical models and method specifications in a language that has no syntactic sugar to mask references, though, it becomes obvious that reference types introduce a substantially greater intellectual load than value types for both specifier and client. Writing specifications for reference types suggests obvious ways in which syntactic sugar can shorten the specifier's typing time, while still acknowledging a distinction between value types and reference types. Yet it is unlikely that such sugaring can in any way simplify the specifier's thinking or the client's ability to understand the specified behavior of reference types.

In general, modular verification remains possible in the face of reference types *if* the abstract state variables needed to specify a reference type are considered part of the state space of all units that have visibility over that type. For verification purposes the abstract state variables used in specifying a reference type can be treated like additional ghost parameters to all calls involving one or more explicit parameters of that type. Reasoning is, of course, far more complicated with these extra variables in the picture than it would be with value types only. But technically you can still have modular verification with reference types if there are no other language constructs that thwart modularity.

It remains common practice to encode indirection in Fortran by using arrays and integer indices as though they were dynamically-allocated storage pools and pointers into them. These arrays and integers are passed among subroutines as parameters, or sometimes placed in named common blocks that are visible to a selected subset of the subroutines in a program. All the subroutines that manipulate these arrays must agree on how they are using them in order to work correctly together. Over the years, programming language designers have simplified the syntax required to do this sort of thing, to the point

where in Java there is almost no syntax at all associated with indirection. But the underlying logic of programming with references in Java is the same as the logic of programming with arrays and indices in Fortran.

Is it, then, a good idea to hide the sharing of global state by making such language "advances"? This sharing is clearly evident in Fortran programs, but not in Java programs. But shared state introduced through references can remain hidden only from the *minds* of programmers who program without specifications and who never try to verify their programs. If specification language designers try to take the same road then they, too, will find they can hide this shared state only from the minds of programmers who never try to verify programs. Eventually, the emperor's thin disguise will reveal itself to all—although perhaps not before some catastrophic software failures cause more people to take a serious look at whether programming languages should offer constructs that are so well known to complicate specification and verification.

7. ACKNOWLEDGMENTS

Murali Sitaraman and Gary Leavens and their students, as well as the members of the OSU Reusable Software Research Group, have provided much food for thought through various personal and electronic discussions of some of the issues mentioned here. Scott Pike and the anonymous referees helped in many other ways, too, especially by suggesting how to focus the paper on its real thesis.

We gratefully acknowledge financial support from the National Science Foundation under grant CCR-0081596, and from Lucent Technologies. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the National Science Foundation or Lucent.

8. REFERENCES

[1] Broy, M. *Experiences with Software Specification and Verification Using LP, the Larch Proof Assistant*. Research Report 93, Compaq Systems Research Center, Palo Alto, CA, 1992.

[2] Clarke, D.G., Potter, J.M., and Noble, J. Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, ACM Press, 1998, 48-64.

[3] Cook, S.A. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing 7*, 1 (1978), 70-90.

[4] Egle, R. *Evaluating Larch/C++ as a Specification Language: A Case Study Using the Microsoft Foundation Class Library*. TR #95-17, Department of Computer Science, Iowa State University, Ames, IA, 1995.

[5] Ernst, G.W., and Ogden, W.F. Specification of abstract data types in MODULA. *ACM Transactions on Programming Languages and Systems 2*, 4 (1980), 522-543.

[6] Ernst, G.W., Hookway, R.J., and Ogden, W.F. Modular verification of data abstractions with shared realizations. *IEEE Transactions on Software Engineering 20*, 4 (1994), 288-207.

[7] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification.* Addison-Wesley, Reading, MA, 1996.

[8] Gosling, J., and McGilton, H. *The Java Language Environment: A White Paper.* Sun Microsystems, Inc., 1996; http://java.sun.com/docs/white/langenv/ viewed 8 August 2001.

[9] Harms, D.E., and Weide, B.W. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering 17*, 5 (1991), 424-435.

[10] Hoare, C.A.R. *Hints on Programming Language Design*. Stanford University Computer Science Department Technical Report No. CS-73-403, 1973. Reprinted in *Programming Languages: A Grand Tour*, E. Horowitz, ed., Computer Science Press, Rockville, MD, 1983, 31-40.

[11] Hogg, J., Lea, D., Holt, R., Wills, A., and de Champeaux, D. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, April 1992. http://gee.cs.oswego.edu/dl/aliasing/aliasing.html viewed 8 August 2001.

[12] Hollingsworth, J.E. and Weide, B.W. Engineering 'unbounded' reusable Ada generics. In *Proceedings of 10th Annual National Conference on Ada Technology*, 1992, ANCOST, 82-97.

[13] Hollingsworth, J.E. Uncontrolled reference semantics thwart local certifiability. In *Proceedings of the Sixth Annual Workshop on Software Reuse*, 1993.

[14] Hollingsworth, J.E., Blankenship, L., and Weide, B.W. Experience report: Using RESOLVE/C++ for commercial software. In *Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering*, 2000, ACM Press, 11-19.

[15] Kieburtz, R.B. Programming without pointer variables. In *Proceedings of the SIGPLAN '76 Conference on Data: Abstraction, Definition and Structure*, 1976. ACM Press.

[16] Leavens, G.T., and Cheon, Y. Extending CORBA IDL to specify behavior with Larch. In *OOPSLA '93 Workshop Proceedings: Specification of Behavioral Semantics in OO Information Modeling*, 77-80; also TR #93-20, Department of Computer Science, Iowa State University, Ames, IA, 1993.

[17] Leavens, G.T., and Antropova, O. *ACL — Eliminating Parameter Aliasing with Dynamic Dispatch*. TR #98-08a, Department of Computer Science, Iowa State University, Ames, IA, 1998.

[18] Leavens, G. T., Baker, A. L., and Ruby, C. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, eds. H. Kilov, B. Rumpe, and I. Simmonds, Kluwer Academic Publishers, Boston, MA, 1999.

[19] Leino, K.R.M., and Nelson, G. *Data Abstraction and Information Hiding*. Compaq SRC Rep. #160, 2000.

[20] Long, T.J., Weide, B. W., Bucci, P., Gibson, D. S., Hollingsworth, J., Sitaraman, M., and Edwards, S. Providing intellectual focus to CS1/CS2. In *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, 1998, ACM Press, 252-256.

[21] Luckham, D.C., and Suzuki, N. Verification of array, record, and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems 1*, 2 (1979), 226-244.

[22] Meyer, B. *Object-oriented Software Construction*. Prentice-Hall, New York, 1988; second edition, 1997.

[23] Müller, P., and Poetzsch-Heffter, A. Modular specification and verification techniques for object-oriented software components. In *Foundations of Component-Based Systems*, eds. G.T. Leavens and M. Sitaraman, Cambridge University Press, 2000, 137-159.

[24] Musser, D.R., Derge, G.J., and Saini, A. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, Upper Saddle River, NJ, 2001.

[25] RESOLVE/C++ Component Catalog Home Page. http://www.cis.ohio-state.edu/~weide/sce/rcpp/RESOLVE_Catalog-HTML viewed 8 August 2001.

[26] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J.E. Reasoning about software-component behavior. In *Software Reuse: Advances in Software Reusability (Proceedings Sixth International Conference on Software Reuse), LNCS 1844*, ed. W. Frakes, 2000, Springer-Verlag, 266-283.

[27] Sitaraman, M., and Weide, B.W. Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes 19*, 4 (1994), 21-67.

[28] Sitaraman, M., Long, T.J., Weide, B.W., Harner, J., and Wang, C. A formal approach to component-based software engineering: education and evaluation. In *Proceedings 2001 International Conference on Software Engineering*, 2001, IEEE, 601-609.

[29] Software Component Engineering Course Home Page. http://www.cis.ohio-state.edu/~weide/sce/now viewed 8 August 2001.

[30] Stroustrup, B. *The C++ Programming Language, 3rd edition*. Addison-Wesley, Reading, MA, 1997.

[31] Sun Microsystems, Java "Frequently Asked Questions". http://java.sun.com/people/linden/faq_b.html viewed 8 August 2001.

[32] Weide, B.W., Edwards, S.H., Harms, D.E., and Lamb, D.A. Design and specification of iterators using the swapping paradigm. *IEEE Transactions on Software Engineering 20*, 8 (1994), 631-643.

[33] Weide, B.W. *Software Component Engineering*. OSU Reprographics, Columbus, OH, 1996.

[34] Weide, B.W. "Modular regression testing": Connections to component-based software. In *Proceedings Fourth ICSE Workshop on Component-Based Software Engineering*, 2001, IEEE, 47-51.

9. APPENDIX: THE SWAPPING PARADIGM

How do you make some variable (say, *y*) get the value of another variable (say, *x*)? For example, suppose *x* and *y* are variables of type *int*, a value type whose mathematical model is a mathematical integer, as discussed in Section 3.1. Obviously, you use an assignment statement:

```
y = x;
```

What if *x* and *y* are variables of a value type *VT*, where *VT*'s mathematical model is relatively complex and its representation data structure is probably large? Suppose, for example, that *VT* is *SetOfInt*, whose mathematical model is a mathematical set of mathematical integers, as discussed in Section 3.1.1. There are now two options for data movement, neither of which is especially attractive:

1. Consider the assignment operator for *SetOfInt* to perform deep copy, so that after the assignment statement we can think of both *x* and *y* as having the same abstract value. Logically, *x* and *y* must behave independently, too, so changes to *x* do not side-effect the value of *y* and vice versa. This can be terribly inefficient, because without using fancy data-structure-specific tricks that frequently do not apply, the assignment operator must take time linear in the size of *x*'s representation. Big sets simply take a long time to copy and hence to assign.

2. Do not view *x* and *y* as value variables, but as reference variables; i.e., change their type from value type *VT* to reference type *RT*, and think of *x* and *y* as references to objects whose values are sets of integers. This fixes the efficiency problem but at the cost of a distressing non-uniformity in reasoning about program behavior: Some variables denote values and others denote references. It also means that the assignment operator creates aliases, which complicates formal specification and reasoning about program behavior, as explained in Section 4.2.

Approach #2 has been codified into most modern languages, notably Java. It is actually far worse than #1 from certain software engineering standpoints. One reason is that the programmer now must be aware that variables of some types have ordinary values while variables of other types hold references to objects (it's the objects that have the values). For template components this creates a special problem. Inside a component that is parameterized by a type *Item*, there is no way to know *before instantiation time* whether an assignment of one *Item* to another will assign a value or a reference. Of course, this can be "fixed" as it is in Java, by introducing otherwise-redundant reference types such as *Integer* to wrap value types such as *int*. Actual template parameters can then be limited to reference types. This is really ugly, though. And there is still the issue of the complication caused by references for specification and verification, as seen in Sections 3 and 4.

Figure 1 summarizes the data movement dilemma faced by someone who wants efficient software about whose behavior it is easy to reason. The conclusion is that this is only attainable

by sticking to built-in value types—not incidentally, the only types available when the assignment operator was introduced into programming languages—or, at best, by inventing only new user-defined types that admit "small" representations.

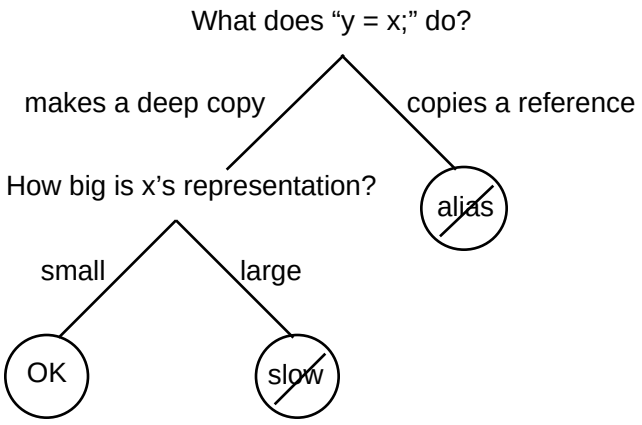


Figure 1: The Data Movement Dilemma

Again, how do you make some variable (say, *y*) get the value of another variable (say, *x*)? There is no inherent requirement that the value of *x* must not change as a result of the data movement process. Realizing this opens the door to other possibilities. The new value of *y* must be the old value of *x*, but the new value of *x* might be:

- the old value of *x* (to get this behavior we use assignment, which works well if *x*'s representation is small); or
- undefined; or
- a defined, but arbitrary and unknown, value of its type; or
- some particular value of its type, e.g., an initial value; or
- the old value of *y*.

It is beyond the scope of this appendix to analyze the pros and cons of all the possibilities beyond the first one, which is unsatisfactory as a general approach to data movement. Suffice to say that leaving *x* undefined complicates reasoning, although not nearly as much as allowing aliasing; and that leaving *x* with either an arbitrary or a distinguished value of its type is actually quite a reasonable thing to do. However, the last approach— *swapping* the values of *x* and *y*—is both efficient and safe with respect to modular reasoning, and it results in remarkably few changes to how most programmers write imperative code [9, 33].

You need to get used to a few new idioms when adopting the swapping paradigm, e.g., for iterating through a collection [32]. The biggest effect of the swapping paradigm, however, is on the design of component interfaces. Consider, for example, a *Set* component (parameterized by the *Item* type it contains) with operations *add*, *remove*, etc. What should *add*(*x*) do to the value of *x*? The analysis of this question parallels the analysis of the data movement dilemma as the question was phrased above. The conclusion is that *add* should *consume x*, i.e., it should leave *x* with an initial value of its type.

How can this be accomplished? A direct implementation of the *Set* component declares a new variable of the parametric type *Item* in the body of *add*, e.g., the data field in a new node that is to be inserted in a linked list of nodes. This variable is then swapped with *x*. Swapping simultaneously puts the old value of *x* into the *Set*'s representation data structure, where it needs to be; and sets the new value of *x* to the initial value for its type that was originally in the data field of the node.

What if there are no pointers in the language, though? In an implementation of the *Set* component that is layered on top of a provided *List* component, for example, the *add* operation simply inserts *x* at the appropriate place into the *List* that represents the *Set*. If the insertion operation for *List* also is designed using the swapping paradigm, so it consumes its argument just like *add* does, then this call does exactly what is needed.

In other words, in both these situations, the code that you would have written if using assignment for data movement is changed in just one respect: assignment of *x* to its place in the *Set*'s representation is replaced by swapping *x* with its place in the *Set*'s representation.

Our experience is that a family of components such as those in the RESOLVE/C++ component catalog [25] can be designed according to the swapping paradigm to compose in such a way that programming with swapping is substantially similar to programming with assignment statements. But the resulting components offer efficiency and/or reasoning advantages over similar components designed in a traditional fashion.

Let's be clear that we still use the assignment operator with built-in value types. There is nothing wrong with the following statement from either the efficiency or reasoning standpoints, assuming that *x* and *y* are variables of some value type with a "small" representation:

```
y = x;
```

The possibly surprising empirical observation that has been substantiated by commercial application development is that, with swapping, there is rarely a need for such a statement when *x* and *y* have user-defined types. You can have value types and efficiency at the same time.

The main advantages of the swapping paradigm are, then:

- The swapping paradigm is easy for imperative-language programmers to learn and apply.
- All types are value types, which allows for understanding of specifications and modular reasoning that are complicated significantly if reference types creep in.
- All pointers and references can be hidden deep within the bowels of a few low-level components and remain invisible to a client programmer layering new code on top of them.
- If these low-level components have no storage leaks, then client programs have no storage leaks, and client programmers do not have to worry about where to invoke **delete** in, e.g., C++, because they simply *never* invoke it. In the case of a garbage-collected language, e.g., Java, there is no need for the complications of general garbage collection because there are no aliases and all collection takes place at predictable times.

Other questions often asked about the interactions between the swapping paradigm and other programming language and software engineering issues, such as the role of function operations, assignment of function results to variables, parameter passing, etc., are discussed in [9].

Modular Verification of Performance Correctness

Joan Krone
Dept. Math. and Comp. Science
Denison University
Granville, OH 43023, USA
+1 740 587 6484
krone@denison.edu

William F. Ogden
Dept. Comp. & Info. Science
The Ohio State Universtiy
Columbus, OH 43210, USA
+1 614-292-5813
ogden@cis.ohio-state.edu

Murali Sitaraman
Dept. Comp. Science
Clemson University
Clemson, SC 29634, USA
+1 864 656 3444
murali@cs.clemson.edu

Abstract
Component-based software engineering is concerned with predictability in both functional and performance behavior, though most formal techniques have typically focused their attention on the former. The objective of this paper is to present specification-based proof rules compositional or modular verification of performance in addition to functionality, addressing both time and space constraints. The modularity of the system makes it possible to verify performance correctness of a module or procedure locally, relative to the procedure itself. The proposed rules can be automated and are intended to serve as part of a system of rules that accommodate a language sufficiently powerful to support component-based, object-oriented software.

Categories and Subject Descriptors
D.2.13 [Software Engineering]: formal specification and verification of software performance.

General Terms
Verification, assertive language, formal specifications.

Keywords
Proof rule, performance, time and space.

1. INTRODUCTION

Predictability is a fundamental goal of all engineering, including software engineering. To show that a program predictably provides specified functional behavior, a variety of ways to apply a system of proof rules to a program for proving functional correctness have been studied since Hoare's work. More recent efforts address the special challenge of modular reasoning for object oriented, component based software [1, 5, 8, 9, 12]. These systems depend on programmer-supplied assertions that serve as formal specifications for the functional behavior of the software. While correct functional behavior is critical to any software system, in order to achieve full predictability, we must ultimately address the issue of performance as well.

A program that carries out the right job, but takes longer than available time to complete is of limited value, especially in modern embedded systems. Similarly, a program that is functionally correct, but that requires more space than the system can provide is not useful either. Cheng, Clemens, and Woodside note the importance of the performance problem in their guest editorial on *Software and Performance* [21]:

> "Performance is a problem in many software development projects and anecdotal evidence suggests that it is one of the principal reasons behind cases where projects fail totally. There is a disconnect between techniques being developed for software analysis and design and the techniques that are available for performance analysis."

Measurement during execution (e.g., using run-time monitoring) is a common approach for analyzing performance of large-scale systems [21]. The objective of this paper is to present static analysis (and hence, a priori prediction) as an alternative to measurement. In particular, the focus is on *modular* or *compositional performance reasoning*: Reasoning about the (functionality and performance) behavior of a system using the (functionality and performance) *specifications* of the components of the system, without a need to examine or otherwise analyze the implementations of those components [17].

Compositionality is essential for all analysis, including time and space analysis, to scale up. To facilitate compositional performance reasoning, we have introduced notations for performance specifications elsewhere [18]. Given functionality and performance specifications (and other internal assertions such as invariants), the rest of this paper describes a proof system for modular verification. Section II sets up the framework to facilitate automated application of rules, using a simple example rule. Section III contains proof rules for verification of procedure bodies and procedure calls, involving possibly generic objects with abstract models as parameters. Section IV contains an example to illustrate a variety of issues involved in formal verification. Section V has a discussion of related work and summary.

2. ELEMENTS OF THE PROOF SYSTEM

Though the underlying principles presented in this paper are language-independent and are applicable to any assertive language that includes syntactic slots for specifications and internal assertions, to make the ideas concrete we use the RESOLVE notation [15, 16]. RESOLVE is intended for predictable component-based software engineering and it includes notations for writing specifications of generic components that permit multiple realizations (implementations) of those components. It also includes notations for specifying time and space behaviors of an implementation. The implementations include programmer-supplied representation

invariants, loop invariants, progress metrics, and other assertions depending on the structure.

The proof rules have been designed so that an automated clause generator can start at the end of a given assertive program and back over the code replacing the executable language constructs with assertions about the mathematical domain over which the program has been written. The clause generator produces a clause that is equivalent to the correctness of the given program. The clause can then be evaluated manually, automatically by a theorem prover, or by a combination to determine whether the clause is provable in the appropriate mathematical domain, and thereby whether the program is correct (with respect to its specification). To illustrate the ideas, we begin with a simple example. First we consider functional behavior and then address performance for the following piece of assertive code:

Assume x = 3;
x := x + 1;
Confirm x = 4;

Exactly how such an assertive code comes into place, given a specification and an implementation, is explained in Section III. In this code segment, the programmer has supplied a pre-condition indicated by the **Assume** keyword and a post-condition following the keyword **Confirm** with some (assertive) code in between. To prove the correctness of this segment, consider the following automatable proof rule for expression assignment:

C \ Code; **Evaluate** (exp); **Confirm** Outcome_Exp[x ⇝ **M_Exp**(exp)]

C \ Code; x := exp; **Confirm** Outcome_Exp;

In this rule, *C* on the left side of both the hypothesis and the conclusion stands for *Context* and it denotes the collection of whatever information is needed about the code in order to reason about its correctness. For example, the types of variables and the mathematical theories on which those types are based would be in the context.

In our example, the `Outcome_Exp` is "x = 4." The `Code` preceding the assignment is the assertion "**Assume** x = 3." In the assertive clauses, the 3 and 4 are the mathematical integers, while the assignment statement is performing an increment on a computer representation of an integer. (The use of mathematical integers in specifying computational Integer operations is documented in *Integer_Template* that specifies Integer objects and operations, and it is assumed to be in the context.)

Applying the proof rule on the example leads to the following assertive code:

Assume x = 3; **Evaluate**(x + 1); **Confirm** x + 1 = 4.

This is the result of substituting the expression "x + 1" for x, the meaning of [x ⇝ **M_Exp**(exp)]. **M_Exp** denotes putting in the mathematical expression that corresponds to the programming expression, thus keeping our assertions over mathematical entities, rather than programming ones. There is a rule for **Evaluate** that causes the expression to be evaluated by the verifier. Similarly, the verifier would simply continue backing through the rest of the code, applying appropriate proof rules, eliminating one more constructs in each step.

Now we augment the above rule to prove functional correctness, with performance-related assertions. Suppose we need to prove the correctness of the following assertive code:

Assume x = 3 ^ **Cum_Dur** = 0 ^ **Prior_Max_Aug** = 0 ^ **Cur_Aug** = 0;
x = x + 1;
Confirm x = 4 ^ **Cum_Dur** + 0.0 = $D_{:=}$ + D_{Int_+} ^
Max(**Prior_Max_Aug**, **Cur_Aug** + 0) ≤ $S_{:=}$[1];

Here, $D_{:=}$ denotes the duration for expression assignment[2] (excluding the time to evaluate the expression itself). $S_{:=}$ denotes storage space requirement for expression assignment (excluding the storage space needed to evaluate the expression itself and the storage for variable declaration of x which is outside the above code). The units for time and space are assumed to be consistent, though we make no assumptions about the units themselves. The rest of the terms (whose need may not become fully clear until after the discussion of procedures in Section III) are explained in the context of the following rule for expression assignment:

C \ Code; **Evaluate**(exp); **Confirm** (Outcome_Exp ∧
Cum_Dur + $D_{:=}$ + Sqnt_Dur_Exp ≤ Dur_Bd_Exp∧
Max(**Prior_Max_Aug**, **Cur_Aug** + $S_{:=}$ +
Fut_Sup_Disp_Exp) ≤ Aug_Bd_Exp))
[x⇝**M_Exp**(exp)];

C \ Code; x := exp; **Confirm** Outcome_Exp ∧ **Cum_Dur** + Sqnt_Dur_Exp ≤ Dur_Bd_Exp ∧
Max(**Prior_Max_Aug**, **Cur_Aug** + Fut_Sup_Disp_Exp) ≤ Aug_Bd_Exp;

The new rule includes everything needed for functional correctness, and also includes new clauses about time and space performance. In spite of past attempts in the literature, it is just not possible to develop rules for performance correctness independently of functional correctness, because in general, performance depends on values of variables (which come from analyzing functional behavior) [17, 18]. In the example and in the rule, terms in bold print are keywords and the terms ending with "_Exp" represent expressions to be supplied by the programmer and kept up to date by the verifier.

---

[1] We have added terms "+ 0.0" and "+ 0" in the expressions here so that it is easy to match the syntactic structure of the rule given next.

[2] In RESOLVE, the right hand side of an assignment statement is restricted to be an expression. In particular, x := y is not allowed on variables of arbitrary types. For copying y to x, the assignment statement needs to be x := Replica(y). This interpretation is implicit for (easily) replicable objects such as Integers for programming convenience. This is what justifies the time analysis in the present rule. To move the value of y to x efficiently on all objects large and small, and without introducing aliasing, RESOLVE supports swapping (denoted by ":=:") as the built-in data movement operation on all objects [3].

First we consider timing. The keyword **Cum_Dur** suggests cumulative duration. At the beginning of a program the cumulative duration would be zero. As the program executes, the duration increases as each construct requires some amount of time to complete. The programmer supplies an over all duration bound expression, noted by `Dur_Bd_Exp`. This is some expression over variables of the program that indicates an amount of time acceptable for the completion of the program. As the verifier automatically steps backward through the code, that expression gets updated with proper variable substitutions as the proof rules indicate.

For example, in the above rule, when the verifier steps backward over an assignment, the variable, "x," receiving the assignment is replaced by the mathematical form of the given expression, "exp," in all of the expressions included within the parentheses.

`Sqnt_Dur_Exp` stands for the subsequent duration expression, an expression for how much time the program will take starting at this point. This expression is updated also automatically by the verifier, along with other expressions in the rule.

The duration (timing) for a program is clearly an accumulative value, i.e., each new construct simply adds additional duration to what was already present. On the other hand, storage space is not a simple additive quantity. As a program executes, the declaration of new variables will cause sudden, possibly sharp, increases in amount of space needed by the program. At the end of any given block, depending on memory management, storage space for variables local to the block, may be returned to some common storage facility, causing a possibly sharp decrease in space.

The right operation for duration is addition and for storage it turns out to be taking the maximum over any given block. It is reasonable to assume that for any given program, there will be a certain amount of space needed for getting the program started. This will include the program code itself, since the code will reside in memory. Assuming real, rather than virtual memory, the code will take up a fixed amount of space throughout the execution. With this in mind, we think of some fixed amount of space for any given program that remains in use throughout the execution. Our rules are written to deal with the space that augments the fixed storage and increases and decreases as the program executes. **Prior_Max_Aug** stands for "prior maximum augmentation" of space. At the beginning of any program, the prior maximum will be zero, since only the fixed storage is in use. As the program executes, over each block, a maximum of storage for that block is taken to be the **Prior_Max_Aug**. At any point in the program, there will be a storage amount over the fixed storage. We call that the current augmentation of space, **Cur_Aug**. Of course, there will be some overall storage bound to represent what is acceptable. We call that the augmentation bound expression, Aug_Bd_Exp. Finally, just as there was an expression to represent how much additional time would be needed, there is an expression for how much storage (displacement) will be needed in the future, the future supplementary displacement expression, `Fut_Sup_Disp_Exp`.

3. PROCEDURES

We examine a more complicated procedure construct in this section, having introduced basic terminology using the expression assignment proof rule. We present a rule for procedure declarations and one for procedure calls. These rules apply not only to ordinary code when all variables and types are previously defined, but to generic code as well, i.e., code written for variables that have not yet been tied to a particular type or value. This capability to handle generic code is critical for reusable, object-based components.

3.1 Procedure Declaration Rule

Associated with every procedure is a heading that includes the name, the parameter list, and assertions that describe both functional and performance behavior:

P_Heading:

Operation P(**updates** x: T);
requires P_Usg_Exp/_ x _\;
ensures P_Rslt_Exp/_ x, #x _\;
duration Dur_Exp/_ x, #x _\;
manip_disp M_D_Exp/_ x, #x _\;

This heading is a formal specification for procedure P. We use separate keywords **Operation** to denote the specification and **Procedure** to denote executable code that purports to implement an operation. We have included only one parameter on the argument list, but of course, if there were more, they would be treated according to whatever parameter mode were to be indicated. The **updates** mode means that the variable is to be updated, i.e., possibly changed during execution.

In the heading, the type `T` may be a type already pinned down in the program elsewhere, or it might represent a generic type that remains abstract at this point. The **requires** and **ensures** clauses are pre and post conditions respectively for the behavior of the operation, and the angle brackets hold arguments on which the clauses might be dependent. Due to page constraints, the rule does not include other potential dependencies such as on global variables.

Details of performance specification are given in [18]. **Duration** is the keyword for timing. `Dur_Exp` is a programmer-supplied expression that describes how much time the procedure may take. That expression may be given in terms of other procedures that `P` calls and it may be phrased in terms of the variables that the operation is designed to affect. We may need to refer both to the incoming value of *x* and to the resulting value of `x` in these clauses. We distinguish them by using `#x` for the value of `x` at the beginning of the procedure and `x` as the updated value when the procedure has completed. The last part of the Operation heading involves storage specification. Here, **manip_disp** (termed **trans_disp** in [18]) suggests manipulation displacement, i.e., how much space the procedure may manipulate as it executes.

Given the operation heading, we next consider a rule for a procedure declaration to implement an operation.

C ∪ {P_Heading} \ **Assume** P_Usg_Exp ∧
Cur_Dur = 0.0 ∧
Prior_Max_Aug = **Cur_Aug** = **Disp**(x);
P_Body;
Confirm P_Rslt_Exp ∧ **Cur_Dur** + 0.0 ≤ Dur_Exp ∧
Max(**Prior_Max_Aug**, **Cur_Aug** + 0) ≤ M_D_Exp;
C ∪ {P_Heading} \ Code; **Confirm** Outcome_Exp;

C \ P_Heading; **Procedure** P_Body; **end** P;
Code; **Confirm** Outcome_Exp;

As in the assignment rule, `C` stands for the context in which the procedure occurs. Note that `P_Heading`, the specification of Operation P, is added to the context making it possible for reasoning about the procedure to take place. The conclusion line of the rule allows the procedure declaration to be made and followed by some code and a clause to confirm after the code.

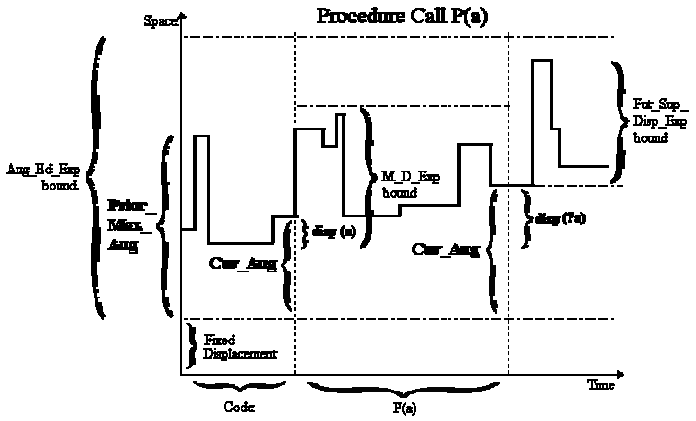
The hypotheses of the rule indicate that the procedure is to be examined abstractly, proving that no matter what value for the parameter is passed in, the result will satisfy both the functional and performance requirements.

The first hypothesis checks functional behavior by showing that if the **requires** clause is met, then the **ensures** clause is satisfied upon completion of the procedure body. For timing, we set the **Cum_Dur** to 0 thereby localizing the proof to just this procedure, avoiding the pitfall of having to consider the entire program when proving correctness for just this procedure. After the procedure body, we confirm that the **Cum_Dur** remains below Dur_Exp, the bound expression given in the specifications. It is assumed that the **Cum_Dur** acts like an auxiliary variable updated automatically at each step.

Finally, we address the storage requirements. Before the procedure body, we set the **Prior_Max_Aug** and the **Cur_Aug** both to be the amount of space required by the parameter, `x`. (Alternatively, the displacement of parameters at the beginning could be subtracted at the end.) This is necessary to retain the local nature of the proof process. The only concern that the procedure rule has about space is what the procedure uses above what has already been used in the past and what might be used in the future. After the body, the rule checks that the max over the stated values is within the specified bound.

3.2 Procedure Call Rule

A picture serves to motivate space-related assertions in the procedure call rule. The timing aspects of the rule are more straightforward and they are not shown in this picture.



Along the lower part of the picture the "fixed displacement" represents some amount of storage necessary for the program to run, an amount that does not vary throughout execution. The code itself is included in this fixed storage. Above the fixed storage the execution of the code requires a fluctuating amount of space, increasing when storage for new variables is allocated and decreasing when it is released.

The auxiliary variable, **Cur_Aug**, represents at any point what the current amount of storage is over and above the fixed storage. Note that the same variable appears twice on the picture, once at the place where a call to procedure `P` is made and again at the point of completion of `P`. **Cur_Aug** has a value at every point in the program and is continually updated. Similarly, as the execution procedes, **Prior_Max_Aug** keeps track of the maximum storage used during any interval. In the picture at the point where the call P(a) is made, **Cur_Aug** is shown, as is **Prior_Max_Aug**. Of course, as the code execution progresses, the value for **Prior_Max_Aug** is updated whenever a new peak in storage use occurs.

Within the procedure body, some local variables may be declared. This augmented displacement is denoted in the figure by a spike in the line representing space allocation for the procedure code. The specifications of the procedure include `M_D_Exp`, an expression that limits the supplementary storage a procedure may use. The procedure must stay within that limit in order to be considered correct in terms of performance. As the picture shows, the `M_D_Exp` is an expression about only local variables and whatever parameters are passed in. These are the only variables under the control of the procedure and they are the only ones the procedure should need to consider for specification and verification purposes.

Disp is an operator that extracts the amount of storage for a given variable. This operator gets its value in the displacement clause given in an implementation of an object-oriented concept, and it is usually parameterized by the object's value [18]. At the

point where the call `P(a)` is made the picture shows **Disp**(a), to denote that `a`'s space allotment is part of the current augmentation displacement. Upon completion of the procedure call, the new value of `a`, shown as `?a` may be different and may require a different amount of space from what its value needed at the time of the call. **Disp**(?a) is part of the current augmentation at the point of completion. `Fut_Max_Sup_Exp`, as noted before, describes a bound on the storage used by the remaining code, i.e., code following the current statement under consideration.

Given his explanation, the procedure call rule follows:

C ∪ {P_Heading} \ Code; **Confirm** P_Usg_Exp[x⇝a] ∧
∀ ?a: **M_Exp**(T), **if** P_Rslt_Exp[#x⇝a, x⇝?a] **then**
Outcome_Exp[a⇝?a] ∧
Cum_Dur + Dur_Exp[#x⇝a, x⇝?a] +
Sqnt_Dur_Exp[a⇝?a] ≤ Dur_Bd_Exp[a⇝?a] ∧
Max(**Prior_Max_Aug**, **Cur_Aug** ,
Max(M_D_Exp[#x⇝a, x⇝?a],
Disp(?a) + Fut_Sup_Disp_Exp[a⇝?a]) – **Disp**(a))
≤ Aug _Bd_Exp[a⇝?a];

C ∪ {P_Heading} \ Code; P(a); **Confirm** Outcome_Exp ∧
Cum_Dur + Sqnt_Dur_Exp ≤ Dur_Bd_Exp ∧
Max(**Prior_Max_Aug**, **Cur_Aug** + Fut_Sup_Disp_Exp) ≤
Aug_Bd_Exp;

The heading for P is placed in the context, making available the specifications needed to carry out any proof. In the conclusion line, a call to `P` with parameter `a` is made at the point in the program following `Code`.

In modular reasoning, verification of this code that calls an operation P is based only on the specification of P. The functional behavior is addressed in the top line of the hypothesis part of the rule. To facilitate modular verification, at the point in the code where the call to `P` is made with parameter `a`, it is necessary to check that the **requires** clause, `P_Usg_Exp` with `a` replacing `x` holds. The second hypothesis, also about functional behavior, checks to see that if the procedure successfully completes, i.e., the **ensures** clause is met with the appropriate substitution of variables, then the assertion `Outcome_Exp` holds, again with the appropriate substitution of variables. These substitutions make it possible for the rules to talk about two distinct times, one at the point where a call to the procedure is made and one at the point of completion. The substitution of what variables need to appear at what points in the proof process avoids the need ever to introduce more than two points in the time line, thereby simplifying the process.

It is important to note here that the specification of Operation P may be relational, i.e., alternative outputs may result for the same input. Regardless of what value results for parameters after a call to P, the calling code must satisfy its obligations. This is the reason for the universal quantification of variable `?a` in the rule.

The next hypothesis in the rule is about timing, and it checks, after variable substitution, that any result from the procedure will lead to satisfaction of specified time bounds for the client program. It is not surprising that any reasoning about time or space must be made in terms of the variables being manipulated, since their size and representation affect both.

Finally, the displacement hypothesis considers the maximum over several values. To understand this hypothesis, the picture helps by illustrating the prior maximum augmentation, current augmentation both at the point of the call and at the point of the return. The picture also shows the displacement for actual parameter `a` at the beginning of the procedure call and the displacement of `?a` at the end.

The displacement hypothesis involves a nested max situation. We consider the inner max first. Here we are taking the maximum over two items. The first is the expression from the procedure heading that identifies how much storage the procedure will need in terms of the local variables and the parameters. The second is the sum of the amount of space required by the final value of the updated parameter referred to as `?a` and the amount of space for the rest of the program represented by `Fut_Sup_Disp_Exp`. From the second quantity we subtract the displacement of `a`, since it was accounted for in the current augmentation. Finally, we take the max over the two items and show that it remains within the overall bound.

The technique used in parameter passing naturally affects the performance behavior of a procedure call. In the rule, we have assumed a constant-time parameter passing method, such as swapping [3]. An additional degree of complication is introduced when an argument is repeated as a procedure call, because extra variables may be created to handle the situation. The present rule does not address this complexity.

4. AN EXAMPLE

In this section, we present a more comprehensive example of a generic code segment, including appropriate expressions for describing time and space. In our example, we reproduce Stack_Template concept from [18], where a detailed explanation of the notation may be found:

```
Concept Stack_Template( type Entry;
                        evaluates Max_Depth: Integer);
        uses Std_Integer_Fac, String_Theory;
    requires Max_Depth > 0;

    Type_Family Stack ⊆ Str(Entry);
        exemplar S;
        constraints |S| ≤ Max_Depth;
        initialization
            ensures S = Λ;

    Operation Push( alters E: Entry; updates S: Stack );
        requires |S| < Max_Depth;
        ensures S = ⟨#E⟩ ∘ #S;

    Operation Pop( replaces R: Entry; updates S: Stack );
        requires |S| > 0 ;
        ensures #S = ⟨R⟩ ∘ S;

    Operation Depth_of( restores S: Stack ): Integer;
        ensures Depth_of = ( |S| );

    Operation Rem_Capacity( restores S: Stack ): Integer;
        ensures Rem_Capacity = ( Max_Depth – |S| );

    Operation Clear( clears S: Stack );
end Stack_Template;
```

This specification is for a generic family of stacks whose entries are left to be supplied by clients and whose maximum depth is a parameter. It exports a family of stack types along with the typical operations on stacks. Any given stack type is modeled as a collection of strings over the given type *Entry* whose length is bounded by the *Max_Depth* parameter.

In order to promote both component reuse and the idea of multiple implementations for any given concept, our design guidelines include the recommendation that concepts should provide whatever operations are necessary to support whatever type is being exported and operations that allow a user to check whether or not a given operation should be called. In the stack example both *Push* and *Pop* must be present because those are the operations that define stack behavior. The *Depth_of* and *Rem_Capacity* enable a client to find out whether or not it is alright to Push or to Pop. These are called primary operations.

Our guidelines suggest that secondary operations, ones that can be carried out -- efficiently -- using the primary ones, should be in an enhancement. An enhancement is a component that is written for a specific concept. It can use any of the exported types and operations provided in that concept. For example, we might write an enhancement to reverse a stack. In it would be an operation whose specifications indicate that whatever stack is passed into the procedure is supposed to be reversed. Given below is the functionality specification of such an enhancement:

```
Enhancement Flipping_Capability for Stack_Template;
    Operation Flip(updates S: Stack);
        ensures S = #S^Rev;
end Flipping_Capability;
```

The advantage of writing this capability as an enhancement is that it is reusable, i.e., it will work for all Stack_Template realizations. For an example of a Stack_Template realization, a reader is referred to [18].

In our implementation, given below, we have included both the code (it is purely generic since any realization of the given stack concept may be used for the underlying stack type) and the performance specifications that deal with time and space.

```
Realization Obvious_F_C_Realiz for
Stack_Template.Flipping_Capability;
    Duration Situation Normal: ∃ C_Pu, C_Po, C_IE, C_EI, C_SIS: ℝ^>0 ∋
            C_Pu = LUB(Dur_Push[Entry×Stack]) and
            C_Po = LUB(Dur_Pop[Entry×Stack]) and
            C_IE = LUB(Dur_Is_Empty[Stack]) and
            C_EI = Dur_Entry.Initialization and
            C_SIS + Max_Depth∗C_EI = Dur_Stack.Initialization;
        Defn const C_1: ℝ^>0 = (C_IE + C_Po + C_Pu);
        Defn const C_2: ℝ^>0 = (Dur_Call(1) + C_EI + C_SIS + C_IE + C_:=);
        Defn const Cnts_Disp( S: Str(Entry) ): ℕ =
        ( ∑ Occurs_Ct(E,S) ∗ Disp(E) );
                E:Entry
        Displacement Situation Normal: ∃ D_SD, D_EID: ℕ ∋
            D_EID = Disp_Entry.Init_Val and
            ∀ S: Stack, Disp(S) = D_SD +
                    D_EID∗(Max_Depth – |S|) + Cnts_Disp( S ) and
            ∀ E: Entry, Disp(E) ≥ D_EID and
            Is_Nominal(Mnp_Disp_Pop(E, S)) and
            Is_Nominal(Mnp_Disp_Push(E, S)) and
            Is_Nominal(Mnp_Disp_Is_Empty(S));

        Procedure Flip( upd S: Stack );
            duration Normal: C_1∗|#S| + Max_Depth∗C_EI + C_2;
             manip_disp Normal: 2∗D_SD + D_EID∗(2∗Max_Depth +
                                    1 – |@S|) + Cnts_Disp( @S );
            Var Next_Entry: Entry;
            Var S_Flipped: Stack;
            While ¬ Is_Empty( S )
              updating S, S_Flipped, Next_Entry;
              maintaining #S = S_Flipped^Rev∘S and
              Entry.Is_Initial(Next_Entry);
              decreasing |S|;
              elapsed_time Normal: C_1∗|S_Flipped|;
              max_manip_space 2∗D_SD + D_EID∗(2∗Max_Depth
                                    + 1 – |#S|) + Cnts_Disp( #S );
            do
                    Pop( Next_Entry, S );
                    Push( Next_Entry, S_Flipped );
            end;
            S :=: S_Flipped;
        end Flip;
end Obvious_F_C_Realiz;
```

In writing performance specifications, there is a trade-off between generality and simplicity. Given that the space/time usage of a call to every operation could depend on the input and outputs values of its parameters at the time of the call, a general version of performance specification can be quite complex. But we can simplify the situation, if we make some reasonable

assumptions about the performance of reusable operations. While the performance specification language should be sufficiently expressive to handle all possibilities, in this paper, we present simplified performance expressions making a few assumptions. When the assumptions do not hold, the performance specifications do not apply.

There may a variety of ways in which time and space are handled, such as the straightforward allocation of space upon declaration and immediate return upon completion of a block as one method, and amortization as another. Here we use the term **Duration Situation** followed by Normal to indicate the former. A specification may also give performance behavior for more than one situation.

We provide constants that represent durations for each of the procedures that might be called, taking least upper bound when those durations might vary according to contents. For example, $\mathbf{Dur}_{\text{Push}}$ stands for the amount of time taken by a Push operation. Since that might vary depending on the particular value being pushed, the least upper bound is used to address that fact.

The way this approach allows the use of generic code is to have specifications that can be given in terms of the procedures they call. We think of initialization as a special procedure, one for each type, that is called when a variable is declared. For example, $\mathbf{Dur}_{\text{Stack.Initialization}}$ means the duration associated with the initialization of a stack. We do not know nor do we need to know what particular kind of stack will be used here, rather our specifications are completely generic, allowing the specific values to be filled in once a particular stack type has been designated.

All of the constants at the beginning of the realization are presented as convenience definitions so that the expressions written in the **duration** and **manip_disp** clauses will be shorter to read.

Just as we have identified what **duration** constants are needed for specifying the duration of the reversing procedure, we also set up definitions to make the storage (**manip_disp**) expression shorter to read. We can now see how the duration and manipulation displacement expressions associated with each procedure can be used when scaling up and using those procedures in a larger program.

In verifying the correctness of the procedure, for the loop statement, the programmer supplies the following information:

- An **updating** clause that lists variables that might be modified in the loop, allowing the verifier to assume that values of other variables in scope are invariant, i.e., not modified;
- A **maintaining** clause that postulates an invariant for the loop;
- A **decreasing** clause that serves as a progress metric to be used in showing that the loop terminates;
- An **elapsed time** clause for each situation assumption in the duration specification to denote how much time has elapsed since the beginning of the loop; and
- A **max_manip_space** clause that denotes the maximum space manipulated since the beginning of the loop in any iteration.

The proof rule for while loop (not given here) checks that each of the programmer-supplied clauses is valid and then employs them in the proof.

In this short version of the paper, we have omitted discussion of several important issues, including proof rules for loop statements as well as other constructs. We have also not explained how the system can accommodate dynamic and/or global memory management, though the framework allows for those complications. Finally, the non-trivial aspects of a framework within which to discuss the soundness and completeness of the proof system need to be presented.

5. RELATED WORK AND SUMMARY

The importance of performance considerations in component-based software engineering is well documented [7, 19, 20, 21]. Designers of languages and developers of object-based component libraries have considered alternative implementations providing performance trade-offs, including parameterization for performance [2]. While these and other advances in object-based computing continue to change the nature of programming languages, formal techniques for static performance analysis have restricted their attention to real-time and concurrency aspects [6, 10, 11, 20].

Hehner and Reddy are among the first to consider formalization of space (including dynamic allocation) [4, 13]. Reddy's work is essentially a precursor to the contents of this paper, and its focus is on performance specification. The proof system for time and (maximum) space analysis outlined in [4] is similar to the elements of our proof system given in section 2 of this paper. Both systems are intended for automation. In verification of recursive procedures and loops, for automation, we expect time remaining and maximum manipulated space clauses to be supplied by a programmer, though the need for the clauses is not made apparent in the examples in Hehner's paper. Our rules for these constructs are, therefore, different. Other differences include performance specification of generic data abstractions and specification-based modular performance reasoning. This becomes clear, for example, by observing the role of the displacement functions in the procedure call rule in Section 3.

This paper complements our earlier paper on performance specification in explaining how performance can be analyzed formally and in a modular fashion. To have an analytical method for performance prediction, i.e., to determine a priori if and when a system will fail due to space/time limits, is a basic need for predictable (software) engineering. Clearly, performance specification and analysis are complicated activities, even when compounding issues such as concurrency and compiler optimization are factored out. Bringing these results into practice will require considerable education and sophisticated tools. More importantly, current language and software design techniques that focus on functional flexibility need to be re-evaluated with attention to predictable performance.

ACKNOWLEDGMENTS

It is a pleasure to acknowledge the contributions of members of the Reusable Software Research Groups at Clemson University and The Ohio State University. We would especially like to thank Greg Kulczycki, A. L. N. Reddy, and Bruce Weide for discussions on the contents of this paper. Our thanks are also due to the referees for their suggestions for improvement.

We gratefully acknowledge financial support from the National Science Foundation under grants CCR-0081596 and CCR-0113181, and from the Defense Advanced Research Projects Agency under project number DAAH04-96-1-0419 monitored by the U.S. Army Research Office.

REFERENCES

1. Ernst, G. W., Hookway, R. J., and Ogden, W. F., "Modular Verification of Data Abstractions with Shared Realizations", *IEEE Transactions on Software Engineering 20,* 4, April 1994, 288-307.
2. *Generic Programming*, eds. M. Jazayeri, R. G. K. Loos, and D. R. Musser, LNCS 1766, Springer, 2000.
3. Harms, D.E., and Weide, B.W., "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Transactions on Software Engineering,* Vol. 17, No. 5, May 1991, pp. 424-435.
4. Hehner, E. C. R., "Formalization of Time and Space," *Formal Aspects of Computing*, Springer-Verlag, 1999, pp. 6-18.
5. Heym, W.D. *Computer Program Verification: Improvements for Human Reasoning.* Ph.D. Dissertation, Department of Computer and Information Science, The Ohio State University, Columbus, OH, 1995.
6. Hooman, J., *Specification and Compositional Verification of Real-Time Systems*, LNCS 558, Springer-Verlag, New York, 1991.
7. Jones, R., Preface, *Proceedings of the International Symposium on Memory Management, ACM SIGPLAN Notices 34,* No. 3, March 1999, pp. iv-v.
8. Leavens, G., "Modular Specification and Verification of Object-Oriented Programs", *IEEE Software,* Vol. 8, No. 4, July 1991, pp. 72-80.
9. Leino, K. R. M., *Toward Reliable Modular Programs*, Ph. D. Thesis, California Institute of Technology, 1995.
10. Liu, Y. A. and Gomez, G., "Automatic Accurate Time-Bound Analysis for High-Level Languages," *Procs. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LNCS 1474, Springer-Verlag, 1998.
11. Lynch, N. and Vaandrager, F., "Forward and backward simulations-Part II: Timing-Based Systems," *Information and Computation*, 121(2), September 1995, 214-233.
12. Muller, P. and Poetzsch-Heffter, A., "Modular Specification and Verification Techniques for Object-Oriented Software Components," in *Foundations of Component-Based Systems*, Eds. G. T. Leavens and M. Sitaraman, Cambridge University Press, 2000.
13. Reddy, A. L. N., *Formalization of Storage Considerations in Software Design,* Ph.D. Dissertation, Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV, 1999.
14. Schmidt, H. W. and Chen, J. Reasoning About Concurrent Objects. In *Proceedings of the Asia-Pacific Software Engineering Conference*, IEEE, Brisbane, Australia, 1995, 86-95.
15. Sitaraman, M., and Weide, B.W., eds. Component-based software using RESOLVE. *ACM Software Eng. Notes 19,4* (1994), 21-67.
16. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J. E., "Reasoning About Software-Component Behavior," *Procs. Sixth Int. Conf. on Software Reuse*, LNCS 1844, Springer-Verlag, 2000, 266-283.
17. Sitaraman, M., "Compositional Performance Reasoning," *Procs. Fourth ICSE Workshop on Component-Based Software Engineering: Component-Certification and System Prediction*, Toronto, CA, May 2001.
18. Sitaraman, M., Krone, J., Kulczycki, G., Ogden, W. F., and Reddy, A. L. N., "Performance Specification of Software Components," *ACM SIGSOFT Symposium on Software Reuse*, May 2001.
19. Szyperski, *C., Component Software: Beyond Object-Oriented Programming,* Addison-Wesley, 1998.
20. Special issue on Real-Time Specification and Verification, *IEEE Trans. on Software Engineeering*, September 1992.
21. Special section: Workshop on Software and Performance, Eds., A. M. K. Cheng, P. Clemens, and M. Woodside, *IEEE Trans. on Software Engineeering*, November/December 2000.

On Contract Monitoring for the Verification of Component-Based Systems

Philippe Collet
Objects and Software Components Group

Laboratoire I3S - CNRS - Université de Nice - Sophia Antipolis
Les Algorithmes- Bât. Euclide B, 2000 route des Lucioles
BP 121, F-06903 Sophia Antipolis Cedex, France

Philippe.Collet@unice.fr

ABSTRACT

This position paper focuses on contract monitoring for component interfaces, considering the verification of functional and non-functional properties in the contracts. We investigate what properties are needed on behavioral and *Quality of Service* contracts. We also define what are the requirements on a monitoring environment to handle properly those contracts. We briefly transpose those requirements to a meta-level architecture.

1. INTRODUCTION

The development of component-based systems intends to deliver the beneficial effects that the object-oriented approach failed to completely provide: reuse of out-sourced pieces of software and thus increased productivity. The definition of component devised during the 1996 Workshop on Component-Oriented Programming [1] is the following: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

In this definition, the specification process is clearly related to *contractually specified interfaces*. This position paper focuses on the general notion of contract for components, that is a component must expose functionalities, through its functional contract, and its *performances*, using some non-functional contract. More precisely, Beugnard et al [3] categorize contracts in four levels:

1. Syntactic contracts, that is signatures of data types.
2. Behavioral contracts, that is some semantic description of data types,
3. Synchronization contracts, which deal with concurrency issues.
4. Quality of Service (QoS) contracts, which encompass all non-functional requirements and guarantees.

We consider it crucial to dispose of such contracts if we want software components to behave like components from other engineering domains. Moreover, software components can certainly be the right software units to justify the additional cost of using more formal approaches, as their life cycle and their marketing strategies might be driven by quality. To create a real market of software components, application developers must be capable of browsing, comparing and choosing components [13] according to all their exposed properties: an expression of services and quality of these services is then obviously necessary. Those *specifications* must then be *verified* one way or another.

Consequently we believe that the specification and verification of component-based systems must take into account those four levels of contract from the start, to provide a broad and consistent framework to handle those different kinds of properties. As we also believe that expressive formalisms are needed to support the contractual approach, we base our work on the hypothesis that it is not possible to fully verify statically that such contracts are never violated. The runtime enforcement of those contracts together leads to specific monitoring problems as a straightforward combination of separate contracts monitoring would interfere with each other. A specific monitoring framework is needed to handle all kinds of contract and we consider that an appropriate meta-level architecture must be defined to provide such a framework.

In this paper, we investigate what properties are needed on contracts, by considering two specific levels, behavioral and QoS. We define the global contract which consists of the combination of the four levels. We describe what are the requirements on a monitoring environment to handle properly those contracts. We briefly transpose those requirements to a meta-level architecture, both in its design and in its implementation.

2. SPECIFICATION OF BEHAVIORAL AND QOS CONTRACTS

In this paper, we only focus on the behavioral and QoS contracts. The first level of contract corresponds to type signatures, and type checking is usually performed statically. The synchronization aspects of contracts still need

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2001 Philippe Collet.

to be studied in a general enough component framework, as concurrency issues are often reduced to the means of communication of a given connector of a component.

2.1 Behavioral Contract

Several specification techniques can be envisaged to specify the behavioral semantics of component interfaces. Currently the contractual approach based on preconditions, postconditions and invariants is promoted as a pragmatic but valuable specification technique for components. Current component frameworks are based on, or at least promote, OO programming languages. The majority of those OO languages provide only the first level of contract, but there are languages, such as Eiffel [12], that inherently incorporate behavioral contracts with preconditions, postconditions and invariants. Many extensions to existing languages with behavioral contracts, such as iContract [10] or JML [11] for Java, have also been designed.

Szyperski [14] showed that behavioral contracts based on pre and postconditions have several drawbacks related to call-backs and re-entrance. Nevertheless, we believe it still constitutes the best trade-off between expressiveness and ease of comprehension by an average developer.

However, there are two different specification approaches for those behavioral contracts:

- The first one can be qualified as *language-based specification*, as the contract expression is a boolean expression of the annotated language, usually using all the functional features (access to fields, method calls, basic types and operations) with some specific operators to refer to previous states (`old, @pre`), to the result of the annotated function (`result`), etc. The main reference is the Eiffel language [12], even if more expressive assertion language have been proposed [10], with the addition of quantification operators for example.

  This approach suffers from its reuse of the annotated language, as it is often hard to provide a complete formal semantics for the assertion language, as the underlying programming language does not provide one either. Despite this problem, this approach is open to partial specification, fits well with subtyping and is well understood by developers.

  Moreover, recent work [7] provides a sound framework for behavioral contracts based on pre and postconditions. This theory [6] brings soundness regarding inheritance and interfaces in the context of the Java programming language. But this theory is applicable to many other cases.

- The other approach relies on model-based specifications [4], that is the annotations that make up the contract are stated in terms of a mathematical model of the states of objects. This kind of specification usually enables static analysis and theorem proving, as they are based on an algebraic *style*. However, as the expressiveness of the language increases, proofs are no longer possible. Moreover, this kind of specification is hard to understand by developers as they are not trained or used to think in such a way.

  A somewhat hybrid approach is now developed : JML (Java Modeling Language) [11] restricts the use of models to model fields and reuses as much as possible the Java syntax and semantics for basic operations in the assertion language (access to fields, method calls, basic operators, etc.). JML also provides very interesting features with the separate specification of normal and exceptional behaviors, quantification operators and refinement of the specification models[1]. The runtime enforcement of some parts of the contract is also possible. As a result, JML provides a well-founded basis from the start, trying to be closer to developers.

Both approaches have advantages and drawbacks, and are trying to eliminate their respective disadvantages: theoretical work is done in the language-based approach, practical issues motivate work in the model-based approach. One can hope that the approaches will merge or that one approach will reuse and adapt all beneficial aspects of the other one.

In the meantime, it must be noted that both JML and language-based contracts systems lead to the same kind of runtime monitoring systems, which will be shown to interfere with other contract levels.

2.2 QoS Contract

Regarding QoS, contracts have been investigated in the world of distributed objects and components systems. Software components, in the broad sense, must be able to expose many different non functional properties, such as performance, reliability, policies related to persistence, transactions or security. Apart from the properties that are usually provided by the container — or context —, designing a QoS contract system expressing time and space performance in function of some resources usage seems quite challenging, as many aspects must be taken into account.

The complexity of algorithms can be easily related to an order using the O notation on both the average and worst cases. Both cases are likely to be relevant for a software component. But this notation expresses complexity bounds, independent of any deployment platform, so these formulas need to be related to absolute bounds [14], showing some real figures. In addition to the issue of comparing performance, contracting QoS leads to the problem of handling negotiation and renegotiation.

To our knowledge, no QoS contract language or system expresses and verifies performance issues based on input parameters and resources usage, but QML (QoS Modeling Language) [8] looks like the most advanced QoS specification language. In QML, the QoS specification is made of three mechanisms: contract type, contract and profile. Contract type are QoS aspects, such as performance or reliability. A contract is an instance of a contract type and a profile associates a QML contract with an interface. The QoS aspects that can be represented in QML are quite powerful, with different domains of value, constraints and even statistics on measured values over a period of time. However QML does not provide any means to express a QoS contract according to some parameters that would come from the component interface, e.g. to specify a time constraint in relation with the size of an input data structure. Moreover resources consumption cannot be specified. QML contracts are made of constraints on domains of values, and a contract can *refine*

[1] JML also provides a `when` clause, which can be seen as part of a level 3 contract on synchronization: if a method is called and its preconditions hold, the call will wait until the `when` clause holds as well.

another one by adding constraints or putting stronger constraints on an already constrained domain. Each kind of constraint that can be defined in QML must specify a total order among its values. A conformance relation is then defined between the contracts.

Monitoring QoS is not considered in QML [8], but similar QoS oriented approaches monitor some properties at runtime by configuring the middleware, or by using meta-level mechanisms [2]. As some categories of QoS can involve pervasive monitoring, like security in Java [5], interferences between the separate QoS monitoring already proposed would certainly occur. As general-purpose QoS specification formalisms are likely to be proposed, a contract monitoring environment must be carefully designed to enable to express the correct combination of each corresponding monitoring process. It must be also kept open enough to take into account the possible new features. The environment must also handle the case of partial conformance between QoS contracts or during monitoring, e.g. a time constraint is respected but a space constraint is not. Different policies are then applicable: termination, renegotiation, etc. The same problem arises on the global contract, as described in the next section.

2.3 Putting Contracts Together

Considering all four levels together (type, behavior, synchronization, QoS), a proper combination can be determined in order to provide a global contract. The general specification can simply be done separately in each contract formalism and the conformance rule of this global contract is the conjunction of all conformance rules. However, it is also important to consider the case where some partial conformance is achieved, which typically leads to contract renegotiation in QoS-aware systems. Different actions regarding the contract can be started:

- Termination if the QoS contract is considered as too important to be renegotiated.
- Renegotiation of the QoS contract with weaker constraints (e.g. a 3D component cannot provide a 30 frames/s rate and the new QoS contract asks for 25).
- Withdrawal of the QoS contract, getting back to a purely functional *best-effort* approach.
- Renegotiation of the functional contract and possibly of the QoS contract (e.g. the same 3D component is asked to lower its resolution and may be asked to maintain the 30 frames/s rate).

Consequently the combination of all contracts must be provided with the addition of dynamic negotiation capabilities, which can be taken for example from the QoS formalism.

2.4 Monitoring Issues

Monitoring at runtime needs a proper support so that a specific contract monitoring does not affect another monitoring process at a different level. For example, behavioral and QoS monitoring can interfere if the monitoring code that evaluates assertions create new objects when the QoS is monitoring space occupancy. In the same way, the time spent in monitoring must not be taken into account in profiling time, unless explicitly specified. Consequently, monitoring behavioral contracts must be done in a framework that will not interfere on any other contract level:

- by not adding new types in the type hierarchy;
- by not modifying the program behavior in relation to synchronization;
- and finally by not consuming any time or space!

Even if the first property can be achieved by modifying all the methods that give access to type information, it is not feasible to completely achieve the second and third properties. Nonetheless, the monitoring environment must strive for minimizing the effect of the observer on the observed phenomena.

3. REQUIREMENTS FOR MONITORING CONTRACTS

In order to provide an appropriate framework to monitor all kinds of contract, we propose exposing the necessary concepts that are manipulated by behavioral contract systems. In the same way, we expect to describe an open enough framework for QoS monitoring, so that the monitoring processes can be manipulated and composed at the global level.

3.1 Behavioral Contract Monitoring

The monitoring technique for such contracts consists in checking the appropriate preconditions at the entry of a method, the postconditions and invariants at the exit. Defining what are the appropriate assertions, i.e. the semantically correct ones, to be monitored on a given object at runtime, according to inheritance, subtyping and implemented interfaces [6] is considered as out of the scope of the monitoring process. We present a list of requirements on the monitoring system:

- The integration of the contract enforcement code with the normal code must not create new *visible* classes or methods — wrapping asserted methods is a common way to integrate assertions —. Even if programmers can be told not to use these, any tool that uses the modified class will consider them as normal unless correctly hidden or specified. Avoiding a pervasive integration is also important for the deployment footprint, which could be constrained in a QoS contract.
- Specific data structures and code are usually necessary to manage the integration, to avoid non-termination of assertion checking due to recursion, to provide assertion triggering at a fine-grained level (class or object) and to make the checker thread-safe!
- All accesses to instance fields and all method calls that are made to evaluate an assertion are recorded as such, i.e. not counted in time measurement.
- All created objects during any evaluation are excluded from space measurement.
- The synchronization policies and behaviors normally defined for the component should not be modified. How this can be achieved, totally or partially, remains an open question. However, the sketched framework is expected to be able to design and experiment proper solutions.

The assertion languages always provide enhancements to boolean expressions in order to increase expressiveness. The most common ones are studied in relation to the monitoring issues:

- Quantification operations (∀, ∃), or more generally higher-level functions, need to be translated to the underlying language, thus generating extra code and new functions. That boils down to the first side-effects presented above.
- Access to the result of the annotated function usually generates side-effects because of methods wrapping or any other techniques used to provide this feature.
- Reference to the previous state of objects in the specification of procedures (`old`, `@pre`) is usually done by generating local variables that keep references or values of the concerned variables by computation at the method entry. They are later referenced at method exit. This additional code generates side-effect. This is also the same for the `let` construct, which is used to avoid repetitions in assertions.

All the prototyped approaches that have been proposed so far generates all, or almost all, side-effects listed above. That includes approaches based on source to source processing, bytecode adaptation, compile-time or runtime reflection and aspect-based processing.

3.2 QoS Contract Monitoring

As a proper general QoS specification language is not available, we infer some principles on how to monitor QoS. Taking QML as an example, the monitoring would be based on time measurement and appropriate recording to compute necessary values and statistics. QoS in general would be based on measurement of many parameters, to compute results ranging from simple constraints to complex function of several parameters, such as time and space requirements together with dependencies on available resources, size of input data, etc.

Consequently, the computation of these results needs to be semantically correct, that is:

- *without any interference* from other contracts, and actually, without any interference coming from the component surrounding, such as the services provided by the container.
- *at the appropriate times.* Considering the method call as the essential point of contracting, the monitoring environment can be kept open by reusing the fine-grained model of aspect-oriented programming [9] to consider the following events: before the method call (client side), at the method entry (provider side), before the exit (provider side), just after the method call (client side). A distinction can also be made between the method call and the effective method execution (late binding).

3.3 Requirements for an appropriate meta-level

Monitoring behavioral contracts can be seen as an aspect in the sense of aspect-oriented programming [9]. Monitoring some QoS properties in middleware has been done through message reflection [2]. Consequently we consider that the monitoring of all forms of contracts must be done in a appropriate meta-level framework supporting message interception, as contracts are mainly monitored on method calls. However this meta-level must satisfy strong constraints, as it must provide a clear separation between the normal behavior and other aspects, so that monitoring of contracts can be as **transparent** as possible to the semantics and to QoS for the client.

We believe that this transparency can be achieved by defining a minimal set of interactions between the two levels, taking into account low-level issues such as object allocation. At the meta level, the careful implementation of the specific contractual constructs we have described is expected to enforce transparency as well. The main issue of such a framework will certainly be performance, as both levels will need to act as almost separate execution environments. The implementation of a prototype to experiment these ideas has begun. It uses a language-based specification language for Java, OCL-J, which adapts the Object Constraint Language of UML to the Java programming language. The developed prototype is intended to use the Java Platform Debugging Architecture (JPDA), as this architecture provides a framework that is close in some points to our requirements. We expect these experiments to provide feedback on how to properly design the interactions between the base and the meta levels, as well as new insights in the area of contract monitoring.

4. CONCLUSION

In order to provide quality software components, the specification and verification of component-based systems must take into account both the functional and non-functional contracting of interfaces. By considering a global contract merging all kinds of contracts, we showed that renegotiation of QoS contracts must be supported and that different monitoring codes must be aware of each other and must not interfere. Consequently we argue that contract monitoring must be handled globally inside a meta-level that clearly separates the base level and the meta-level in all functional and non-functional aspects.

5. ACKNOWLEDGMENTS

Thanks to Jacques Malenfant for discussions on non-functional contracts and for pinpointing the QoS specification language QML.

6. REFERENCES

[1] *International Workshop on Component-Oriented Programming (WCOP'96)*, 1998.

[2] M. Aksit, A. Noutash, M. van Sinderen, and L. Bergmans. QoS Provisioning in Corba by Introducing a Reflective Aspect-Oriented Transport Layer. In *1st ECOOP Workshop on Quality of Service in Distributed Object Systems (QoSDOS 2000)*, 2000.

[3] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *Computer*, 32(7), July 1999.

[4] Y. Cheon and G. T. Leavens. A Quick Overview of Larch/C++. *Journal of Object Oriented Programming*, 7(8):39–49, Oct. 1994.

[5] U. Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2000.

[6] R. B. Findler and M. Felleisen. Contract Soundness for Object-Oriented Languages. In *Proceedings of OOPSLA'2001*, 2001.

[7] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral Contracts and Behavioral Subtyping. In *Proceedings of Foundations of Software Engineering (FSE'2001)*, 2001.

[8] S. Frolund and J. Koistinen. Quality of Service Specification in Distributed Object Systems Design. *Distributed System Engineering*, December 1998.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'2001)*, Lecture Notes in Computer Science. Springer Verlag (Berlin), 2001.

[10] R. Kramer. iContract - the Java Design by Contract Tool. In M. Singh, B. Meyer, J. Gil, and R. Mitchell, editors, *International Conference on Technology of Object-Oriented Languages and Systems (Tools 26, USA'98)*, IEEE Computer Society Press (New York), 1998.

[11] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A Notation for Detailed Design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.

[12] B. Meyer. *Object-Oriented Software Construction*. The O-O series. Prentice Hall Inc. (Englewood Cliffs, NJ), 2nd edition, 1997.

[13] H. L. Nielsen and R. Elmstrom. Proposal for Tools Supporting Component Based Programming. In *Fourth International Workshop on Component-Oriented Programming (WCOP'99)*, 1999.

[14] C. Szyperski. *Component Software — Beyond Object-Oriented Programming*. Addison-Wesley Publishing Co. (Reading, MA), 1998.

A Framework for Formal Component-Based Software Architecting

M.R.V. Chaudron, E.M. Eskenazi, A.V. Fioukov, D.K. Hammer

Department of Mathematics and Computing Science, Technische Universiteit Eindhoven,

Postbox 513, 5600 MB Eindhoven, The Netherlands

+31 (0)40 – 247 4416

{m.r.v.chaudron, e.m.eskenazi, a.v.fioukov, d.k.hammer}@tue.nl

ABSTRACT

The assessment of quality attributes of software-intensive systems is a challenging issue. This paper outlines a method aimed at quantitative evaluation of certain quality attributes that are important for embedded systems: timeliness, performance and memory consumption.

The paper sketches out the key principles for building a formal model for evaluating quality attributes: (1) Dependability constraints are specified in an end-to-end fashion; (2) Components are attributed with resource demands; (3) Specification of component interaction is separated from specification of component behavior.

The method is aimed to be applicable in practice. Therefore we investigate combining widely used software modeling notations with existing formal methods. In particular, the proposed approach combines Message Sequence Charts and Timed Automata. We illustrate the approach with an example.

Categories and Subject Descriptors

D.2.11 Software Architectures; D.2.4 Software / Program Verification

General Terms

Performance, Design, Reliability, Verification.

Keywords

Component-based software, software architecture, quality attributes, architecture evaluation, timeliness, memory consumption, formal methods.

1. INTRODUCTION

Nowadays, more and more “intelligent” devices contain sophisticated embedded software to fulfill a broad scope of functions. As more devices are developed, the scope of functions to be implemented must be broader. This growing complexity complicates the development of embedded software. Thus, new approaches for software development are heavily needed.

To reduce development cost and development time, the reuse of existing solutions is vital. For this purpose, the construction of software with reusable components is highly desirable. This method of software development requires techniques for assessing the composability of components. This assessment can be done reasonably well for the functional aspects of components, but no adequate techniques exists for analyzing composability of non-functional ones.

The possibility to estimate relevant quality attributes in the early development phase is crucial. These kinds of predictions will reduce the risk of developing non-competitive, infeasible or flawed products. Thus, for quantitative software architecting a formal mathematical basis is needed.

Most contemporary architecting approaches deal only with the functional aspects of software. But, there are also non-functional quality attributes, a typical example being dependability quality attributes: performance, timeliness, reliability, availability, safety, and security. These attributes emerge as a result of collaborative functioning of all parts, which a system is built from, and they can make a significant impact on the entire architecture. Consequently, one of the most important issues in software architecting is dealing with quality attributes.

Many modern techniques for software evaluation use expert-based approaches (e.g. ATAM [4], SAAM [1]) which evaluate overall quality of the architecture. The quantitative methods used in these approaches are applied in an ad-hoc manner and, in many cases, are too context-oriented to be generalized. As a consequence, it is difficult to make the architecting process *reliable*, *predictable*, *and repeatable*.

For the time being, architecting is still more an art than an engineering discipline. Substantial efforts have to be invested in making the architecting process more *rationalized and precise*. One of the ways to make the architecting process more precise is the use of formal methods. However, formal methods are not a silver bullet; they have their own drawbacks. One should be aware that the application of formal methods usually requires precise and complete specification. In many cases, an architect does not have this information. He or she may not even be interested in a very detailed design at the early stages of design, but prefer to postpone design decisions to later stages. Thus, a balance between the architect's freedom of design and the precision of analysis and specification should be found. Note, accuracy of a specification is not always a drawback, as it also stimulates one to think more accurately and systematically.

Support by automatic tools could be very helpful for architects. However, tooling usually requires strict non-ambiguous semantics of the models being processed; thus, formal methods are also essential here.

There exist no general approaches for evaluating non-functional properties of a system at the architectural level. The interesting approach combining structural description of architecture (Darwin Architecture Description Language) with behavioral description of components (Labeled Transition System) was proposed in [8]. However, this approach only allows checking safety and liveness properties, but it does not model timing aspects which are needed for the analysis of quality attributes like timeliness and performance. Also, the existing methods for quantitative evaluation focus on one quality attribute only.

We aim to integrally model multiple quality attributes–*timeliness*, *performance*, and *memory consumption*–to support the making of architectural design trade-offs.

1.1 Requirements on the Method

The aim is to develop a method for supporting software architects during early stages of component-based software architecting of resource-constrained systems. It is necessary to find suitable techniques for architecture description and methods for quality attributes evaluation and to merge them into an integrated framework.

The requirements on the method are the following:

1. Compositionality of resource-constraint systems. The methodology should be *compositional*. This means that quality attributes of a composite system can be calculated from the constituents components and composition mechanisms. In particular, the method should focus on predicting quality attributes based on resource consumption of the components.

Even if the functional interfaces and the interaction mechanisms of components are precisely specified, component composition may not function properly because non-functional properties of the entire system were not considered beforehand. Typical instances of this problem are resource conflicts (e.g., race conditions, conflicts on access to shared resources etc). These conflicts make it difficult to ensure the proper and predictable behavior of a system in advance.

2. Dependability assessment during the early development phases. The dependability attributes (timeliness, performance, reliability, availability, safety and security) of component-based systems cannot be evaluated by the current approaches. So, the method must help architects to estimate the dependability attributes and ensure a certain level of *estimation accuracy*. As a simple example, the results can be described in terms of worst-case and best-case estimations.

Also, there are non-technical requirements on the method. In order to be easily comprehensible and easy to learn by software engineers, the method should be based on widely accepted software specification and design techniques. These techniques should make engineer's work more efficient after the engineer has gained some experience with them.

2. KEY PRINCIPLES FOR COMPONENT-BASED ARCHITECTING

Component-based architecting (e.g., see [15] and [21]) is one of the most promising approaches for managing complexity and boosting reuse. However, current component-based approaches do not address the behavioral and non-functional aspects of software. Therefore, we propose the following extensions.

Explicit specification of component behavior and interaction. A formal specification of the dynamic aspects of components and their interaction is necessary for reasoning about the behavior of their composition and its non-functional properties.

Support of hierarchical component description. A component can be either atomic or compound. The atomic component cannot be further subdivided, but the compound component can consist of atomic and(or) other compound components. As a result, one has more flexibility in choosing the unit of the reuse: either a single atomic component or an entire package. Another advantage is the possibility to apply compositional design approach at the different levels of hierarchy: a system is composed from subsystems; subsystems are composed from compound components etc.

Separation of component interaction from component behavior. The description of behavioral aspects is structured in separate parts. There are specifications of *component behavior* and specifications of *component interaction*. A rationale for independent specification of the interaction relationships is presented in [2]. We identified the following additional reasons for this separation:

1. Genericity/Tailorability: The interaction specification may be used to tailor the behavior of generic components to particular context. This helps to avoid coding of context-driven aspects within components and, hence, allows more general component designs.
2. Specifying constraints end-to-end: Dependability constraints are often concerned with the end-to-end interaction between components. Having a separate specification of the interaction constitutes a better means for structuring the specification than the alternatives: (1) placing constraints at one of the components involved or (2) dividing up an end-to-end timing constraint over multiple components.
3. Loose coupling: In existing component models the way that a component is intended to interact with other components is programmed into a component (endogenous binding). This has to change if the behavior of other components changes. Hence, it constitutes a dependency on the behavior of other components. By specifying interaction separately (exogenously), this dependency is avoided.

Explicit specification of the resource requirements of components. The dependability of a system is related to the amount of resources consumed by the components and provided by the execution platform. There are three types of resources: computation resources, communication resources and storage. The definition of component resource requirements in a platform-independent way broadens the scope of component application.

Specification of dependability constraints in an end-to-end fashion. The basic idea is that timing and dependability

constraints should not be component attributes, because this would jeopardize reusability. They are rather considered as constraints on the dynamics of the system, i.e. on the component interaction.

Distinguish resource constraints and resource consumption. The former are described at the overall system level in an end-to-end fashion, and the latter is associated with a component description. This separation enables the development of reusable components and gives designers freedom in the satisfaction of the resource constraints.

All the aforementioned aspects have to be properly elaborated in order to constitute a practical method.

3. ARCHITECTING ENVIRONMENT

To reconcile the goals of using accepted software engineering notations and automated analysis tools, we aim for a framework that consists of three parts (see Figure 1).

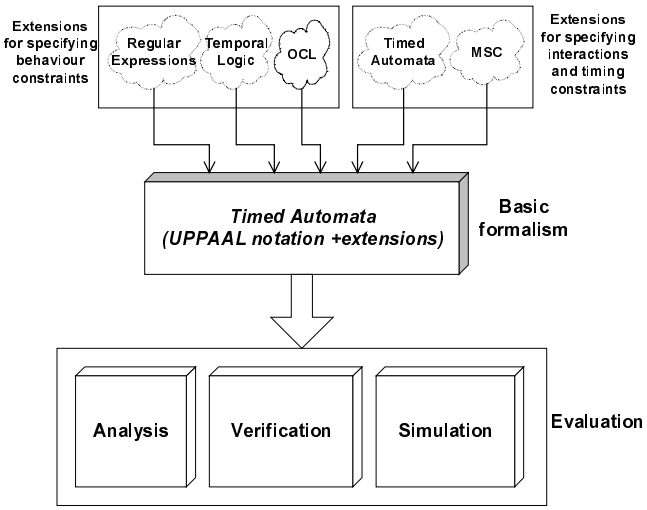


Figure 1. Architecting environment

The architecting environment provides engineers the possibility to use a combination of notations for describing architectures. For the time being, we focus on Message Sequence Charts (MSC) [20] and Timed Automata (TA) [3]. Also, the architecting environment is to be extendible with other notations, such as the Object Constraint Language (OCL) [24] or temporal logic.

For analyzing architectural designs, the different notations need to be related. To this end, we devise mappings of notations onto a *basic formalism*. In our approach, Timed Automata are used as basic formalism. We do not expect to find mappings of all constructs of all modeling notations onto a single basic modeling formalism. In co-operation with engineers, we have to select subsets of the notations that comprise the most important modeling constructs, yet also provide the information needed for automatic analysis.

To support different types of analyses we envisage a *collection of analysis tools* such as schedulability-, simulation- or verification-tools. These tools operate on the representation of the architectural design in terms of the basic formalism.

4. ARCHITECTURE DESCRIPTION TECHNIQUES

This section enumerates the requirements on architecture description techniques and outlines the framework. Also, it contains an example to illustrate the method proposed.

4.1 Requirements for description techniques

For specifying component behavior, a number of formal description techniques were inspected and compared. Before the actual comparison, essential requirements on the description techniques were identified. These requirements and their rationale follow below. The requirements are marked as *compulsory (C)* or *optional (O)*.

1. The description techniques should support quantitative models for timeliness analysis. *(C)*

   Rationale: to enable timeliness assessment at the early architecting phase, before system implementation.

2. The description techniques should support quantitative evaluation of memory consumption. *(C)*

   Rationale: to enable memory consumption estimation at the early architecting phase.

3. The description techniques should support specification of timing constraints in an end-to-end fashion. *(C)*

   Rationale: to avoid unnecessary reduction of design space (caused by artificial subdivision of the initial deadlines).

4. The description techniques should support the possibility to specify interaction behavior exogenously *(C)*.

   Rationale: to increase reusability of the components and to build flexible architectures.

5. The description techniques should allow one to reason about the properties of a component composition, based on the properties of the components. *(C)*

   Rationale: To enable effective (automated) formal reasoning.

6. The description techniques should support the specification of resource requirements (processing, storage and communication. *(C)*

   Rationale: to enable analysis of effects of resource conflicts.

7. The specifications must be comprehensible for engineers *(C)*.

   Rationale: reduce efforts for education of engineers.

8. Support for automatic code generation *(O)*.

   A description technique should enable creating of tools that can generate code for a given specification.

   Rationale: to enable efficient development.

9. Use of existing design, simulation and verification tools *(O)*. It is preferable to use the existing tools instead of developing new ones.

   Rationale: to design the software quickly and easily.

4.2 Basic formal framework

In this section we explain the architecting approach by listing the models that should be constructed for describing an architecture. We motivate the choices of the formal description techniques for these models.

4.2.1 General view

A general overview of the approach to modeling architectures is given in Figure 2. Three essential architecting models are considered.

The *"Structural model"* represents the static configuration of a system through the dependencies and connections between components.

The *"Behavioral Model"* is used to describe the dynamic aspects of the components, component interaction and resource constraints (e.g. end-to-end deadlines). A component description specifies resource requirements in terms of the "Resource Model".

The *"Resource model"* describes the available resources. This model also defines a sharing strategy for each resource.

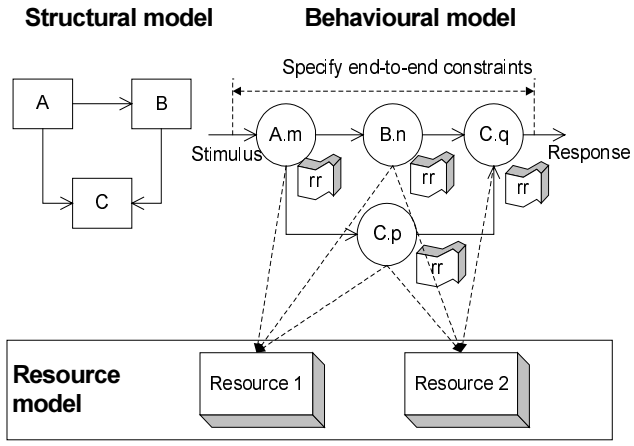


Figure 2. Overview of the approach

4.2.2 Choice of appropriate formalisms

For structural description we consider one of the existing component models supporting the notions of provided and required interfaces (e.g. Koala [19] and Darwin [18]). Since we aim to the evaluation of timing properties, the proper behavioral description formalisms are to be found.

After comparing several formalisms, the extended notion of State Machines (Timed Automata, [3]) was chosen for the specification of component behavior. Basic Message Sequence Charts [20] were chosen for specifying component interaction. This section motivates this choice.

4.2.2.1 Timed Automata

Timed Automata are supported by a wide scope of existing tools for modeling and simulation (e.g. UPPAAL, for details see [6]). Furthermore, their graphical description makes them comprehensible for engineers.

The theory on Timed Automata describes constructions for obtaining an automaton that describes the behavior of the parallel composition of timed automata. For the simplest cases, it is possible to use the Cartesian product.

However, when using timed automata, certain principles must be followed in order to be able to reason compositionally. For example, one should not use global clock variables to define constrains on the behavior of multiple components.

4.2.2.2 Message Sequence Charts

As mentioned before, component interaction is specified separately from component behavior. For that, a specification language is required that can address the following issues:

- It should enable restricting the behavior of generic components
- It should support the specification of timing constraints in an end-to-end fashion.

The MSC notation allows one to vividly express timing constraints between stimulus and response events at a single place in a specification. This is in contrast to timed automata, where timing constraints are specified by means of two (or more) constraints on shared clock variables that are distributed over separate states or transitions of the model (this will be illustrated later by an example). This reduces the intelligibility of a specification.

MSC are a well-accepted software notation that is easy to learn and understand. Also, they have formal semantics in terms of automata (see e.g. [14]) that makes it possible to relate them to timed automata (which we use as basic formalism).

Because of the above advantage, MSC were preferred to Timed Automata for specifying the timing constraints.

4.3 Example

To give a flavor of our approach, we will demonstrate some of the description principles (analysis is not included) with an example of an "Automatic Teller Machine".

The structural model of the architecture is depicted in Figure 3.

Figure 3. Key components

The system consists of the following components: User, ATM (modeling a cash dispenser), and Bank (modeling some aspects of bank operation). User and Bank interact only with ATM, but not with each other.

The behavioral model consists of a specification of the behavior of the individual components using UPPAAL and a specification of their interaction using MSCs.

We briefly explain the UPPAAL notation [6]. In UPPAAL, time is modeled using clock variables: timing constrains are expressions over clock variables. These constraints can be attached both to transitions and states. A condition on a state is an *invariant*; the system is allowed to be in a state only if its invariant holds. A condition on a transition is a *guard*; the transition can only be taken if the guard holds.

The labels on transitions denote events. Labels with a question mark "?" define input events; labels with an exclamation mark "!" define output events.

Returning to the example, Figure 4 describes the behavior of User.

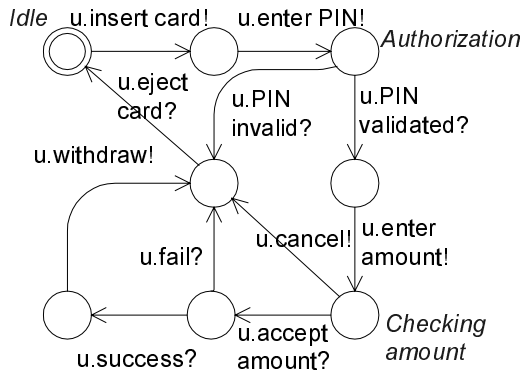


Figure 4. Behavior of User

The specification of User describes the behavior of an individual who wants to withdraw cash from an ATM. The automaton defines an order on the events for the withdrawal process.

The behavior of ATM and Bank is illustrated in Figure 5 and Figure 6, respectively.

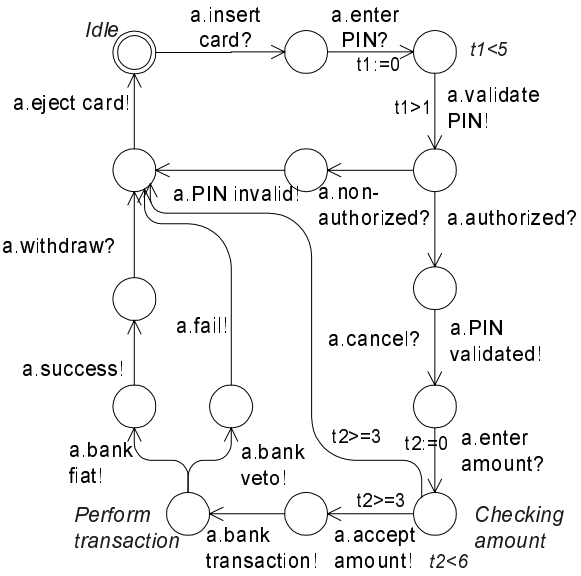


Figure 5. Behavior of ATM

For the specification of Bank we use two automata. The semantics of this is that they operate in parallel. This allows further enhancing of the specifications to support more than one User and one ATM: the unnecessary serialization of the authorization and transaction requests from different cash dispensers, which would be enforced by modeling the behavior as a single automata, is avoided.

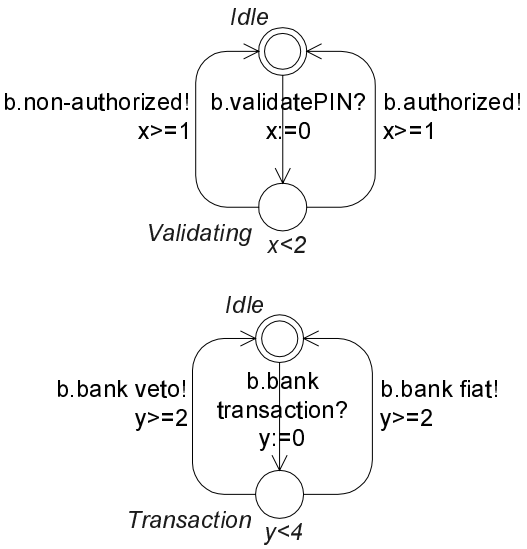


Figure 6. Behavior of Bank

We use clock variables x and y to specify the resource consumption of Bank. The specification states that the time of processing authorization requests (by a hypothetical CPU) is at least one time unit and at most two time units. The former is indicated with the guards ($x>=1$) on both transitions, and the latter is specified with the invariant ($x<2$) of the *"Validating"* state. Likewise, the time necessary for performing a transaction is at least two time units but at most four time units. Similarly, one can specify consumption of CPU capacity for other components.

Finally, we demonstrate the specification of end-to-end timing constraints and the interaction between the components User, ATM, and Bank with the Message Sequence Chart in Figure 7.

On the one hand, the message sequence chart in Figure 7 specifies through which transitions all the three automata interact. For example, to indicate that the actions "u.insert card" and "a.insert card" of User and ATM, respectively, need to synchronize, we use an ampersand symbol "&". Likewise, the other labels of all the three automata are bound. This type of specification technique allows one to bind components in an exogenous manner.

Additionally, this message sequence chart indicates that the time between the occurrence of "enter PIN" and the occurrence of "PIN validated" must not exceed five time units. At the same time, it indicates that time between "enter amount" and "transaction success" must not exceed six time units. In a similar way, message sequence charts can be used to specify other dependability constraints in an end-to-end manner for relevant execution scenarios.

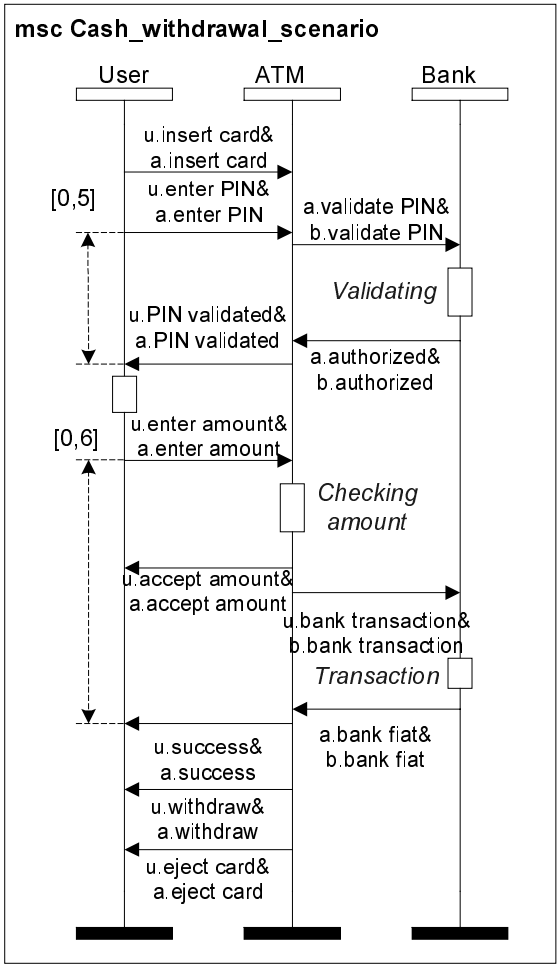


Figure 7. Specification of deadlines and component interaction

5. TECHNIQUES FOR EVALUATION OF QUALITY ATTRIBUTES

This section summarizes our evaluation of methods for the assessment of timeliness and memory consumption and interprets their use for the software architecting.

5.1 Timeliness evaluation

As already mentioned, timeliness is an important quality attribute. Timeliness can be reasoned about either (analytically) through a schedulability test, a typical example being Rate Monotonic Analysis (RMA) (see [17], [16], [11], and [12]), or through the construction of an explicit schedule (e.g. in [23]).

Both approaches model the scheduling policy adopted for the system. Depending on whether the priority assignment strategy of the scheduling policy is fixed or dynamic, different models have to be used.

For the fixed-priority scheduling policy, RMA is usually applied. This method is based on the analysis of a so-called critical instant, when all tasks in the task set are released simultaneously. It is proven that the worst-case response time appears for each task during the critical instant. The RMA method calculates response time for each task (for a given real-time situation describing a set of tasks being analyzed [12]), based on the specified worst-case execution time and deadline. In addition, if the tasks share some resources, the blocking time induced by the ones with lower priority should be given. Finally, the periodicity information (for a simple case, in a form of task inter-arrival periods) has to be provided to enable the application of RMA.

However, RMA can only be used with some, rather strict, assumptions on the tasks of the system. The main restriction is the assumption that the arrival pattern of a stimulus has some form of *periodicity*. Early versions of RMA have dealt only with strictly periodic events; however, later extensions have incorporated a-periodic tasks and sporadic servers. Another drawback of RMA is that it does not allow *interaction* between the tasks. The only allowed interaction is mutual access to shared resources.

New schedulability modeling approaches have emerged as a result of the substantial progress in the development of model checking techniques both for ordinary and hybrid timed automata. These approaches partially address the drawbacks of RMA-like techniques, as they take interaction between the tasks into account.

The main principle on which these techniques are built is the replacement of the initial schedulability problem with the reachability problem for a timed automaton encompassing all peculiarities of a concrete schedulability policy. The automaton modeling the scheduler, combined with automata modeling inter-task communication, is analyzed for reachability of the state corresponding to a non-schedulable situation [23].

In general, automata-based methods cover a broader scope of possible scheduling policies, than RMA-like methods do, as they also take into account task interactions. However, the automata-based methods have two disadvantages. The first is that these methods only indicate whether a task set is schedulable or not: they do not provide the response time of the tasks, which would be very useful for architects. The second drawback is that these methods, being based on the construction of an automaton modeling the scheduler, suffer from the state explosion problem. Fortunately, during the last three years, a number of successful accounts about the application of automata-based techniques have been published [5], [7], and [13]. Because of the flexibility of these kinds of techniques, their application is feasible at the architecting level, especially in the cases when standard techniques like RMA are not applicable.

Both types of techniques require information about the worst-case execution time of tasks. Unfortunately, most contemporary methods for the estimation of worst-case execution time cannot be directly applied to architecting, as they are based on already existing code. But at the architectural level, some estimations are often needed before the code is written. Furthermore, there are two problems with traditional approaches:

- Predictions are over-pessimistic due to excluding effects of the acceleration facilities of modern CPU's (pipelines, branch prediction blocks, caches etc). These effects are excluded from the analysis because of unpredictable behavior of the acceleration facilities.
- Variation in behavior due to different input parameters is difficult to account for. This requires analysis of all possible paths of control flow, which is not possible for many situations without providing additional information

describing the relation between input data and program behavior.

It is foreseen that these problems of traditional approaches must also be solved for performing worst-case execution time estimation at the architectural level.

5.2 Memory consumption evaluation.

In many cases, analysis of memory consumption is needed to reason about the feasibility of an embedded system. The allocation of memory can be dynamic or static. Static memory allocation is performed at compile- or load-time, while dynamic memory allocation is performed during run-time. For most real-time operating systems, the memory layout of an application can be presented as follows:

1. Statically allocated memory: the image of program code, static data, stack, and heap
2. Dynamically allocated memory: stacks (for different threads), data objects allocated in the heap.

Analysis of memory availability for the static allocation mechanism is trivial in most cases. It is enough just to summate the sizes of all memory blocks needed for all tasks and compare the result with the amount of available system memory. However, this holds only for binary components.

The situation worsens when dealing with the mechanisms for dynamic memory allocation. In this case, the phenomenon of fragmentation can be observed. Usually, the fragmentation is caused by interleaved sequence of memory block allocations and de-allocations with greatly varying block size. It is rather difficult to evaluate the impact on memory allocation induced by the fragmentation. Moreover, having the memory shared between different tasks results in additional interference that makes the memory behavior even less predictable.

The most common practice for hard real-time systems is to avoid the use of dynamic memory management to increase the predictability and efficiency. Instead, data is allocated statically. Nevertheless, some research on the evaluation of dynamic memory allocation has been done, e.g. in [25] by Zorn et al. Their method employs synthetic allocation traces: the allocation trace of an actual program is modeled with a stochastic process. This method is reported to provide results with 80% accuracy. Thus, it might be applicable for the early analysis of worst-case dynamic memory consumption, as more precise estimations are not needed during the architecting phase. Another approach is based on the abstract interpretation theory; this method automatically transforms a high-level language program into a function calculating the worst-case usage of stack and heap space (see [22]). This approach might also be applicable during the architecting.

6. CONCLUSION

The foreseen framework for component-based software architecting is supposed to address the following: (1) reasoning about *composability of behavior*, (2) the early *assessment* of quality attributes.

For the specification of component *behavior,* timed automata are suggested, while the specification of component interaction is described with Message Sequence Charts (MSC) which allow to vividly represent timing constraints in the end-to-end manner.

For the *assessment* of timeliness, two alternative classes of techniques were considered: analytic and constructive techniques. The applicability of these techniques in the context of component-based software architecture is being validated with industrial case studies.

Currently, we are working on the integration of the proposed architecture description techniques with the evaluation techniques for the analysis of timeliness and memory consumption. Here, timed-automata-based techniques are especially promising, as they have the same formal basis both for the evaluation of quality attributes and the description of component behavior.

There are a number of challenging topics for further research:

- To find an appropriate level of abstraction for component behavior description. There should be a balance between the accuracy of the description (relevant parts of behavior are not omitted) and the complexity of the evaluation
- To elaborate a method for the specification of resource consumption. It is important to be able to integrate the specification of resource consumption in the description of components to enable the evaluation of quality attributes
- To develop a formal methodology for the evaluation of worst-case memory consumption.
- To develop a method for relating MSC-based descriptions to timed-automata-based ones.

7. REFERENCES

[1] G. Abowd, L. Bass, R. Kazman, M. Webb, "SAAM: A Method for Analyzing the Properties of Software Architecture," in Proceedings of the 16th International Conference on Software Engineering, Italy, May 1994

[2] R. Allen and D. *Garlan: A Formal Basis for Architectural Connection*, ACM Transactions on Software Engineering and Methodology, 6(3):213---249, July 1997.

[3] R. Alur, D.L. Dill. *A Theory of Timed Automata*. in: Theoretical Computer Science Vol. 126, No. 2, April 1994, pp. 183-236.

[4] M. Barbacci, S. J. Carriere, P. Feiler, R. Kazman, M. Klein, H. Lipson, T. Longstaff, and C. Weinstock, "Steps in an Architecture Tradeoff Analysis Method: Quality Attribute Models and Analysis", Technical Report CMU/SEI-97-TR-029, 1998

[5] A. Burns. *How to Verify a Safe Real-Time System*. The Application of Model Checking and a Timed Automata to the Production Cell Case Study. Technical report, Real-Time System Research Group, Department of Computer Science, University of York, 1998

[6] A. David, UPPALL 2k: Small Tutorial, http://www.docs.uu.se/docs/rtmv/uppaal/tutorial.pdf

[7] A. Fehnker. *Scheduling a steel plant with timed automata*. In Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA99), pages 280-286. IEEE Computer Society, 1999

[8] D. Giannakopoulou, J. Kramer and S. Cheung, Analysing the Behaviour of Distributed Systems using Tracta. Journal of Automated Software Engineering, special issue on

Automated Analysis of Software. Vol. 6(1) pp. 7-35., January 1999

[9] D. K. Hammer and M.R.V. Chaudron, Component Models for Resource-Constraint Systems: What are the Needs?, Proc. 6th Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), Rome, January 2001.

[10] D.K. Hammer, Component-based architecting for distributed real-time systems: How to achieve composability?, Int. Symposium on Software Architectures and Component Technology (SACT), Enschede, Netherlands, January 2000, to be published by Kluwer.

[11] D. I. Katcher, S. S. Sathaye, J. K. Strosnider. Fixed priority scheduling with limited priority levels. IEEE Transactions on Computers, 44(9):1140--1144, 1995

[12] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, H. Gonzalez, *A Practitioners Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston, MA: Kluwer Academic Publishers, 1993

[13] K. J. Kristoffersen, K. G. Larsen, P. Pettersson, and C. Weise, Experimental Batch Plant - VHS Case Study 1 Using Timed Automata and UPPAAL, *Deliverable of EPRIT-LTR Project 26270 VHS* (Verification of Hybrid Systems), 1999

[14] P.B. Ladkin, S. Leue, Interpreting message flow graphs, Formal Aspects of Computing, 7(5):473-509, 1995

[15] G.T. Leavens, M. Sitaraman, Foundations of component-based systems, Cambridge University Press, 2000.

[16] J. Lehoczky, L. Sha, Y. Ding, "*The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior*," IEEE Real Time Systems symposium, 1989

[17] C. Liu, J. Layland, "*Scheduling Algorithms for Multiprogramming in Hard Real Time Environment*", JACM, 1973

[18] J. Magee, N. Dulay, S. Eisenbach, J. Kramer. *Specifying Distributed Software Architectures*. In Proceedings of 5th European Software Engineering Conference, Spain, 1994

[19] R. van Ommering, F. van der Linden and J. Kramer and J. Magee, The Koala Component Model for Consumer Electronics Software. Computer 33, 3 (2000), pp 33-85, 2000

[20] M.A. Reniers, Message Sequence Chart, Syntax and Semantics, Ph.D. thesis, TUE, 1999

[21] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1998.

[22] L. Unnikrishnan, S. D. Stoller, Y. A. Liu. *Automatic accurate stack space and heap space analysis for high-level languages*. Technical Report TR 538, Computer Science Department, Indiana University, Feb. 2000

[23] A. Wall, C. Ericsson, and W. Yi. *Timed Automata as Task Models for Event-Driven systems*. In Proceedings of RTSCA 99. IEEE Press, 1999.

[24] J. Warner, A. Kleppe, "The Object Constraint Language", Addison Wesley, 1999.

[25] B. Zorn, D. Grunwald. *Evaluating models of memory allocation*. ACM Transactions on Modeling and Computer Simulation, 1(4):107--131, 1994.

Type Handling in a Fully Integrated Programming and Specification Language

Gregory Kulczycki
Clemson University
Clemson, SC
gregwk@cs.clemson.edu

ABSTRACT

Integrated languages combine formal specification and programming features, and make it possible to specify, implement, and verify programs within the same framework. This paper examines the consequences of this fundamental integration on the type system of a software engineering language, using RESOLVE as an example. It explains why name matching for program types coexists naturally with structural matching for math types. It describes a formulation of set theory and its relationship to the type system. And it poses a variety of discussion questions concerning the use of types and subtypes in the specification portion of the language.

Keywords

Type checking, verification, subtypes, set theory, software engineering

1. INTRODUCTION

Verification of component-based software requires languages that integrate programming and specification features, and types are at the heart of this integration. Programming languages are not suited for specification, and specification languages are not used for implementation. The elements of both languages must be integrated to verify that an implementation is correct with respect to a specification. This requires that programming objects—in particular, types—be described in mathematical terms. A wealth of papers have been written about types and type systems, but these papers invariably focus on types in programming (implementation) languages or types in specification languages. The contribution of this paper is its description of a type system for languages concerned with both implementations and specifications.

The desire to build predictable, component-based software has compelled many in the software verification community to develop integrated languages—languages that combine formal specification with programming. Examples of such unions include JML, Eiffel, RESOLVE/C++[1], Z variants, and Larch variants [3, 7, 9, 15]. Most of these integrated languages have resulted from appending a specification language onto a preexisting programming language. In contrast, RESOLVE [12, 14] has been developed from the beginning as both a specification and a programming language. The language is only one part of the RESOLVE system for predictable software development. The system also includes a framework and discipline for building software that is—among other things—reusable, verifiable, efficient, and understandable. The language is intimately tied to the framework and discipline.

For the past few years the author of this paper has been involved in designing and implementing tools that would bring RESOLVE into the world of practical programming. The current focus of this effort is the development of a RESOLVE compiler. The project is complex, not only because the compiler must deal with a programming and specification language combined, but because ongoing research makes the language a moving target (e.g., performance specification and verification [13]). During the course of writing the compiler we have been forced to refine our ideas about how types should be handled in both mathematical and programming contexts.

This paper addresses the following question: What are the implications for the type system in a language that integrates programming and specification? Using RESOLVE as an example, we look for answers to this question. Section 2 presents the type model of RESOLVE and demonstrates how types are treated in programming and mathematical contexts. Section 3 summarizes Ogden's formulation of set theory in RESOLVE [10] and explains how it relates to types. Finally, section 4 examines a few specific issues involving math types and subtypes.

2. OVERVIEW OF TYPES

An integrated language is much more complex than either a programming language or specification language alone, so simplicity is a primary concern. It is essential to have type matching rules that are easily understandable. A programmer (or compiler) should not have to sift through a myriad of rules and exceptions simply to evaluate the type of an expression.

[1]RESOLVE/C++ uses only the specification portion of the RESOLVE language

```
Concept Stack_Template(type Entry;
        evaluates Max_Depth: Integer);
    uses Std_Integer_Fac, String_Theory;
    requires Max_Depth > 0;

    Type Family Stack is modeled by Str(Entry);
        exemplar S;
        constraints |S| ≤ Max_Depth;
        initialization ensures S = Λ;

    Operation Push(alters E: Entry; updates S: Stack);
        requires |S| < Max_Depth;
        ensures S = ⟨#E⟩ ∘ #S;

    . . .

end Stack_Template;
```

Figure 1: A Concept for Stack

Mathematical and programming elements in the RESOLVE language are kept as distinct as possible. Thus, assertions in requires and ensures clauses[2] of operations are strictly mathematical expressions, and conditions in while loops and if statements are strictly programming expressions. Likewise, all variables and types found in a mathematical expression are math variables and math types, and those found in programming expressions are program variables and program types. This means that the same name has a different type depending on whether it appears in a programming or mathematical context. Furthermore, mathematical expressions and programming expressions are type-checked differently—in mathematical expressions, types are matched according to structure, whereas in programming expressions, they are matched strictly by name.

2.1 Math vs Program Context

Figure 1 shows a RESOLVE specification of a Stack component. This simple example turns out to be sufficiently powerful to illustrate the ideas in this paper. The **Type Family** declaration introduces the program type Stack and gives its mathematical model. We use *Type Family* instead of just *Type* because the concept (and therefore the type) is generic until it is instantiated, so the declaration of Stack here encompasses an entire family of types. In the type family declaration, the left side contains the program type Stack, and the right side contains the math type Str(Entry). The fact that mathematical and programming elements come together in a type declaration underscores the fundamental role that types play in an integrated language. The **exemplar** introduces a variable of type Stack to describe properties that hold for any arbitrary variable of type Stack. For example, the **constraints** clause indicates that the length of any Stack must always be less than Max_Depth.

In the specification of **Operation** Push, parameters E and S are program variables. When a call is made to this operation, the compiler checks that the first argument to Push is of type Entry, and the second argument is of type Stack. When S appears in the requires clause, however, the compiler analyzes it as a math variable. The variable S has been declared as program type Stack, but since the variable occurs in a mathematical context, the compiler instead uses the mathematical model of Stack given in the type family declaration. So the variable S appearing in the requires clause has math type Str(Entry). The rest of the concept is analyzed similarly.

[2]preconditions and postconditions

```
Realization Array_Realiz for Stack_Template;

    Type Stack is represented by Record
            Contents: Array 1..Max_Depth of Entry;
            Top: Integer;
        end;
        conventions 0 ≤ S.Top ≤ Max_Depth;
        correspondence
            Conc.S = ( ∏_{i=1}^{|S.Top|} ⟨S.Contents(i)⟩ )^Rev ;
        initialization
            S.Top := 0;
        end;

    Procedure Push(alters E: Entry; updates S: Stack);
        . . .
    end Push;

    . . .

end Array_Realiz;
```

Figure 2: A Realization for Stack

In RESOLVE, like in other model-based languages such as VDM and Z, a handful of math types are used for modeling many different program types. This mirrors scientific disciplines like Physics, where the same mathematical model is used to capture widely different concepts. Different program types such as Stack and Queue may both be modeled using mathematical strings. This makes it convenient to write specifications such as the one shown here:

```
Operation Stk_Q_Transfer(clears S: Stack;
                         replaces Q: Queue);
    ensures Q = #S^Rev;
```

2.2 Structural vs Name Matching

The implementation or realization of Stack_Template in Figure 2 introduces a Stack type with a specific programming structure. It indicates how a Stack is represented for this particular realization. The **conventions** clause provides the representation invariant—it indicates which representation states are permitted. The **correspondence** clause, or abstraction relation, shows how this representation is related to the mathematical model of Stack given in the concept. Notice that the correspondence clause contains two variables, S and **Conc**.S, that are not declared directly in this scope. These variables are derived from the special exemplar variable in the concept's type declaration. Figure 3 illustrates what the compiler does when analyzing the declaration of Stack in a realization. It locates the exemplar from the type

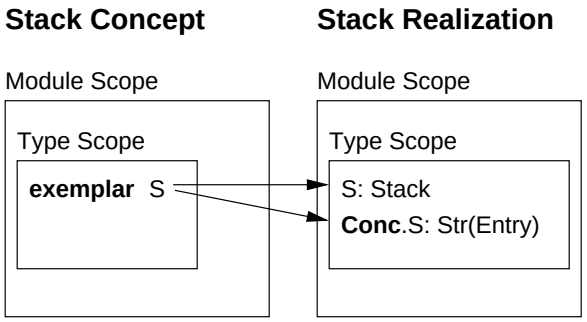


Figure 3: Affect of Exemplar on Realization

family declaration in the corresponding concept, and uses its name to create two variables within the type scope of the realization. The first variable is named S and has program type Stack. The second variable is named **Conc**.S (read as "the conceptual value of S") and has math type Str(Entry), the mathematical model of Stack. The correspondence clause describes the relationship between these two variables.

Program type matching in RESOLVE is done strictly by name. This is reasonable because a primary motivation for introducing different type names is to keep objects of different types distinct. Also, in a language that separates interfaces (as specifications) from implementations, clients will not have access to the structural programming representation of a type, so that structural matching can not be accomplished consistently.

Math type matching is done by structure. The structure consists of math types that can be simple or composite. If a program type name is encountered in a mathematical context, the compiler uses its corresponding mathematical model to convert it to a math type expression. In RESOLVE, like in Z, built-in composite types include set theory operators $\times$, $\rightarrow$, and $\mathcal{P}$. Composite types are parameterized types that take other types as arguments. RESOLVE also permits the use of user-defined composite types. In the type expression Str(Entry), Str is a user-defined composite type (defined in the module String_Theory, which is imported through the **uses** clause in Figure 1). For a composite math type to match another by structure, the types of their arguments must also match. For example, Queue $\times$ Fahrenheit matches Stack $\times$ Centigrade if and only if the mathematical models of Queues and Stacks match, *and* the mathematical models of Fahrenheit and Centigrade match. Constraints on mathematical models—given by the constraint clause in the type family declaration—are ignored by the analyzer; checking constraints is the responsibility of the verifier.

To illustrate the difference between name matching in the programming world and structural matching in the math world, consider the type declaration of Stack in figure 2. The representation uses both a record and an array, which are composite program types. In RESOLVE, the type Record is modeled by a Cartesian product (denoted by the infix operator $\times$), and the type Array is modeled by a function

Table 1: Type Evaluations of Variables

Variable	Program Type	Math Type
S	Stack	$(\mathbb{Z} \rightarrow \text{Entry}) \times \mathbb{Z}$
S.Contents	%Array(10,20)	$\mathbb{Z} \rightarrow \text{Entry}$
S.Contents(1)	Entry	Entry
S.Top	Integer	$\mathbb{Z}$

(denoted by the infix operator $\rightarrow$)[3] [10]. Also, the program type Integer is modeled by the mathematical integers $\mathbb{Z}$.[4] The generic type Entry is treated as a primitive type when seen from a math context because its math model is not known before instantiation.

Now consider a variable S of type Stack. Table 1 shows how a compiler will evaluate the variables in the first column depending on whether they occur in a program or math context. For example, if S.Contents occurs in a requires clause, it evaluates to the math type $\mathbb{Z} \rightarrow$ Entry. If the variable S.Top occurs in the condition of a while loop, it evaluates to the program type Integer. The type %Array(10,20) is a unique name created by the compiler.

A compiler for RESOLVE must keep track of more type information than typical compilers. It must have access to the program name of the type, the program structure of the type, and the math structure of the type. The program structure of the type is not needed for matching purposes, but it is needed to determine whether variables of that type may use the special syntax of Records or Arrays. For example, since the type Stack in the realization above is structurally a record, any variable S of type Stack can use special syntax to refer its fields—S.Contents and S.Top.

3. SET THEORY

Sets are the fundamental building blocks of the RESOLVE language. There are several reasons why sets are a natural choice. First and most importantly, sets are foundational to Mathematics. All programming objects must have a mathematical model, and sets can be used to describe any mathematical domain. No matter how complicated a real world problem is, it can be captured with sets. The same could not be said if we were to use, say, real numbers, as the building blocks of the language. Another reason for using sets is that the basic notions of sets—membership, union, subset, and so forth—are familiar to most students and programmers. Finally, sets are flexible enough to describe the language itself.

3.1 Echelons

The particular flavor of set theory used in RESOLVE has been developed by Bill Ogden at The Ohio State University [10]. The core of the theory is traditional: It starts with the notion of a universe of all sets ($\mathcal{S}et$) and uses the notion of membership ($\in$) as a basis for defining all the operators we expect to see on sets ($\cup$, $\cap$, $\subseteq$, $\mathcal{P}$, $\rightarrow$, etc.). A distinguishing

[3]Strictly speaking, the RESOLVE type Array is modeled by a Cartesian product composed of a function and two integers—one for each bound.

[4]This model will obviously have constraints, involving minimum and maximum values, but recall that constraints are ignored during type-checking

aspect of the theory is the notion of special sets known as *echelons*. Echelons are large universes of sets that are closed under the operations of ordinary set theory, such as unions and power sets.

The motivation for echelons comes from the need to provide a collection of sets that is (1) large enough to model everything one would normally want to model in a computer program, and (2) small enough that it does not exhaust all the sets in $\mathcal{S}et$. Henceforth, let the set $\mathbb{S}$et (pronounced "fat set") denote the collection which we draw from to model all program objects in our language. Certainly $\mathbb{S}$et must have sufficient modeling power for all programming objects. Using $\mathcal{S}et$ as $\mathbb{S}$et, however, would not leave a specifier any sets to describe the language with. For example, one would require sets that were *larger* than $\mathbb{S}$et when writing the specifications for a RESOLVE compiler that was written in RESOLVE.

To provide sufficient models for programming objects, $\mathbb{S}$et must be closed under the basic type operations of the language. Assume A and B are types that are modeled by sets in $\mathbb{S}$et. Then any type expression that can be derived from A and B must also be contained in $\mathbb{S}$et. RESOLVE currently permits the operators $\times$, $\rightarrow$, and $\mathcal{P}$ in type expressions. Therefore, if A and B are elements of $\mathbb{S}$et, A $\times$ B, A $\rightarrow$ B, and $\mathcal{P}$(A) must also be elements of $\mathbb{S}$et.

Echelons are closed under these basic operations. The properties of echelons include closure under membership, pairing, unions and power sets, which means they are also closed under operators $\times$ and $\rightarrow$.[5] We can define an echelon operation on A, $\mathcal{E}$(A), to be the smallest echelon that contains A. If we take $\mathbb{E}_0 = \phi$, then $\mathbb{E}_1 = \mathcal{E}(\mathbb{E}_0)$ contains the sets $\phi, \mathcal{P}(\phi), \mathcal{P}(\mathcal{P}(\phi)), \ldots$, which are traditionally used to model the natural numbers. It can be shown that $\mathbb{E}_1$ is only countably infinite, so it will not be large enough for real world models. $\mathbb{E}_2 = \mathcal{E}(\mathbb{E}_1)$, however, does provide sufficient sets. It contains models for $\mathbb{N}, \mathbb{R}, \mathcal{P}(\mathbb{R}), \mathbb{R} \times \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$, etc.

If $\mathbb{S}$et is at least $\mathbb{E}_2$ we know it has sufficient modeling power for all ordinary programs. In RESOLVE, $\mathbb{S}$et is generally assumed to be $\mathbb{E}_2$, but whether it is $\mathbb{E}_2$, or $\mathbb{E}_3$, or $\mathbb{E}_{100}$, the important fact is that a specifier still has access to $\mathcal{E}(\mathbb{S}\text{et})$ to describe the language itself. A rigorous treatment of echelons can be found in [10]. The objective of this summary is only to present enough information to give an idea of their significance for program specification.

3.2 Primitive Types

If sets are the building blocks of the RESOLVE language, then primitive types are the cornerstones on which the other blocks rest. Declarations of primitive types take the form:

$$\begin{aligned} \mathrm{T}_0 &: \ \mathbb{S}\text{et} \\ \mathrm{T}_1 &: \ \mathbb{S}\text{et} \rightarrow \mathbb{S}\text{et} \\ \mathrm{T}_2 &: \ \mathbb{S}\text{et} \times \mathbb{S}\text{et} \rightarrow \mathbb{S}\text{et} \\ \mathrm{T}_3 &: \ \mathbb{S}\text{et} \times \mathbb{S}\text{et} \times \mathbb{S}\text{et} \rightarrow \mathbb{S}\text{et} \\ &\vdots \end{aligned}$$

[5] Assuming appropriate definitions of $\times$ and $\rightarrow$, we can show that A $\times$ B $\subseteq \mathcal{P}(\mathcal{P}(\bigcup\{\mathrm{A}, \mathrm{B}\}))$, and $A \rightarrow B \subseteq \mathcal{P}(A \times B)$.

Most often only the first two will be seen. The first type, T_0, is a simple type, while the remaining types are composite. Like all composite types, primitive composite types cannot be used in isolation—they must have parameters. For example, if Str: $\mathbb{S}$et $\rightarrow$ $\mathbb{S}$et, then one cannot declare x: Str, but one can declare an x: Str(Gamma), where Gamma: $\mathbb{S}$et.

A primitive type, like every other object in RESOLVE, is a set. Abstractly, a type is distinguished from other sets of the same cardinality by its properties. For example, it can be shown that the sets $\mathbb{N}$ and $\mathbb{Z}$ have the same cardinality, but the set $\mathbb{N}$ is not closed under subtraction, while the set $\mathbb{Z}$ is. Primitive types in RESOLVE are introduced via two constructs. First, a definition spells out the properties of the type:

Def Is_Natural_Number_Like(N: $\mathbb{S}$et, 0: N,
suc: N $\rightarrow$ N): $\mathbb{B}$ =
(*P1*) $\forall$n : N, suc(n) $\neq$ 0 **and**
(*P2*) Is_Injective(suc) **and**
(*P3*) $\forall$S : $\mathcal{P}$(N),
if 0 $\in$ S $\wedge \forall$n : N, n $\in$ S $\Rightarrow$ suc(n) $\in$ S
then S = N;

Then an assumption introduces a set that satisfies that definition:

Assumption Is_Natural_Number_Like($\mathbb{N}$, 0, suc);

The properties in the definition (P1–P3) mirror the axioms one would normally see in an axiomatic description of the natural numbers. The approach of using definitions to describe the properties of a type simplifies the semantics of the language—we do not have to concern ourselves with special syntax and semantics for signatures and axioms. In RESOLVE, definitions are used to introduce all mathematical objects, whether they are constants, variables, functions, or types. The above assumption indicates that the set $\mathbb{N}$ (together with sets 0 and suc) is any arbitrary model of the natural numbers. This enforces abstractness because the natural numbers are not identified with one particular representation.

3.3 Objects as Sets

Every programming object in the RESOLVE language can be modeled by a set contained in $\mathbb{S}$et. That is, any variable, function, or type that occurs in a programming context must lie within $\mathbb{S}$et. Though math objects are exempt from this restriction, most math objects seen in programs will also be in $\mathbb{S}$et because they are typically used to describe program objects. When we want to describe complex software like compilers and verifiers, our specifications will draw on objects that lie outside of $\mathbb{S}$et.

Simple primitive types are directly contained in $\mathbb{S}$et, and composite primitive types always take parameters, which also puts them in $\mathbb{S}$et. The set operators that we are permitted to use in math type expressions ($\times$, $\rightarrow$, and $\mathcal{P}$) are all closed under echelons. Since all math types that are used to model program types are constructed by applying composite types and set operators to other math types, all such math types are in $\mathbb{S}$et.

All program types have a mathematical model, which is a math type expression. The declaration:

Type Family Stack **is modeled by** Str(Entry);

is the text equivalent to:

Type Family Stack $\subseteq$ Str(Entry);

The subset operator is used instead of the equal operator because of constraints on the model. For an example, see Figure 1.

All programming objects in RESOLVE belong to some type, as indicated by the type membership operator (:). Since all types are modeled by sets in $\mathbb{S}$et, the type membership operator can be replaced with set membership ($\in$) to describe the mathematical relationship between an object and its type. Finally, since $\mathbb{S}$et is closed under membership, all programming objects must be in $\mathbb{S}$et.

4. DISCUSSION TOPICS

During type-checking, a compiler needs to be concerned with a number of questions, such as how to treat subtypes, when to require casts, when to report errors, and when to give warnings. Although these questions must be answered for both program and math types, we focus on how they apply to math types, mainly because of the rich diversity of views on how types should be handled in specification languages, ranging from traditionalists [2, 3, 15] to those whose type systems incorporate theorem provers [11] to those who question the necessity of type systems altogether [6]. Issues involving program subtypes will largely depend on how the language in question handles polymorphism, a topic that merits a separate paper.

The distinction between types and other objects (variables and functions) is quite clear in the programming world: program types are introduced by the keyword **Type**. However, in the math world all objects—variables, functions, and types—are introduced by the keyword **Definition** or through quantifiers in expressions. This uniformity is intentional, since all objects are sets, but it forces specifiers and compilers to rely on other cues to tell them which mathematical objects can be used as types. Examples based on subtypes are discussed in this section.

If an object T is declared to be of type $\mathcal{P}(A)$ where A is a type, then T is also a type, and we say that T is a subtype of A. Permitting such declarations requires the language to have reasonable semantics for handling the relationship between the type T being declared and the type A being used in the declaration. Consider the definition:

Definition Even : $\mathcal{P}(\mathbb{N}) = \{n : \mathbb{N} \mid n \textbf{ mod } 2 = 0\}$;

It is reasonable to want to declare objects of type Even and add them together using the + operator defined in Natural_Number_Theory (the theory introducing $\mathbb{N}$). The type of the result would be $\mathbb{N}$, so a specifier could write:

$\forall e1, e2 : \text{Even}, \exists n : \mathbb{N} \ni 2 \cdot n = e1 + e2;$

To analyze this expression, a compiler needs to know that Even is a subtype of $\mathbb{N}$, and it must have an algorithm that determines which + operator to use if there is more than one choice. This can become non-trivial, and as a rule, if something is complex for the compiler, it is also conceptually complex for the programmer or specifier. One way to simplify things is to require explicit type casting, so the above expression would produce an error if there were no + operator defined that took two objects of type Even. To use the + from natural number theory, a specifier might be forced to write:

$\forall e1, e2 : \text{Even}, \exists n : \mathbb{N} \ni 2 \cdot n = (\mathbb{N})e1 + (\mathbb{N})e2;$

This makes the expression harder to write since the specifier must do the work that the compiler would have done to decide which + should be be used. In RESOLVE, where emphasis is on qualities such as reuse and understandability, readability usually takes precedence over writability. In this example, there is an argument for both sides in terms of readability. If the + operator is overloaded in an unconventional way, explicit casting may clarify things; if the + operator is not overloaded at all, explicit casting simply adds unnecessary clutter to the expression.

In some programming languages, casting to a parent type is implicit, but casting to a subtype must be explicit. An analogous example in the math world might define:

Definition Vertex : $\mathcal{P}(\mathbb{Z}) = \{z : \mathbb{Z} \mid 1 \leq z \leq \text{Max\_Vert}\}$;

Definition Cost(G : Graph; v1, v2 : Vertex) : $\mathbb{R}$[6]

If casting to a subtype is required, one must write

$$\forall G : \text{Graph}, \forall z1, z2 : \mathbb{Z}, \quad \text{Cost}(G, (\text{Vertex})z1, (\text{Vertex})z2) \leq 4.7; \tag{1}$$

instead of

$$\forall G : \text{Graph}, \forall z1, z2 : \mathbb{Z}, \text{Cost}(G, z1, z2) \leq 4.7; \tag{2}$$

This may seem quite reasonable for a programmer, but some specifiers may consider the following expression perfectly reasonable:

$$\forall G : \text{Graph}, \forall z1, z2 : \mathbb{Z}, \textbf{ if } z1, z2 \in \text{Vertex} \quad \textbf{then } \text{Cost}(G, z1, z2) \leq 4.7; \tag{3}$$

There is nothing wrong with expression (3) as a mathematical formula, and it is obvious that the Cost function is defined for all $z1, z2 \in$ Vertex. But if we insist that the compiler must report a type error for expression (2), then we must insist that it does the same for expression (3). There may be merit in exploring ways that allow the specifier more flexibility in writing expressions while still insisting that he provide sufficient clues to the compiler of his intentions. For example, we might allow:

$$\forall G : \text{Graph}, \forall z1, z2 : \mathbb{Z}, \textbf{ if } z1, z2 : \text{Vertex} \quad \textbf{then } \text{Cost}(G, z1, z2) \leq 4.7; \tag{4}$$

[6]We can imagine that the Cost function indicates the expense of traveling from v1 to v2 in graph G.

Expression (4) replaces the set membership operator ($\in$) of expression (3) with a type membership operator (:). This could indicate to the compiler that z1 and z2 are to be treated as belonging to type Vertex for the remainder of the expression scope. Unfortunately, we would have to develop another mechanisms for the case where the **if** part of expression (4) were in a precondition and the **then** part of the expression were in a postcondition. If we introduce too many distinct mechanisms for handling a conceptually similar situation we run the risk of significantly complicating the language.

All of the questions that arise with subtypes due to the power set operator may occur with primitive types as well. It is reasonable to think of $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$ as distinct types—after all, their algebraic structures are different. It is also reasonable to want to treat $\mathbb{N}$ as a subset of $\mathbb{Z}$ and $\mathbb{Z}$ as a subset of $\mathbb{R}$. If these relationships between primitive types are desired, a mechanism different from the one for subtypes must be provided that allows the compiler to treat them as such.

5. RELATED WORK

Many examples of integrated languages exist, though the degree of integration varies widely. Eiffel [9] is essentially a programming language with a few specification features built in. Like RESOLVE, it was created independently of any preexisting programming language; unlike RESOLVE, its specification features are limited—it does not include a complete formal specification language (see p. 400 of [9]). JML (Java Modeling Language) [7] is a behavioral specification language that was created for Java. Used together, JML and Java form an integrated language. Unlike Eiffel, JML provides models for its programming objects. Mathematical expressions in both Eiffel and JML are designed to look similar to programming expressions. Accordingly, they will also type-check similarly. Recall that RESOLVE type-checks mathematical expressions by structure and programming expressions by name. RESOLVE/C++ [5] applies the RESOLVE framework and discipline to the C++ programming language. It uses the specification portion of the RESOLVE language for reasoning. Integrated languages formed by combining a preexisting specification language with a preexisting programming language will type-check mathematical expressions in accordance with the rules of the specification language and will type-check programming expressions in accordance with the rules of the programming language.

Most practical specification languages allow some form of subtyping [2, 7, 15]. The PVS verification system [11] permits downcasting to predicate subtypes by generating a proof obligation when a type is detected in a place where its subtype is expected. Problems similar to those presented in section 4 cause Lamport to question whether specification languages should be typed at all [6]. Topics relating to program subtypes include behavioral subtypes [8] and matching [1].

6. CONCLUSION

Integrated languages must have an effective method for handling program and math types. Integration requires that a mechanism exist for relating programming and mathematical elements. Type declarations are a natural place to describe this relationship. Practical concerns compel us to treat programming and mathematical objects differently. Program types should match according to their names, and math types should match according to their structure.

The theoretical basis of the specification language will affect which objects can be used as types, and will determine the kinds of models that can be constructed for program objects. We need to distinguish sets that model real world objects in a language from larger sets that are needed to describe compilers and verifiers for that language.

The handling of subtypes in the specification portion of an integrated language offers a series of trade-offs. Systems that allow a specifier greater flexibility in writing expressions run the risk of permitting poor expressions that could be caught quickly with a less tolerant type system.

7. ACKNOWLEDGMENTS

Several people contributed important ideas to this work and made helpful comments about drafts of this article. I would especially like to thank Murali Sitaraman, Bill Ogden and Steven Atkinson.

We also gratefully acknowledge financial support from our own institutions, from the National Science Foundation under grants CCR-0113181, DUE-9555062, and CDA-9634425, from the Fund for the Improvement of Post-Secondary Education under project number P116B60717, and from the Defense Advanced Research Projects Agency under project number DAAH04-96-1-0419 monitored by the U.S. Army Research Office. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation, the U.S. Department of Education, or the U.S. Department of Defense.

8. REFERENCES

[1] M. Abadi and L. Cardelli. On subtyping and matching. *ACM Transactions on Programming Languages and Systems*, 18(4):401–423, July 1996.

[2] D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math.* Springer-Verlag, New York, 1993.

[3] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, 1993.

[4] D. E. Harms and B. W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

[5] J. Hollingsworth, L. Blankenship, and B. W. Weide. Experience report: Using RESOLVE/C++ for commercial software. In *Proceedings SIGSOFT FSE.* ACM, November 2000.

[6] L. Lamport and L. C. Paulson. Should your specification language be typed? *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, May 1999.

[7] G. T. Leavens, A. A. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe,

and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12. Kluwer, 1999.

[8] G. T. Leavens and K. K. Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, Cambridge, United Kingdom, 2000.

[9] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, Upper Saddle River, New Jersy, 2nd edition, 1997.

[10] W. F. Ogden. *The Proper Conceptualization of Data Structures*. The Ohio State University, Columbus, OH, 2000.

[11] J. Rushby. Subtypes for specifications. In *Software Engineering - ESEC/FSE '97*, pages 4–19. ACM SIGSOFT, September 1997.

[12] M. Sitaraman, S. Atkinson, G. Kulczycki, B. W. Weide, T. J. Long, P. Bucci, W. Heym, S. Pike, and J. E. Hollingsworth. Reasoning about software-component behavior. In *Procs. Sixth Int. Conf. on Software Reuse*, pages 266–283. Springer-Verlag, 2000.

[13] M. Sitaraman, W. F. Ogden, G. Kulczycki, J. Krone, and A. Reddy. Performance specification of software components. In *Proceedings of SSR '01*, pages 3–10. ACM/SIGSOFT, May 2001.

[14] M. Sitaraman and B. W. Weide. Component-based software using RESOLVE. *ACM Software Engineering Notes*, 19(4):21–67, 1994.

[15] J. Spivey. *The Z Notation*. Prentice Hall, New York, 1989.

A Formal Approach to Software Component Specification

Kung-Kiu Lau
Department of Computer Science
University of Manchester
Manchester M13 9PL
United Kingdom
kung-kiu@cs.man.ac.uk

Mario Ornaghi
Dip. di Scienze dell'Informazione
Universita' degli studi di Milano
Via Comelico 39/41, 20135 Milano
Italy
ornaghi@dsi.unimi.it

Abstract

There is a general consensus that the paradigm shift to component-based software development should be accompanied by a corresponding paradigm shift in the underlying approach to specification and reasoning. Work in modular specification and verification has shown the way, and following its lead, in this position paper, we outline our approach to specifying and reasoning about components, which uses a novel notion of correctness.

1 What is this paper about?

As the title suggests, this paper is about an approach to formal specification of software components. The purpose of such an approach is to allow formal reasoning about components. The ultimate goal of Component-based Software Development (CBD) is *third-party assembly*. To achieve this, it is necessary to be able to specify components in such a way that we can reason about their construction and composition, and correctness thereof, *a priori*. Work in modular specification and verification, e.g. [9, 14] has shown the way, and our approach follows its lead. However, our approach is novel and hence different in the way we define correctness. In this paper, we will discuss how we specify components, and in particular how we define and reason about correctness, and why this is useful for CBD.

2 Specifying Components

Ideally components should be *black boxes*, in order that users can (re)use them without knowing the details of their innards. In other words, the *interface* of a component should provide *all* the information that users need. Moreover, this information should be the *only* information that they need. Consequently, the interface of a component should be the *only* point of access to the component. It should therefore contain all the information that users need to know about the component's *operations*, i.e. what its code does, and its *context dependencies*, i.e. how and where the component can be deployed. The code, on the other hand, should be completely inaccessible (and invisible), if a component is to be used as a black box.

The *specification* of a component is therefore the specification of its *interface*, which must consist of a precise definition of the component's operations and context dependencies, and nothing else.

3 Reasoning about Components

To reason about components and their construction and composition, we will coin a phrase, *a priori reasoning*, which is essential for CBD to achieve its goal of third-party assembly. As its name suggest, *a priori reasoning* takes places *before* the construction takes place, and should therefore provide an *assembly guide* for component composition.

For CBD, *a priori reasoning* would work as follows:

- it requires that it is possible to show *a priori* that the individual components in question are *correct* (wrt their own specifications);

  (This enables us to do *component certification*, see below.)

- it then offers help with reasoning about the *composition* of these components:

  - to guide their composition in order to meet the specification of a larger system;
  - to predict the precise nature of any composite, so that the composite can in turn be used as a unit for further composition.

  (This enables us to do *system prediction*, see below.)

4 Predictable Component Assembly

A priori reasoning addresses an open problem in CBD, viz. *predictable component assembly*. It does so because it enables component certification and system prediction.

Consider Figure 1. Two components A and B each have their own interface and code. If the composition of A and

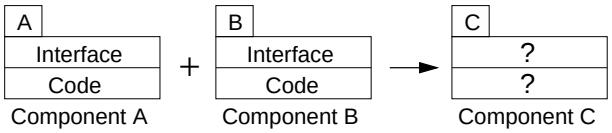


Figure 1. Predicting component assembly.

B is C, can we determine or deduce the interface and code of C from those of A and B? The answer lies in component certification.

4.1 Component Certification

Certification should say what a component does (in terms of its *context dependencies*) and should guarantee that it will do precisely this (for all contexts where its dependencies are satisfied). A certified component, i.e. its interface, should therefore be specified properly, and its code should be verified against its specification. Therefore, when using a certified component, we need only follow its interface. In contrast, we cannot trust the interface of an uncertified component, since it may not be specified properly and in any case we should not place any confidence in its code.

In the context of *a priori reasoning*, a certified component A is *a priori correct*. This means that:

- A is guaranteed to be correct, i.e. to meet its own specification;
- A will always remain correct even if and when it becomes part of a composite.

This is illustrated by Figure 2, where component A has been

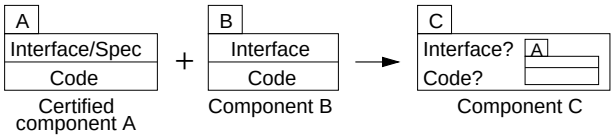


Figure 2. Component certification.

certified, so we know how it will behave in the composite C.

However, we do not know how B will behave in C, since it is not certified. Consequently, we cannot expect to know C's interface and code from those of A and B, i.e. we cannot predict the result of the assembly of A and B.

4.2 System Prediction

For system prediction, obviously we need *all* constituent components to be certified (*a priori correct*). Moreover, for any pair of certified components A and B whose composition yields C:

- *before* putting A and B together, we need to know what C will be;
- and furthermore, we need to be able to certify C.

This is illustrated by Figure 3. The specification of C must

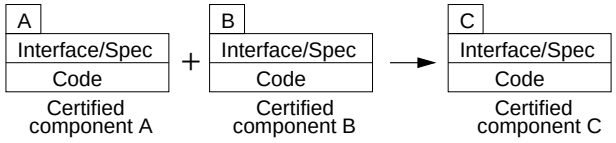


Figure 3. System prediction.

be predictable prior to composition. Moreover, we need to know how to certify C properly, and thus how to use C in subsequent composition. *A priori correctness* is just what we need in order to do system prediction.

5 Modular Specification and Verification

Current approaches to modular (formal) specification and verification, e.g. [9, 14], use *modular reasoning*. This is specification-based reasoning that tries to say before running the software whether it will behave as specified or not (subject to relevant assumptions). This is illustrated in Figure 4. Before a composite module C is deployed, we can

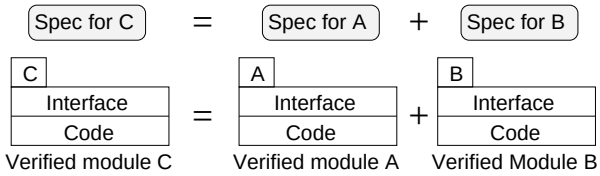


Figure 4. Module composition.

predict whether it will work according to its specification. For example, if component modules, say A and B, are to be used in C, the correctness of C is established based on the specifications of A and B (even before A and B have been implemented). The components A and B are then verified independently. The contexts of A and B are taken in account when using and verifying A and B.

Thus modular reasoning is *a priori* in nature. It predicts correctness, based on specification. This kind of prediction is we believe subtly different from the prediction that we

intend to convey in Figure 3, which predicts specification, based on (certified) correctness (we will discuss this in Section 11).

6 Our Approach to Specifying Components

In the rest of this paper, we outline our approach to specifying components, so that we can carry out a priori reasoning about their construction and composition. Our approach differs from current work in modular specification and verification, however, in that we use a novel notion of a priori correctness.

Diagrammatically, our component looks like Figure 5, and in the subsequent sections, we will explain the key in-

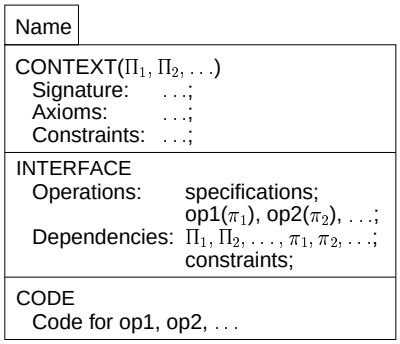


Figure 5. Ingredients of a component.

gredients, viz. the *context* and the *interface*, and their specifications.

We should point out that this is work in progress, so we do not yet have all the answers, so to speak.

7 Context

A component is defined in a *problem domain*, or a *context*. We will represent a context as a full first-order logical theory with an intended (mathematical) model.

7.1 Signature and Axioms

A *context* $\mathcal{C} = \langle \Sigma, X \rangle$ is composed of a *signature* Σ (containing *sort* symbols, *function* declarations and *relation* declarations) and a finite or recursive set X of Σ-axioms. A context axiomatises a problem domain and thus enables us to reason about it. More specifically, a context contains the *abstract data types* (ADTs) and all the concepts that are needed to build a model of the application at hand. A context is thus a (first-order) theory with an intended model.

We distinguish between *closed* and *open* (or *parametric*) contexts. A context $\mathcal{C} = \langle \Sigma, X \rangle$ is *closed* if its signature Σ does not contain any parameters. In this case, $\mathcal{C}$'s axioms X have one fixed model. By contrast, a context $\mathcal{C} = \langle \Sigma, X \rangle$ is *open* if its signature Σ contains parameters. In this case, $\mathcal{C}$'s axioms X have many potential models, depending on the parameters in the signature Σ.

Example 7.1 A simple example of a closed context is first-order arithmetic $\mathcal{NAT} = \langle \Sigma_{PA}, PA \rangle$. Σ_{PA} contains the unary function s (successor) and the binary functions $+$ (sum) and $*$ (product). PA contains the usual Peano's axioms for $s, +, *$ (and all the instances of the first-order induction schema).

CONTEXT $\mathcal{NAT}$;
SIGNATURE:
Sorts: N;
Functions: $0 : [\,] \to N$;
$s : [N] \to N$;
$+, * : [N, N] \to N$;
AXIOMS: $\{s\}$: $\forall x : N \,.\, \neg s(x) = 0$;
$\forall x, y : N \,.\, s(x) = s(y) \to x = y$;
$\{+\}$: $\forall x : N \,.\, x + 0 = x$;
$\forall x, y : N \,.\, x + s(y) = s(x + y)$;
$\{*\}$: $\forall x : N \,.\, x * 0 = 0$;
$\forall x, y : N \,.\, x * s(y) = (x * y) + x$.

The standard structure of natural numbers is the intended model of $\mathcal{NAT}$.

Example 7.2 A simple example of an open context is the following, which axiomatises lists with generic elements X and a generic total ordering $\lhd$ on X.

CONTEXT $\mathcal{LIST}(X, \lhd : [X, X])$;
IMPORT: $\mathcal{NAT}$;
SIGNATURE:
Sorts : X, L;
Functions: $nil : [\,] \to L$;
$| : [X, L] \to L$;
$nocc : [X, L] \to N$;
Relations: $pos : [X, N, L]$;

AXIOMS:
$\{nil, |\}$: $\forall x, y, z : X,\ \forall j, k, l : L \,.$
$(\neg nil = x|j \wedge (y|k = z|l \to y = z \wedge k = l))$;
$\{nocc\}$: $\forall x : X \,.\, nocc(x, nil) = 0$;
$\forall x, y : X,\ \forall l : L \,.$
$x = y \to nocc(x, y.l) = nocc(x, l) + 1$;
$\forall x, y : X,\ \forall l : L \,.$
$\neg x = y \to nocc(x, y.l) = nocc(x, l)$;
$\{pos\}$: $\forall x : X,\ \forall l : L \,.$
$(pos(x, 0, l) \leftrightarrow \exists y : X,\ j : L \,.\, l = y|j \wedge x = y)$;
$\forall x : X,\ \forall l : L \,.$
$(pos(x, s(i), l) \leftrightarrow \exists y : X,\ j : L \,.$
$l = y|j \wedge pos(x, i, j))$.

The context (ADT) $\mathcal{NAT}$ is imported, together with its signature Σ_{PA} and axioms PA.

nil and $|$ are the *constructors* for the sort L of lists of elements of sort X. (For an element x and a list y, $x|y$ stands for the list with head x and tail y.) Their axioms are the list *constructor axioms* (plus structural induction).

$pos(x, i, l)$ means that the element x occurs at position i in the list l, where positions start from 0.

$nocc(x, l)$ is the number of occurrences of the element x in the list l.

7.2 Constraints

In an open context, some of the parameters in the signature may not be instantiated just anyhow. In fact their instantiation must be subject to strictly defined constraints.

Example 7.3 In the context $\mathcal{LIST}(X, \lhd : [X, X])$, in order to ensure that $\lhd$ is a total ordering, we have to add the following *constraints*:

CONTEXT $\mathcal{LIST}(X, \lhd : [X, X])$;
IMPORT: $\mathcal{NAT}$;
SIGNATURE:
Sorts : X, L;
Functions: ...
Relations: ...
AXIOMS: ...
CONSTRAINTS: $\forall x, y, z : X \,.\, (x \lhd y \wedge y \lhd x) \leftrightarrow x = y$;
$\forall x, y, z : X \,.\, (x \lhd y \wedge y \lhd z) \rightarrow x \lhd z$;
$\forall x, y, z : X \,.\, x \lhd y \vee y \lhd x$.

The purpose of constraints is to filter out illegal parameters of the context: only parameters that satisfy the constraints are allowed. For example, if in the context $\mathcal{LIST}($ $X, \lhd : [X, X])$, we want to substitute X by the sort N of natural numbers, and the ordering $\lhd$ by $\leq$ on N, then we can express this as a *closure* (or *instance*):

CLOSURE $\mathcal{NATLIST}$ OF $\mathcal{LIST}(X, \lhd : [X, X])$;
CLOSE: X BY N;
$\lhd$ BY $\forall x, y : N \,.\, (x \lhd y \leftrightarrow x \leq y)$.

This closure of the context $\mathcal{LIST}(X, \lhd : [X, X])$ satisfy the constraints of the context since $\leq$ is a total ordering on N.

In this example, we have closed X and $\lhd$ within $\mathcal{LIST}$ itself, for simplicity. In general, of course, they could also be closed within another context $\mathcal{C}$, after importing $\mathcal{LIST}$ into $\mathcal{C}$.

Obviously constraints define context dependencies.

8 Interface

The interface of a component is defined in the context of the component. The interface is the only part of the component that is visible to the users, and it should provide all the information that the users need in order to deploy the component. Since the interface is defined within the context, the latter should be regarded as part of the former. As we already made clear, the interface should contain specifications for the operations, and the context dependencies, of the component.

8.1 Operations

In the interface, operations are represented by their specifications. In a context $\langle \Sigma, X \rangle$, a specification of a new (relation) symbol r is a set of axioms that define r in terms of the symbols of the signature Σ. For example, suppose in $\mathcal{LIST}$ we have operations for sorting, such as *insertion sort* and *bubble sort*. The specification for these two operations are as follows:[1]

$\forall l : L \,.\, ord(l) \leftrightarrow$
$\forall i : N,\ \forall x, y : X \,.\, ((pos(x, i, l) \wedge pos(y, s(i), l)) \rightarrow x \lhd y)$
$\forall j, k : L \,.\, perm(j, k) \leftrightarrow \forall x : X \,.\, nocc(x, j) = nocc(x, k)$
$\forall j, k : L \,.\, sort(j, k) \leftrightarrow perm(j, k) \wedge ord(k)$
$\forall j, k, l : L \,.\, ord(j) \wedge ord(k) \rightarrow$
$(merge(j, k, l) \leftrightarrow ord(l) \wedge perm(j \| k, l))$.

We represent operations as logic programs. For example, the operations *insertion sort* and *bubble sort* are represented by the following logic programs:

Operation: insertionSort($merge$)		
$sort([\,], [\,])$	$\leftarrow$	
$sort(x.j, l)$	$\leftarrow$	$sort(j, k), merge([x], k, l)$

Operation: bubbleSort($\lhd$)		
$sort([\,], [\,])$	$\leftarrow$	
$sort(x.j, y.l)$	$\leftarrow$	$part(x.j, [y], k), sort(k, l)$
$part([\,], [\,], [\,])$	$\leftarrow$	
$part([x], [x], [\,])$	$\leftarrow$	
$part(x.j, [x], y.l)$	$\leftarrow$	$x \lhd y, part(j, [y], l)$
$part(x.j, [y], x.l)$	$\leftarrow$	$y \lhd x, part(j, [y], l)$

The operation insertionSort computes the relation $sort$ (as specified by the specification given above) in terms of the relation $merge$ (also as specified above). It therefore needs a program for $merge$ in order to complete the sorting operation. As a result insertionSort has $merge$ as a parameter, hence we write insertionSort($merge$). In any context that is a closure (instance) of $\mathcal{LIST}$, insertionSort will need a program for $merge$.

[1] For lists j and k, $j \| k$ stands for their concatenation.

Thus parameters to operations also define context dependencies.

By contrast, the operation bubbleSort has only the parameter $\lhd$, which is the parameter of the context. So bubbleSort will work for any context in which $\lhd$ is instantiated (closed) by any total ordering.

8.2 Context Dependencies

These consist of the (global) parameters in the signature of the component, the (local) parameters of the operations, together with the constraints in the context.

So now we can define the context dependencies completely in a component.

9 Code

The code should be inaccessible (invisible) to the user. It is usually binary. However, if we allow parameters in the operations, then the code has to be source code, which has to be instantiated before execution.

If the source code is available, then the user or the developer can also verify its correctness with respect to the specifications in the context.

10 A New Notion of A Priori Correctness

In our work the basis for a priori reasoning is a new notion of a priori correctness. So having laid out the specification of a component, we now turn to our definition of a priori correctness of a component. Specifically, we consider a notion of a priori correctness of the operations in a component, that we call *steadfastness*.

10.1 Steadfastness

A *steadfast* operation (program) Op is one that is correct (wrt to its specification) in each intended model of the context $\mathcal{C}$ of the component. Since the (reducts of the) intended models of its specialisations and instances are intended models of $\mathcal{C}$, a steadfast program Op is correct, and hence correctly reusable, in all specialisations and instances of $\mathcal{C}$.

A formalisation of steadfastness is given in [8], with both a model-theoretic, hence *declarative*, characterisation and a proof-theoretic treatment of steadfastness. Here we give a simple example (based on an example in [8]) to illustrate the intuition behind steadfastness.

Example 10.1 Consider the following component:
where the open context $\mathcal{ITER}\ (D, \circ, e)$ is defined as follows:

Iterate
CONTEXT $\mathcal{ITER}(D, \circ, e)$
INTERFACE Operations: $S_{iterate}, S_{unit}, S_{op}$; iterate($unit$, op); Dependencies: $D, \circ, e, unit, op$;
CODE Code for iterate

Figure 6. The Iterate component.

CONTEXT $\mathcal{ITER}(D, \circ, e)$;
IMPORT: $\mathcal{NAT}$;
SIGNATURE:
Sorts: D;
Functions: $e : [\,] \to D$;
$\circ : [D, D] \to D$;
$\times : [D, N] \to D$;
AXIOMS: $\forall x : D\,.\ \times(x, 0) = e$;
$\forall x : D,\ \forall n : N\,.\ \times(x, s(n)) = \times(x, n) \circ x$.

where $\mathcal{NAT}$ is the closed context for first-order Peano arithmetic defined in Example 7.1.

In the open context $\mathcal{ITER}(D, \circ, e)$:

(i) D is a (generic) domain, with a binary operation $\circ$ and a distinguished element e (see the first axiom);

(ii) the usual structure of natural numbers is imported;

(iii) the function symbol $\times$ represents the iteration operation $\times(a, n) = e \circ a \underbrace{\circ \cdots \circ}_{(n \text{ times})} a$
(see the second axiom).

We can use the Iterate component to iterate n times the binary operation $\circ$ on some (generic) domain D.

Suppose in Iterate, or more precisely its context $\mathcal{ITER}$, we specify the $iterate$ operation by the following relation:

$$S_{iterate}: \qquad iterate(a, n, z) \leftrightarrow z = \times(a, n) \qquad (1)$$

The predicate $iterate(x, n, z)$ means that z is the result of applying the iteration operation $\times$ to (a, n), i.e. $z = \times(a, n) = e \circ a \underbrace{\circ \cdots \circ}_{(n \text{ times})} a$.

This specification of $iterate$ can be implemented by the operation iterate($unit$, op) defined by the following logic program:

Operation: iterate($unit$, op)		
$iterate(a, 0, v)$	$\leftarrow$	$unit(v)$
$iterate(a, s(n), v)$	$\leftarrow$	$iterate(a, n, w)$, $op(w, a, v)$

where s is the successor function for natural numbers, and the relations $unit$ and op are specified in $\mathcal{ITER}$ by the

specifications:

$$\begin{array}{lcll} S_{unit} & : & unit(u) & \leftrightarrow \quad u = e \\ S_{op} & : & op(x,y,z) & \leftrightarrow \quad z = x \circ y \end{array} \qquad (2)$$

The predicate $unit(u)$ means u is the distinguished element e, and $op(x,y,z)$ means that z is the result of applying the operation $\circ$ just once to x and y. Therefore in the program for iterate, if $unit(v)$, i.e. v is just e, then $iterate(a,0,v)$ computes $\times(a,0) = v = e$. Otherwise, if $iterate(a,n,w)$, i.e. $w = \times(a,n)$, and $op(w,a,v)$, i.e. $v = w \circ a$, then $iterate(a,s(n),v)$ computes $\times(a,s(n)) = v = w \circ a = \times(a,n) \circ a = e \circ a \underbrace{\circ \cdots \circ}_{(n+1 \text{ times})} a$.

The operation iterate($unit$, op) is defined in terms of the parameters $unit$ and op. If we can assume that operations for $unit$ and op are a priori correct, i.e. they are correct wrt their specifications (2) in *any* interpretation of $\mathcal{ITER}$, then we can prove that the operation iterate($unit$, op) is *steadfast*, i.e. it is always correct wrt (1) (and (2)).

For example, suppose we have a component Naturals as shown in Figure 7, in which the context is $\mathcal{NAT}$, and the

Naturals
CONTEXT $\mathcal{NAT}$
INTERFACE Operations: S_{unit}, S_{op}; unit, op;
CODE Code for unit, op

Figure 7. The Naturals component.

operations unit and op are specified as follows:

$$\begin{array}{lcll} S_{unit} & : & unit(u) & \leftrightarrow \quad u = 0 \\ S_{op} & : & op(x,y,z) & \leftrightarrow \quad z = x + y \end{array} \qquad (3)$$

(i.e. $unit(u)$ means u is 0, and $op(x,y,z)$ means $z = x + y$) and defined as follows:

Operation:- unit
$unit(0)$.

Operation:- op
$op(x,y,z) \quad \leftarrow \quad z = x + y$

Then in Naturals, unit and op are (trivially) a priori correct wrt to their specifications (3), and if we compose the components Iterate and Naturals, the operation iterate in the composite Iterate+Naturals will be fully instantiated (and therefore executable), and more importantly it will be correct wrt its specification (1) (and (2)).

The composition here is of course just the simple closure operation on Iterate, but it is sufficient to illustrate the idea of steadfastness. In this closure of Iterate:

(i) D is the set of natural numbers;

(ii) $\circ$ is $+$;

(iii) e is 0;

(iv) $\times(a,n) \leftrightarrow 0 + a = na$.

Consequently, the specification $S_{iterate}$ (1) specialises to

$$iterate(x,n,z) \leftrightarrow z = na$$

and similarly S_{unit} (in (2)) specialises to

$$unit(u) \leftrightarrow u = 0$$

(in (3)), and S_{op} (in (2)) to

$$op(x,y,z) \leftrightarrow y\, z = x$$

(in (3)). Since, the operations unit and op are correct with respect to their (specialised) specifications (3), the operation (iterate($unit$, op) unit $\cup$ op) will compute na, and is correct wrt its (specialised) specification in Iterate+Naturals.

To illustrate the correct reusability of the iterate operation in Iterate, suppose now we have a component Integers

Integers
CONTEXT $\mathcal{INT}$
INTERFACE Operations: S_{unit}, S_{op}; unit, op;
CODE Code for unit, op

Figure 8. The Integers component.

as shown in Figure 8, where the operations unit and op are specified by:

$$\begin{array}{lcll} S_{unit} & : & unit(u) & \leftrightarrow \quad u = 0 \\ S_{op} & : & op(x,y,z) & \leftrightarrow \quad z = x - y \end{array} \qquad (4)$$

and defined by:

Operation:- unit
$unit(0)$.

Operation:- op
$op(x,y,z) \quad \leftarrow \quad z = x - y$

Obviously the operations unit and op in Integers are a priori correct wrt their specifications (4). We can compose Iterate and Integers by a closure operation on Iterate, and get a correct iterate operation in the composite Iterate+Integers.

In Iterate+Integers:

(i) D is the set of integers;

(ii) $\circ$ is $-$;

(iii) e is 0;

(iv) $\times(a, n) = 0 \quad a - - \quad a = -a$,

and the specification $S_{iterate}$ ((1) in Iterate) specialises to

$$iterate(x, n, z) \text{ and } z =$$

S_{unit} (in (2)) specialises to

$$unit(u) \leftrightarrow u = 0$$

(in (4)), and S_{op} (in (2)) to

$$op(x, y, z) \leftrightarrow z = x \quad y$$

(in (4)). Since unit and op are correct wrt their specifications (4), the operation (iterate$(unit, op)$ unit$\cup$op) computes $-na$ for an integer a, and is correct wrt its (specialised) specification in Iterate+Integers.

The iterate operation is thus a priori correct in Iterate and we say it is *steadfast*. It can be correctly reused in any composite with operations for unit and op as long as these operations are in turn steadfast.

The component Iterate has no constraints in its context dependencies. To further illustrate the notion of steadfastness, we now consider a component whose context dependencies include constraints.

Example 10.2 Consider the component Iterate* (Figure 9) obtained from Iterate (Figure 6) by adding the following

Iterate*
CONTEXT $\mathcal{ITER}(D, \circ, e)$
INTERFACE Operations: $S_{iterate}, S_{unit}, S_{op}$; iterate*$(unit, op)$; Dependencies: $D, \circ, e, unit, op$; constraints;
CODE Code for iterate*

Figure 9. The Iterate component.

constraints to its context dependencies:

$$\begin{array}{l} \forall x : De \circ x = \quad x \\ \forall x, y, z \ : Dx \ y \ (\circ \ z) = (x \circ \end{array} \tag{5}$$

(these constraints stipulate that $\circ$ should be associative) and by replacing the iterate operation in Iterate by the following operation iterate*:

Operation: iterate*$(unit, op)$		
$iterate(a, 0, v)$	$\leftarrow$	$unit(v)$
$iterate(a, n, v)$	$\leftarrow$	$m + m = n, iterate(a, m, w),$ $op(w, w, v)$
$iterate(a, n, v)$	$\leftarrow$	$m + m + 1 = n, iterate(a, m, w),$ $op(w, w, z), op(z, a, v)$

The operation iterate* has the same specification $S_{iterate}$ (1) as iterate in Iterate, but it computes the relation $iterate$ more efficiently than iterate: the number of recursive calls is linear in iterate, whereas it is logarithmic in iterate*. However, iterate* would *not* be steadfast in Iterate. For example, if we were to use iterate* in place of iterate in Iterate, then iterate* would be correct wrt (1) and (2) in Iterate+Naturals, but it would not be correct wrt (1) and (2) in Iterate+Integers, where, for instance, for

$$iterate(a, s(s(s(s(0))))), v)$$

iterate* would compute 0 instead of the correct answer $-4a$. Thus despite the a priori correctness of unit and op in both Naturals and Integers, iterate* would not be correct in both Iterate+Naturals and Iterate+Integers. Therefore iterate* would *not* be steadfast in Iterate.

The reason for this is that in Iterate*, the constraints (5) require that the parameters e and $\circ$ of the context satisfy the unit and associativity axioms. These imply that

$$\begin{cases} \times(a, n) = \times(a, n \div 2) + \times(a, n \div 2) + a & \text{if } n \text{ is odd} \\ \times(a, n) = \times(a, n \div 2) + \times(a, n \div 2) & \text{if } n \text{ is even} \end{cases}$$

which means that whenever $\circ$ is associative, $\times$ can be computed in logarithmic time. So, if we were to use iterate* in place of iterate in Iterate, then iterate* would be correct in Iterate+Naturals because here (D is the set of natural numbers) e is 0, $\circ$ is $+$, and so they actually satisfy the constraints (5) anyway, even though these constraints are not present in Iterate. On the other hand, iterate* would not be correct in Iterate+Integers because here (D is the set of integers) e is 0, $\circ$ is $-$, and since $-$ is not associative, they do not satisfy (5).

However, we can prove that iterate* is steadfast in Iterate*, again assuming a priori correctness of operations for unit and op defined in some other component. It will be correct in any composite Iterate*+C as long as C satisfies the constraints (5) in the context dependencies of Iterate*. For example, as can be seen from the above discussion, iterate* will be correct in Iterate*+Naturals since $+$ is associative.

For something completely different, suppose Matrices is a component with a context of m-dimensional square matrices. Then in the composite Iterate*+Matrices, D is the set of m-dimensional square matrices, e is the m-dimensional identity matrix, and since matrix multiplication $\times$ is associative, iterate* will be correct, where op computes matrix products.

11 Discussion

Since a steadfast program is correct, and hence correctly reusable, in all specialisations and instances of its context, a component with steadfast operations, which we will call

a *steadfast component*, when composed with another steadfast component will also be steadfast. In other words, steadfastness is not only *compositional*, but is also preserved through *inheritance* hierarchies.

Consequently, in the context of system prediction (as shown in Figure 3) when composing steadfast components, not only can we be sure that the composite will be steadfast, but we can also predict the specification of the composite. This is illustrated in Figure 10.

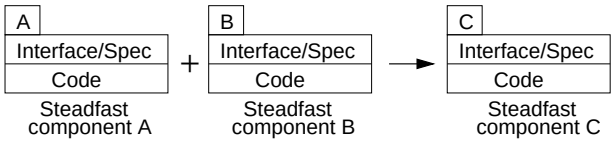


Figure 10. Composing steadfast components.

In the context of modular specification and verification (as shown in Figure 4), steadfast modules can be verified and the specification of the composite can be predicted, prior to composition. This is illustrated in Figure 11. We understand that current approaches to modular reasoning need to know the specification of the composite before predicting if the composite will work according to its specification. If this is the case (as shown in Figure 4), then steadfastness offers the advantage of being able to predict the specification of the composite prior to composition. Thus, with steadfast modules, we can do system prediction as shown in Figure 10.

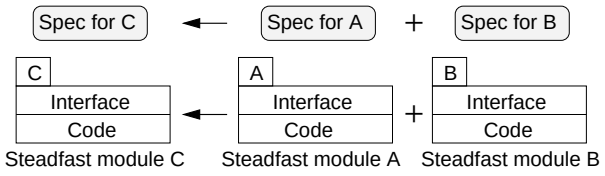


Figure 11. Composing steadfast modules.

12. Conclusion

For lack of space, we have presented the intuition behind steadfastness by means of simple examples. We hope this does not detract from its presentation.A full account of steadfastness can be found in [8]. Steadfastness is defined in terms of model-theoretic semantics. It is thus declarative in nature. We believe that declarative semantics in general will be important for lifting the level of abstraction.

Our approach to specifying components is very generic. The component may be just a class or ADT. It may be a module, in particular what Meyer [10] calls an *abstracted module*, which is the basic unit of reuse in the CBD methodology RESOLVE [14]. It may be an object model [2] as in OMT [11] or UML [12]. It may yet be an OOD framework, i.e. a group of interacting objects [6], such as frameworks in the CBD methodology *Catalysis* [3, 5]. It could even be a design pattern or schema [4].

We believe that our approach to component specification can enable predictable component assembly, which is currently an open problem in CBD. In addition, we believe it can provide a hybrid, spiral approach to CBD [7] that is both top-down and bottom-up for CBD, as illustrated in Figure 12. First a library of steadfast components has to be built. The nature of steadfastness, coupled with the use of *a priori reasoning*, then allows these components to be composed into larger systems in either a top-down (following the traditional *waterfall model* or the *software architecture* approach [13, 1]), or bottom-up manner, or indeed a combination of both.

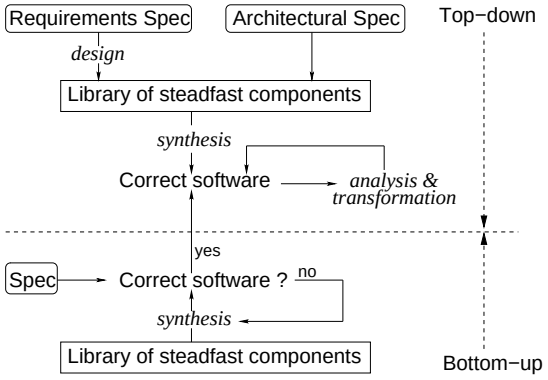


Figure 12. A spiral model for CBD.

Bottom-up development in particular is more in keeping with the spirit of CBD. Composition of steadfast components can show the specification of the composite, and therefore the specification of any software constructed can be compared with the initial specification for the whole system. Guidance as to which components to 'pick and mix' can also be provided by component specifications.

Acknowledgements

We are grateful to Murali Sitaraman and the reviewers for their helpful comments in general, and for correcting our misunderstanding of modular specification and verification in particular.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[2] R. Bourdeau and B. H. Cheng. A formal semantics for object model diagrams. *IEEE Trans. Soft. Eng.*, 21(10):799–821, 1995.

[3] D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.

[4] P. Flener, K.-K. Lau, M. Ornaghi, and J. Richardson. An abstract formalisation of correct schemas for program synthesis. *Journal of Symbolic Computation*, 30(1):93–127, July 2000.

[5] J. Küster Filipe, K.-K. Lau, M. Ornaghi, K. Taguchi, A. Wills, and H. Yatsu. Formal specification of Catalysis frameworks. In J. Dong, J. He, and M. Purvis, editors, *Proc. 7th Asia-Pacific Software Engineering Conference*, pages 180–187. IEEE Computer Society Press, 2000.

[6] J. Küster Filipe, K.-K. Lau, M. Ornaghi, and H. Yatsu. On dynamic aspects of OOD frameworks in component-based software development in computational logic. In A. Bossi, editor, *Proc. LOPSTR 99, Lecture Notes in Computer Science*, volume 1817, pages 43–62. Springer-Verlag, 2000.

[7] K.-K. Lau. Component certification and system prediction: Is there a role for formality? In I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau, editors, *Proceedings of the Fourth ICSE Workshop on Component-based Software Engineering*, pages 80–83. IEEE Computer Society Press, 2001.

[8] K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. Logic Programming*, 38(3):259–294, March 1999.

[9] G. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, pages 72–80, July 1991.

[10] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, second edition, 1997.

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Sorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[12] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[13] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[14] M. Sitaraman and B. Weide, editors. *Component-based software using RESOLVE*. Special feature, ACM Sigsoft Software Engineering Notes 19(4): 21-65, October 1994.

A Pi-Calculus based Framework for the Composition and Replacement of Components

Claus Pahl
Dublin City University, School of Computer Applications
Dublin 9, Ireland
cpahl@compapp.dcu.ie

Abstract

Evolution in component systems is critical with respect to the maintainability of these systems. Systems evolve due to changes in the environment or due to improvements of individual components. Even though component technology aims at reducing the dependencies between components, necessary replacements of components might affect the composition of systems. We introduce a composition and replacement calculus based on the π-calculus, allowing us to specify composition and to reason about replacements and their effects.

1. Introduction

A formal semantics for components and component composition is essential if rigorous analysis and reasoning in the development and maintenance of component systems shall be deployed. Our objectives are a formalisation of basic composition principles, similar to [1, 2, 3], and the provision of a framework for change and evolution analysis. The motivation to use the π-calculus lies in a similarity between the notions of *mobility* in the π-calculus and *evolution* in component technology. Mobility is defined as the capacity to change the connectivity of a network, i.e. to change the spatial configuration. Evolution in large component system is also about the change of connections between components. Therefore, the π-calculus seems to be a suitable formal notation to develop a framework for specification and reasoning about component composition and, in particular, evolution in these systems. As we will see later on, we will propose some change to basic π-calculus semantics and develop a type system reflecting component technology principles. The result is a mixed calculus, based on π-calculus basics and concepts from component technology. The type system plays the role of the integrator. Types govern how names can be used in process calculi [4]. They classify patterns of behaviour, and they can also reflect connectivity and the control mobility and evolution.

We essentially define a composition protocol with different phases: matching and connector establishment, invocation and execution of the service, an invocation reply, and, later on, dynamic replacements. The different phases will be described by different transition rules. We argue that a process-oriented look at component composition is of major importance for the reliability and maintainability of evolving systems. We will complement this process view on compositions with methods to reason about replacements of components in changing environments. The evolution of systems is often neglected in formal approaches to software engineering problems. Some papers have addressed dynamic reconfiguration and replacement, see for example [5] or [6], but most of these papers have addressed the problem from a pragmatic or not fully rigorous point of view. We devise a consequent approach in this direction, overcoming these deficiencies, by formulating a formal framework of change and reconfiguration for component composition.

Section 2 illustrates component composition. In Section 3 we introduce some basics of our composition and replacement calculus. Section 4 focusses on contract-based matching and connector establishment. The type system is introduced in Section 5. Section 6 specifies the life cycle of component composition, without looking at replacements. Replacements and how to reason about their effect is the subject of Section 7. Finally, we discuss related work and end with some conclusions.

2. Components and Composition

Our component model is based on port-based components, where ports represent services, with export and import interfaces for provided and requested services. Two component interfaces are presented below - `Interface` requesting services and `DocServer` providing services - both part of a document manipulation and storage system.

Component `Interface`
import services
`servReqDoc(uri:URI):Doc`
`servModDoc(doc:Doc,upd:Txt)`
export services
`openDoc(uri:URI)`
`saveDoc(uri:URI, upd:Txt)`

The interfaces uses services from the server component to request (load) and modify (store) documents.

Component `DocServer`
import services
...
export services
`reqDoc(uri:URI):Doc`
`modDoc(doc:Doc,upd:Txt)`

Documents are identified by URIs – uniform resource identifiers. The request service `reqDoc` returns a document, but does not change the state of the server component, whereas the modification service `modDoc` updates a document on the servicer side without returning a value.

Service ports are points of access described in component interfaces. We assume these ports to be specified in some kind of logic that allows to express pre- and postconditions as abstractions for ports [7, 8], enabling the design-by-contract approach [8, 9]. Hoare logic or modal logics are suitable frameworks [10, 11]. A requirements specification of the service user `servModDoc` could look like:

```
servModDoc ( myDoc:Doc, myUpd:Txt )
     pre   valid()
     post  updated()
```

Documents shall be XML-documents here, which can be well-formed (correct tag nesting) or valid (well-formed and conform to a document type definition DTD). A service provider specification could look like:

```
modDoc ( doc:Doc, upd:Txt )
     pre   wellFormed()
     post  updated() ∧ acknowledged()
```

A contract can be formed between interface and document server. The service `modDoc` of the document server matches the requirements of `servModDoc` - a service that might be called in methods provided by the interface. `modDoc` has a weaker, less restricted precondition - `valid()` implies `wellFormed()` - and a stronger postcondition - the conjunction `updated()` ∧ `acknowledged()` implies `updated()`. This means that the provided service satisfied the requirements; it is even better than requested.

We will neglect a detailled presentation of component semantics in this presentation; our focus is on the *composition semantics*. The composition of components [12, 13] will be defined using the π-calculus [14, 15, 16]. Interaction is the composition principle. Ports represent services of a component. We will distinguish different roles of ports, e.g. whether a service is provided or requested. The semantics of service execution is reflected in the type system through pre- and postconditions. The type system is the link between the component and the composition semantics. However, the pre- and postconditions, forming a contract, are also important for the composition process. A conformance condition expresses that two ports match based on whether a provided service satisfies the needs of a requested service. Technically, the matching process is facilitated through contract ports $\texttt{servModDoc}_C$ and $\texttt{modDoc}_C$. The user interface requires a service (annotation REQ) and the document server provides a service (annotation PRO).

$$\texttt{Interface} \stackrel{\text{def}}{=} \text{REQ } \overline{\texttt{servModDoc}_C}\langle\texttt{servModDoc}_I\rangle.\texttt{Interface}'$$
$$\texttt{DocServer} \stackrel{\text{def}}{=} \text{PRO } \texttt{modDoc}_C(\texttt{modDoc}_I).\texttt{DocServer}'$$

The names are not relevant for components to match - only the pre- and postconditions represented through types are. Successful matching results in the establishment of a connector - a private channel between the components that allows one component to use services provided by the other.

$$\texttt{Interface}' \stackrel{\text{def}}{=} \text{INV } \overline{\texttt{servModDoc}_I}\langle\texttt{doc}, \texttt{upd}\rangle.\texttt{Interface}''$$
$$\texttt{DocServer}' \stackrel{\text{def}}{=} \text{EXE } \overline{\texttt{modDoc}_I}\langle\texttt{x}_1, \texttt{x}_2\rangle.\texttt{DocServer}''$$

The user interface can invoke the service (INV) through the interaction port $\texttt{servModDoc}_I$, which will trigger the execution (EXE) of $\texttt{modDoc}_I$ by the server. The composition, i.e. establishment of a connector, is one of the key activities; the other is replacement. Systems evolve over time, components are replaced by improved or modified versions. Methods to answer whether for example the user interface or the document server can be replaced shall be introduced.

3. Calculus - Syntax and Type Basics

This section shall introduce basics of our calculus, such as syntax of the notation and some type issues, before we look at rules for component composition.

3.1 Syntax of Component Composition

The basic element describing activity in the π-calculus are actions. Actions are combined to process expressions. Actions are expressed as prefixes to these process expressions:

$$\pi ::= \text{PTYPE } \overline{x}\langle y\rangle \mid \text{PTYPE } x(y) \mid \tau$$

Actions can be divided into output actions $\overline{x}\langle y\rangle$ - the name y is sent along channel (or port) x -, input $x(y)$ - y is received along x, and a silent non-observable action τ. We have annotated these action prefixes by port types, which will explain the role of the port with respect to life cycle activities such as service request or service invocation. Here is the full list of action prefixes, their port types and polarities (types and polarities will be explained soon):

π	::=	REQ $\overline{m_C}\langle m_I\rangle$	+	Request
		PRO $n_C(n_I)$	−	Provide
		INV $\overline{m_I}\langle a_1, \ldots a_l, m_R\rangle$	+	Invoke
		EXE $n_I(x_1, \ldots, x_k, n_R)$	−	Execute
		REP $\overline{n_R}\langle b\rangle$	+	Reply
		RES $m_R(y)$	−	Result

The syntax of composition expressions involving the action prefixes is the following:

P	::=	$\nu m\ P$	Restriction
		$P_1 \mid P_2$	Parallel composition
		$!P$	Iteration
		$\Sigma_{i\in I}\ \pi_i.P_i$	Summation
		0	Inaction

Restriction means that m is only visible in P. Summation $\pi_i.P_i$ means that one π_i is chosen and the process tranfers to state P_i. Iteration $!P$ means that the process is executed an arbitray number of times. This follows the presentation of the π-calculus in [15]. We also need abstractions, i.e. defining equations of the form $A(a) = P_A$. Even though the polyadic π-calculus is intended to be used, we often use the monadic variant here in order to keep the notation simple. The substitution $\{b/a\}$P means that b replaces a in P.

3.2 Ports and their Types

The entities in our composition system are values, ports and components. Values are characterised by the usual value domains as types. The list of basic types $t_1, t_2, \ldots$ shall be assumed, but not explicitly specified. Components are syntactically characterised by an interface with service signatures, separated into import and export elements.

The most important entities are the ports. Each port p is essentially a family of ports $p = (p_C, p_I, p_R)$. The first port p_C is the *contract port*, essentially an abstract interface described by a signature, a precondition and a postcondition. p_I is the connector activation (or interaction) port. This port is used to invoke a service. The port p_R carried

m_C : $\textsc{Ctr}(\textsc{Sig}(T_1, \ldots, T_n, +\textsc{Cre}(T)), \textsc{Prd}(\textsc{pre}), \textsc{Prd}(\textsc{post}))$
n_C : $\textsc{Ctr}(\textsc{Sig}(T'_1, \ldots, T'_n, \textsc{Cre}(T')), \textsc{Prd}(\textsc{pre'}), \textsc{Prd}(\textsc{post'}))$
m_I : $\textsc{Cac}(T_1, \ldots, T_n, \textsc{Cre}(T))$
n_I : $\textsc{Cac}(T_1, \ldots, T_n, \textsc{Cre}(T))$
m_R : $\textsc{Cre}(T)$
n_R : $\textsc{Cre}(T)$

Figure 1: Ports and their channel types.

the reply from the service invocation. We distinguish a port type and a channel type for each port.

Port types describe the functionality of the port within the component (e.g. contract or connector ports) and its orientation (in- and out-ports). Port types $\mathcal{T}_p(p)$ or $p :_p t$ for port p are denoted by 3-letter lower case abbreveations. The annotations in the action prefix syntax (see Section 3.1) denote port types, e.g. $\overline{m_C}$ is a request port; n_C is the dual provide-port: $\mathcal{T}_p(\overline{m_C}) = \textsc{Req}$ and $\mathcal{T}_p(n_C) = \textsc{Pro}$. Each port has also an orientation - called the *polarity*; all ports of one port family follow always the same orientation pattern: $+, +, -$ for requested (imported) services, saying that contract and connector ports are output ports ('+') and the reply port is an input port ('−'), and $-, -, +$ for provided (exported) services. Each '+' stands for an output capability (the port can only send); '−' stands for an input capability (the port can only receive).

Channel types describe what kind of entities can be transported: a contract port $p_C :_c \textsc{Ctr}(\textsc{sig}, \textsc{pre}, \textsc{post})$, a connector activation or interaction port $p_I :_c \textsc{Cac}(T_1, \ldots, T_n, \textsc{Cre}(T))$, and a connector reply port $p_R :_c \textsc{Cre}(T)$. This characterises the channel by specifying the expected capacity - what data can be transported. It will constrain the composition and interaction between components. Contract ports can transport connectors, which are characterised by a contract type. Connectors provide the connection between components to invoke a service. Channel types $\mathcal{T}_c(p)$ or $p :_c t$ for port p are denoted by 3-letter abbreveations starting with an upper case character, see Figure 1.

A contract consists of a service signature, a pre- and a postcondition. Connectors when transfered on channels have to satisfy a contract type. On connector activation ports, data values and a reply channel can be transfered; on connector reply ports, data can be transfered. The key criterion for matching, i.e. the succesful connection of two components through a connector, are are contracts (this will be explained in the next subsection). Opposite orientations also have to match in a successful composition of component ports. The signature for a remote method execution is: $\textsc{Sig}(T_1, \ldots, T_n, \textsc{Cre}(T))$. This is an adequate representation, reflecting the fact that parameters are passed, and possibly a result has to be transfered back on a channel with a different capacity (type). For local method executions, the usual notation $T_1 \times \ldots \times T_n \to T$ apply. Pre- and postconditions are formed using the predicate type constructor $\textsc{Prd}$.

Connector ports represent services. Connectors are channels that can carry data elements - for the connector activation additionally a reply channel. Connector ports and connector reply ports are only used as restricted (private) channels between components that match based on contracts. This means that these channels are only available to these two components.

3.3 Subtypes

In principle, the definition of a subtype relation is possible for all kinds of entities in our notation. However, we will focus on ports here. The subtype relation will help us to determine whether two ports, representing services, match and whether they can be composed. Later on, this concept will also be used to determine consistent (effect-free) replacements.

We have already seen that the channel types of contract ports are contracts consisting of a service signature, a precondition and a postcondition. For a required service $m_C :_c \textsc{Ctr}(\textsc{sig},\textsc{pre},\textsc{post})$ and a provided service $n_C :_c \textsc{Ctr}(\textsc{sig'},\textsc{pre'},\textsc{post'})$, we say that n_C matches m_C if

$$\textsc{sig} = \textsc{sig'} \wedge \textsc{pre} \to \textsc{pre'} \wedge \textsc{post'} \to \textsc{post}$$

This is the combination of two classical refinement relations (weaken the precondition and strengthen the postcondition) from the Refinement Calculus [17, 18], see also [19] for other matching approaches.

4. Contract Matching and Connectors

We define the operational semantics of component composition in this section. m_C, n_C are contract ports, m_I, n_I are connector activation or interaction ports, and m_R, n_R are connector reply ports. (m_C, m_I, m_R) is an output port, i.e. actively requests services from other components, and (n_C, n_I, n_R) is an input port, providing services to other components. Ports are typed. The connector ports (activation and reply) are restricted to components matched based on contracts. We will introduce a number of transition rules describing state changes, including activities such as contract matching, connector establishment, interaction and interaction reply. The rules will define the *transition semantics*. They will replace some of the transition rules for the classical π-calculus, in particular the reaction rules - see e.g. [16] Table 1.5. Other rules are still valid.

4.1 Contracts

Two components can react, i.e. can be connected, if their contract types (end points of possible channels) form a subtype relationship. This changes the original π-calculus reaction rule which requires channel names to be the same. Here we only require a subtype relationship between the ports. The receiver can *accept* an input based on the type, not the name. We assume the following channel types $t_{m_C} = \textsc{Ctr}(\textsc{sig}, \textsc{pre}, \textsc{post})$ and $t_{n_C} = \textsc{Ctr}(\textsc{sig'}, \textsc{pre'}, \textsc{post'})$ for contract ports m_C and n_C, respectively. The contract type t_{n_C} is a subtype of t_{m_C}, if the precondition is weakened and the postcondition is strengthened. Signatures are assumed to be equal. The following transition rule – the **contract rule** [T-Ctr] – describes matching of services:

$$\frac{\textsc{Req}\ \overline{m_C}\langle m_I\rangle.C \xrightarrow{\overline{m_C}\langle m_I\rangle} C \qquad \textsc{Pro}\ n_C(n_I).P \xrightarrow{n_C(n_I)} P}{\textsc{Req}\ \overline{m_C}\langle m_I\rangle.C | \textsc{Pro}\ n_C(n_I).P \xrightarrow{\tau} C \frown P} \langle\ \Phi$$

where side condition is $\Phi \equiv t_{n_C} \leq t_{m_C}$. The connection $C \frown P \stackrel{\text{def}}{=} \nu c(\{c/m_I\}C | \{c/n_I\}P)$ introduces a fresh variable c - free in C and P - creating a private (restricted) channel c called the connector. This rule expresses the connector

establishment. Ultimately, we will chain together several components that will import services from others.

A second typing constraint is hidden. REQ and PRO denote port types, i.e. $\overline{m_C} :_p \textsc{Req}$ and $n_C :_p \textsc{Pro}$. These are type annotations to the ports. Here, the port types match: REQ is the complement of PRO, and the polarities are opposite. We write $\mathcal{T}(\overline{m_C}) \simeq \mathcal{T}(n_C)$ in this case.

Type systems for the π-calculus usually constrain data that is sent, here we constraint reaction (the interaction between agents). The contract rule cannot be translated to the match-rule found in some π-calculus variants. Our contract rule is similar to transition rules describing reaction that are based on bounded output $\overline{x}(z)$ where z is introduced as a bound variable forming a restricted channel [16]. We have chosen to introduce a fresh variable c instead.

4.2 Connectors

We assume that a private channel c - the connector - has been established between client and provider. This channel is used by the client to invoke a service n_I at the server side. Parameter data $a : t_a$ with $t_a \leq t_x$ and a reply channel $m_R : t_{m_R}$ are send to the provider. The **connector activation rule** [T-CAC] is defined as follows:

$$\frac{\textsc{Inv}\ \overline{m_I}\langle a, m_R\rangle.C \xrightarrow{\overline{m_I}\langle a,m_R\rangle} C \quad \textsc{Exe}\ n_I(x, n_R).P \xrightarrow{n_I(x,n_R)} P}{\textsc{Inv}\ \overline{m_I}\langle a, m_R\rangle.C|\textsc{Exe}\ n_I(x, n_R).P \xrightarrow{\tau} C \frown \{a/x\}P}\langle\ \Phi$$

where $\Phi \equiv t_{n_I} \leq t_{m_I}, a : pre$. The types t_{m_I} and t_{n_I} are the connector activation types $\textsc{CAc}(t_1, \ldots, t_m, \textsc{CRe}(t))$ and $\textsc{CAc}(t'_1, \ldots, t'_n, \textsc{CRe}(t'))$, respectively. The reply channel is again a private channel between the two components that replaces m_R and n_R. Type equality (or a subtype relation) for m_I and n_I is not required if we can guarantee that the connector types satisfy the contract types and that the contract matching has successfully been executed. A protocol - specified in form of a component life cycle - can guarantee this. We will discuss the side condition $a : pre$ shortly.

The last rules is the **connector reply rule** [T-CRE]:

$$\frac{\textsc{Res}\ m_R(y).C \xrightarrow{m_R(y)} C \quad \textsc{Rep}\ \overline{n_R}\langle b\rangle.P \xrightarrow{\overline{n_R}\langle b\rangle} P}{\textsc{Res}\ m_R(y).C|\textsc{Rep}\ \overline{n_R}\langle b\rangle.P \xrightarrow{\tau} \{b/y\}C \frown P}\langle\ \Phi$$

where $\Phi \equiv t_{n_R} \leq t_{m_R}, b : post$. We assume $t_b \leq t_y$. b is the result of the internal computation, i.e. b is a function of x.

The contract with pre- and postconditions can be reflected at the connector level. We associate pre- and postconditions with the in- and out-ports. Input a is required to satisfy the precondition pre and output b is required to satisfy the postcondition $post$. These assertions are obligations, formalised by contracts, to be satisfied by client (pre) and provider (post) at runtime. This attachment of obligations to the connectors results in more symmetry and links contracts and connectors. Pre- and postconditions are formulas, but here they are evaluted at runtime when the corresponding method is invoked and executed.

5. Types and Subtypes

We use the type system to control the correct establishment, use and replacement of connections between components. Especially subtypes are important for this purpose.

We use typing rules to describe our type system. Syntactial aspects of our notation have been dealt with in previous sections. We will address the relation between the type system and the transition semantics. The type safety property guarantees that well-typed expressions (expressions whose types can be infered using the type system) do not fail under transition. We show that the well-typedness is preserved.

T	::=	B	Basic type
	|	L	Link type
	|	$\textsc{Sig}(T \times \ldots \times T \times L)$	Signature
	|	$\textsc{Prd}(T)$	Predicate
L	::=	$P\ C$	Port and channel type
P	::=	$\pm$ (REQ | PRO | INV | EXE | REC | REP)	Port type
C	::=	$\textsc{Ctr}(T \times T \times T)$	Contract
	|	$\textsc{CAc}(T \times \ldots \times T \times L)$	Connector activation
	|	$\textsc{CRe}(T)$	Connector reply

Figure 2: The syntax of the type language.

5.1 Typing Rules

A typing context Γ is a finite set of bindings - mappings from names to types. Three types of judgments are used:

$\Gamma \vdash x : T$	name x has type T
$\Gamma \vdash S \leq T$	type S is subtype of T
$\Gamma \vdash P$	expression P is well-typed

The type language syntax is defined in Figure 2. The constructors CTR, CAC, CRE are the link-type constructors. Their purpose is to classify channels based on the data that is transfered along them. We leave the set of value types unspecified. We assume that there is at least one basic type. SIG and PRD are standard constructors for service signatures and predicates, the other type constructors are application-specific to the component context.

The semantics of the type system will be defined by typing rules for basic types, type constructors, subtypes and process expressions. We will now address these different kinds of rules, see Figure 5.1. Transition rules based on these typing rules have been given in the previous section.

Typing rules for the type constructors (contract, connector, signature, predicate) shall be omitted, except for the one for contracts, I-CTR. If the three names s, p_1 and p_2 are of type signature, predicate, and predicate, respectively, then the contract $\textsc{Ctr}(s, p_1, p_2)$ is of contract type:

$$\textsc{Ctr}(\textsc{Sig}(T_1, \ldots, T_n, \textsc{CRe}(T), \textsc{Prd}(F_1), \textsc{Prd}(F_2))$$

Subtype relations are in principle possible between types constructed with the same constructor. Two structural rules contribute to the definition of the subtype relation $\leq$: the reflexivity rule S-REFL and the transitivity rule S-TRANS:

$$[\text{S-REFL}]\ \frac{S =_\beta T}{\Gamma \vdash S \leq T}$$

$$[\text{S-TRANS}]\ \frac{\Gamma \vdash S \leq T \quad \Gamma \vdash T \leq U}{\Gamma \vdash S \leq U}$$

They show that $\leq$ is a preorder. The subtyping rules for signatures and predicates are S-SIG and S-PRD - see Figure 5.1. The names COND, PRE, POST, SIGN and their primed variants are type variables. A condition is subtype of another if it implies it: $\textsc{cond} \leq \textsc{cond}'$ if $\textsc{cond} \rightarrow \textsc{cond}'$. A

$$[\text{I-Ctr}]\ \frac{\Gamma \vdash s :_c \text{Sig}(T_1, \ldots, T_n, \text{CRe}(T))\quad \Gamma \vdash p_1 :_c \text{Prd}(T)\quad \Gamma \vdash p_2 :_c \text{Prd}(T)}{\Gamma \vdash \text{Ctr}(s, p_1, p_2) :_c \text{Ctr}(\text{Sig}(T_1, \ldots, T_n, \text{CRe}(T)), \text{Prd}(T), \text{Prd}(T))}$$

$$[\text{S-Sig}]\ \frac{\Gamma \vdash T_1' \leq T_1 \quad \ldots \quad \Gamma \vdash T_k' \leq T_k \quad \Gamma \vdash \text{CRe}(T) \leq \text{CRe}(T')}{\Gamma \vdash \text{Sig}(T_1', \ldots, T_n', \text{CRe}(T')) \leq \text{Sig}(T_1, \ldots, T_n, \text{CRe}(T))}$$

$$[\text{S-Prd}]\ \frac{\text{cond}' \rightarrow \text{cond}}{\Gamma \vdash \text{Prd}(\text{cond}') \leq \text{Prd}(\text{cond})}$$

$$[\text{S-Ctr}]\ \frac{\Gamma \vdash \text{pre} \leq \text{pre'}\quad \Gamma \vdash \text{post'} \leq \text{post}\quad \Gamma \vdash \text{sig'} \leq \text{sig}}{\Gamma \vdash \text{Ctr}(\text{sig'}, \text{pre'}, \text{post'}) \leq \text{Ctr}(\text{sig}, \text{pre}, \text{post})}$$

$$[\text{S-CAc}]\ \frac{\Gamma \vdash T_1' \leq T_1 \quad \ldots \quad \Gamma \vdash T_k' \leq T_k \quad \Gamma \vdash \text{CRe}(T) \leq \text{CRe}(T')}{\Gamma \vdash \text{CAc}(T_1', \ldots, T_k', \text{CRe}(T')) \leq \text{CAc}(T_1, \ldots, T_k, \text{CRe}(T))}$$

$$[\text{S-CRe}]\ \frac{\Gamma \vdash T' \leq T}{\Gamma \vdash \text{CRe}(T') \leq \text{CRe}(T)}$$

Figure 3: Typing rules.

contract forms a subtype of another if its precondition is weakened and its postcondition is strengthened - see S-Ctr - where sig, pre, post, sig', pre', and post' denote signature and predicate types. The port orientation also has to be considered. We assume that ports do not change their orientation. For connector activations we expect subtype relations for the value types to hold - see S-CAc. This definition is - similar to the signature subtypes - contravariant on the reply channel. A connector reply channel is a subtype of another if the value types that can be carried form a subtype - see S-CRe. Subtypes for the value kind shall be neglected for the rest of the paper - which has as a consequence that there are no proper subtypes between signatures and connector activations and replies.

5.2 Type Safety

Type safety concerns the relation between the type system and the operational semantics. The operational semantics are defined as transition semantics, specified by rules such as contract matching and connector establishment. Type safety comprises two issues. Firstly, evaluation should not fail in well-typed programs - we will introduce a notion of well-typedness shortly. Secondly, transitions should preserve typing. The judgment $\Gamma \vdash C$ denotes the well-typedness of composition expression C. This will be the construct to investigate type preservation under transition.

We need to define a notion of satisfaction before we can define well-typedness. A connector type satisfies a contract type if the signatures correspond and, if the precondition holds, the execution of the service attached to the connector port establishes the postcondition.

Definition 5.1. *A connector type* $T_I = \text{CAc}(T_1, \ldots, T_n, \text{CRe}(T))$ *satisfies a contract type* $T_C = \text{Ctr}(\text{sig}, \text{pre}, \text{post})$, *or* $T_I \models T_C$, *if for a service port* p *the connector port* p_I *satisfies the following constraints:* $\text{Sig}(T_1, \ldots, T_n, \text{CRe}(T)) = \text{sig}$ *and, if* pre *holds, then the execution of* p_I, *if it terminates, establishes* post.

We assume an analogous definition of satisfaction between data types and connector reply types and their connector activation type.

Definition 5.2. *We define well-typedness for simple actions as follows:*

- $\Gamma \vdash \text{Req}\ \overline{m_C}\langle m_I\rangle$ *if* $\mathcal{T}_c(m_I) \models \mathcal{T}_c(m_C)$ – *otherwise* $\text{Req}\ \overline{m_C}\langle m_I\rangle$ *fails.*
- $\Gamma \vdash \text{Pro}\ n_C(n_I)$ *if* $\mathcal{T}_c(n_I) \models \mathcal{T}_c(n_C)$ – *otherwise* $\text{Pro}\ n_C(n_I)$ *fails.*
- $\Gamma \vdash \text{Inv}\ \overline{m_I}\langle a, m_R\rangle$ *if* $type(a), \mathcal{T}_c(m_R) \models \mathcal{T}_c(m_I)$ – *otherwise* $\text{Inv}\ \overline{m_I}\langle a, m_R\rangle$ *fails.*
- $\Gamma \vdash \text{Exe}\ n_I(y, n_R)$ *if* $type(y), \mathcal{T}_c(n_R) \models \mathcal{T}_c(n_I)$ – *otherwise* $\text{Exe}\ n_I(y, n_R)$ *fails.*

The execution of an action fails, if the data sent along the channel does not satisfy the channel constraint. A reaction fails if both participating actions are well-typed, but the type constraint is not satisfied.

Definition 5.3. *The well-typedness of parallel compositions is defined by rule* [W-ParComp]*:*

$$\frac{\Gamma \vdash \text{Req}\ \overline{m_C}\langle m_I\rangle\quad \Gamma \vdash \text{Pro}\ n_C(n_I)\quad \Gamma \vdash \mathcal{T}_c(n_C) \leq \mathcal{T}_c(m_C)}{\Gamma \vdash \text{Req}\ \overline{m_C}\langle m_I\rangle | \text{Pro}\ n_C(n_I)}$$

If $\Gamma \vdash \text{Req}\ \overline{m_C}\langle m_I\rangle$ *and* $\Gamma \vdash \text{Pro}\ n_C(n_I)$, *but not* $\Gamma \vdash \mathcal{T}_c(n_C) \leq \mathcal{T}_c(m_C)$, *then* $\text{Req}\ \overline{m_C}\langle m_I\rangle | \text{Pro}\ n_C(n_I)$ *fails.*

Well-typedness guarantees correct composition and interaction behaviour according to the specifications given through the type system (pre- and postconditions) constraining behaviour and matching. The objective later on will be to show whether replacements preserve well-typedness; for example to show that if $\Gamma \vdash \text{Req}\ \overline{m_C}\langle m_I\rangle | \text{Pro}\ n_C(n_I)$ and if port m is replaced by m', then to show whether $\Gamma \vdash \text{Req}\ \overline{m_C'}\langle m_I'\rangle | \text{Pro}\ n_C(n_I)$ holds, i.e. whether correct behaviour is preserved by replacements.

We shall note type safety properties as conjectures only, without a formal proof.

Conjecture 5.1.

1. *Substitution lemma: if* $\Gamma \vdash C$ *and* $\Gamma \vdash x : T, v : T$, *then* $\Gamma \vdash \{v/x\}C$.
2. *Evaluation cannot fail in well-typed programs: if* $\Gamma \vdash C$ *then the execution of* C *does not fail.*
3. *Transition preserves typing: if* $\Gamma \vdash C_1$ *and* $C_1 \to C_2$ *then* $\Gamma \vdash C_2$.

5.3 Types as Formulas

There is a relationship between the contracts and connector types. Contract types can be see as Hoare logic or dynamic (modal) logic formulas consisting of a precondition and a postcondition, complemented by a signature. We have a two-layered type system with a layer of contract types and a layer of connector types with a notion of satisfaction between them. These types correspond to the distinction of specification and implementation for a component. The contract type CTR(SIG, PRE, POST) corresponds to the formula PRE $\to [n(a_1, \ldots a_k)]$ POST in dynamic logic where n : SIG. This refers to the specification of services of a component. The lower type layer corresponds to the implementation. Types for parameters are value types.

6. Client and Provider Life Cycles

In the previous sections, we have seen several stages in the life cycle of a component such as service matching and connector establishment, or service invocation. The full life cycle of clients, providers and systems consisting of both clients and providers shall now be specified in a standard form. The client, parameterised by a list of required services, can be specified as follows:

$$
\begin{array}{l}
C_i(m_1, \ldots, m_l) \stackrel{\text{def}}{=} \\
\quad \text{REQ } m_C^1\langle m_I^1\rangle.!(\text{INV } \overline{m_I^1}\langle a^1, m_R^1\rangle.\text{RES } m_R^1(y^1).0) \\
\quad | \\
\quad \ldots \\
\quad | \\
\quad \text{REQ } m_C^l\langle m_I^l\rangle.!(\text{INV } \overline{m_I^l}\langle a^l, m_R^l\rangle.\text{RES } m_R^l(y^l).0)
\end{array}
$$

Requests have to be satisfied before any interaction can happen. Once a connection is established, a service can be used several times. In order to function properly all service requests need to be satisfied - expressed by the parallel composition of all individual ports.

Service providers need to be replicated $!P$ in order to deal with several clients at the same time. Otherwise their behaviour is the dual to that of the clients.

$$
\begin{array}{l}
P(n_1, \ldots, n_k) \stackrel{\text{def}}{=} \\
\quad !(\ \text{PRO } n_C^1(n_I^1).!(\text{EXE } n_I^1(y^1, n_R^1).\text{REP } \overline{n_R^1}\langle b\rangle.0) \\
\quad\quad + \\
\quad\quad \ldots \\
\quad\quad + \\
\quad\quad \text{PRO } n_C^k(n_I^k).!(\text{EXE } n_I^k(y^k, n_R^k).\text{REP } \overline{n_R^k}\langle b\rangle.0)\ )
\end{array}
$$

A provider does not need to engage in interactions with all its ports, which is modelled by using the choice operator instead of the parallel composition.

Clients and a server are composed in parallel $CompSys \stackrel{\text{def}}{=} P(n_1, \ldots, n_k) \mid C_1(m_{1_1}, \ldots, m_{1_{m_1}}) \mid \ldots \mid C_j(m_{j_1}, \ldots m_{j_{m_j}})$ to form a composed system. Another case which also needs to be considered is that a component can be both client and provider, i.e. can both import and export services.

$$
\begin{array}{rcl}
Comp & \stackrel{\text{def}}{=} & (\text{REQ } m_C^1\langle m_I^1\rangle.0 | \ldots | \text{REQ } m_C^l\langle m_I^l\rangle.0). \\
& & \quad !(\ !(\ \text{INV } \overline{m_I^1}\langle \ldots\rangle.\text{REC } m_I^{R^1}(\ldots).0 \\
& & \qquad + \\
& & \qquad \ldots \\
& & \qquad + \\
& & \qquad\quad \text{INV } \overline{m_I^l}\langle \ldots\rangle.\text{REC } m_I^{R^l}(\ldots).0\ \ ) \\
& & \quad\quad + \\
& & \quad\quad P(n_1, \ldots, n_l)\ \ )
\end{array}
$$

The requirements have to be satisfied, i.e. connectors have to be established, before any service can be provided. A service which is provided and actually invoked can then trigger the invocation of imported services. The specification of composed systems does not involve the possibility for evolution - through the replacement of components - so far. This will be looked at in the next section.

7. Replacements and Evolution

In evolving systems, components might change in their specification or implementation, or are replaced by other components with different specification and implementation. Two questions arise. Firstly, can a component be replaced by another component without affecting the behaviour and the overall consistency of the system? This can be answered using a static analysis based on the component contracts. Secondly, what are the consequences if a replacement fails? This can affect a running system. The analysis has to be carried out based on the actual connectors between components in a running system. We assume that only a single component is replaced by another at a time. Components are the unit of change.

We will address replacements based on the type system – statically and dynamically – and the determination of effects if such a replacement results in inconsistencies. Types are explicit in our notation. That will allow us to change the type (and implementation) of a component, see Section 7.1. This can even be done dynamically for a running system, see Section 7.2. In case the types cannot be preserved, the effects of a change need to be determined, see Section 7.3. We shall look at replacements firstly as a meta-construct, then we will introduce it into the notation. We assume that replacing a component means replacing existing ports, possibly adding new ones. We discuss the replacement of a single port only in order to illustrate the issue. The following definitions formalise a consistent (effect-free) replacement.

Definition 7.1. *A* **context** X *is obtained when a hole* [.] *replaces an occurrence of* 0 *in a process expression. We write* $X[P]$ *for the replacement of* [.] *by* P *in* X.

Definition 7.2. *Given an arbitrary context* X, *a component (a process expression)* C *can be* **consistently replaced** *by a component* C', *if* $\Gamma \vdash X[C]$ *implies* $\Gamma \vdash X[C']$.

This describes the preservation of well-typedness under replacement. It guarantees that replacements do not affect the composition behaviour.

Proposition 7.1. *If* $\Gamma \vdash C$ *implies* $\Gamma \vdash C'$, *then* $\Gamma \vdash X[C]$ *implies* $\Gamma \vdash X[C']$ *for all contexts* X.

PROOF. Obvious. □

The dynamic replacement analysis based on types gives more flexibility. However, the result might be an effect on other components in form of a change of connections (mobility).

7.1 Replacement and Subtypes

Changes in structure - reflected by changes in connector types - are usually difficult to deal with, but changes in behaviour - here reflected by changes in contract types -, do not always affect the overall consistency of a composition (the consistency is affected if the specified behaviour is not preserved). Preservation of well-typedness is the technical criterion for this kind of analysis.

Since our aim is to determine whether one component can replace another, we can consider the type system and its subtypes. We will look at bound names in providers and free names in clients in particular. We do not consider type equivalence here; our concern is the replacement of one component by another relying on the subtype relation.

Clients, or service requestors, shall be addressed first. A port $m = (m_C : t_C, m_I : t_I, m_R : t_R)$ shall be replaced by $m' = (m'_C : t'_C, m'_I : t'_I, m_R' : t_R')$. Later on, we will assume that names do not change, only their types will.

In some situations, replacement preserves well-typedness: for $\Gamma \vdash X[C]$ and C' replaces C, we get $\Gamma \vdash X[C']$ for any context X. This shall now be investigated - firstly for a single component.

Proposition 7.2. *A requested service port $m_C :_p$ REQ can be consistently replaced by a port $m'_C :_p$ REQ if*

$$\mathcal{T}_p(m_C) = \mathcal{T}_p(m'_C) \ \wedge \ \mathcal{T}_c(m_C) \leq \mathcal{T}_c(m'_C)$$

PROOF. m is a refinement of m'. m' has consequently a stronger (more restrictive) precondition and a weaker (less specific) postcondition. $\Gamma \vdash \overline{m'_C}\langle m_I \rangle$ if $\Gamma \vdash \overline{m_C}\langle m_I \rangle$ and $\mathcal{T}_c(m'_C) \leq \mathcal{T}_c(m_C)$ and we assume that $\mathcal{T}_c(m'_I) \leq \mathcal{T}_c(m_I)$ for connectors. Therefore, well-typedness is preserved. □

We shall look at this issue considering one particular context: that of a parallel composition where a client and a provider match. In this particular context, we can loosen the constraint for well-typedness.

Proposition 7.3. *A component C' can replace a client component C in a composition $C|P$ preserving well-typedness, i.e. $\Gamma \vdash C|P \rightarrow \Gamma \vdash C'|P$, if $\mathcal{T}_c(n_C) \leq \mathcal{T}_c(m'_C)$ for a service n provided by P, a service m requested by C and replacement m' for m.*

PROOF. Suppose a composition $C|P$ exists where n of P is connected to m of C, i.e. $\mathcal{T}_c(n_C) \leq \mathcal{T}_c(m_C)$. As long as $\mathcal{T}_c(n_C) \leq \mathcal{T}_c(m'_C)$ the provider satisfies the requirements. This means that C' can replace C without affecting the behaviour of the composition. Well-typedness as formulated in the well-typedness rule for compositions (Definition 5.3) in Section 5.2 is preserved. □

Strengthening the client specification might be acceptable. A refinement m'_C of m_C is acceptable as long as $\mathcal{T}_c(n_C) \leq \mathcal{T}_c(m_C)$ is guaranteed. The condition $\mathcal{T}_c(m_C) \leq \mathcal{T}_c(m'_C)$ does not need to be satisfied, but would, if true, guarantee the well-typedness of the replacement. Proposition 7.3 is more flexible than Proposition 7.2, but Proposition 7.3 can only be checked dynamically for a composed system. This condition would have to be checked for all connections in a running system.

We have a similar situation for the service provider.

Proposition 7.4. *A provided service port $n_C :_c$ PRO can be consistently replaced by a port $n'_C :_c$ PRO if*

$$\mathcal{T}_p(n_C) = \mathcal{T}_p(n'_C) \wedge \mathcal{T}_c(n'_C) \leq \mathcal{T}_c(n_C)$$

PROOF. Analogously to Proposition 7.2. □

Here, refinements are always permitted as replacements. Analogously to clients, replacements are consistent (effect-free) as long as the connector remains intact.

Proposition 7.5. *A component P' can replace a server component P in a composition $C|P$ preserving well-typedness if $\mathcal{T}_c(n'_C) \leq \mathcal{T}_c(m_C)$ for a provided service n connected to m and the replacement n'.*

PROOF. Analogously to Proposition 7.3. □

For this form of analysis we have looked at contract ports and their types only. Firstly, because the contract matching is the crucial activity; we essentially consider only contract port related activities as observable. Secondly, their channel types involve the contracts - which the connectors are expected to oblige to. The client is expected to guarantee the precondition and the provider is expected to guarantee the postcondition if the precondition is satisfied. The types of the connector and connector reply ports have therefore been neglected.

7.2 Dynamic Replacement

In order to allow dynamic compositions and replacements, we introduce a new feature into our notation. We introduce an explicit configuration CFG $s(p : t_p)$ for a port that allows components to change their specification dynamically:

$$\begin{array}{lcl} Client_i & \stackrel{\text{def}}{=} & !(\text{CFG } s(m : t_m).\overline{C_i}\langle m \rangle) \\ Provider & \stackrel{\text{def}}{=} & !(\text{CFG } s(n : t_n).\overline{P}\langle n \rangle) \end{array}$$

where C_i and P are defined as in Section 6. The port specification and implementation, i.e. contract and connectors, are provided by some external process.

Ignoring name changes - they can always be introduced easily via renamings - this construct essentially allows us to change the type of a port dynamically. Changes in contract types can be dealt with. The type system shall therefore be revisited. Ports are associated with types, e.g. the typing context Γ can contain a binding $m_C \mapsto$ CTR(SIG,PRE,POST). These associations in the typing context can change through the execution of for example

$$\text{CFG } s(m_C : \text{CTR(SIG',PRE',POST')})$$

where SIG,PRE,POST and SIG',PRE',POST' denote signature and predicate types. We assume that the connector types do not change. The configuration has an effect on the type context. The semantics of CFG $s(p : t_p)$ is that of a dynamic declaration on Γ; we write $\Gamma[\text{CFG } s(p : t_p)]$. We need an initial configuration for a port, but a replacement can

essentially happen at any time:

$$
\begin{array}{lcl}
C & \stackrel{\text{def}}{=} & !\textsc{Cfg}\ s(m_C : t_{m_C}).\overline{C'}\langle m_C\rangle \\
C' & \stackrel{\text{def}}{=} & (\ \textsc{Cfg}\ s(m_C : t_{m_C}).\overline{C'}\langle m_C\rangle \\
 & & \quad + \\
 & & \quad \textsc{Req}\ \overline{m_C}\langle m_I\rangle.(\\
 & & \quad\quad \textsc{Cfg}\ s(m_C : t_{m_C}).\overline{C'}\langle m_C\rangle \\
 & & \quad\quad + \\
 & & \quad\quad !(\textsc{Inv}\ \overline{m_I}\langle\ldots\rangle.(\\
 & & \quad\quad\quad \textsc{Cfg}\ s(m_C : t_{m_C}).\overline{C'}\langle m_C\rangle \\
 & & \quad\quad\quad + \\
 & & \quad\quad\quad \textsc{Rec}\ m_R(\ldots).0)\ \ )\ \ )
\end{array}
$$

This describes that a replacement will always result in a re-establishment of the connector, i.e. the request for a service will be made again. If we want to make use of the results of well-typedness preserving replacements (a re-establishment of connectors is not necessary in that case), we need to make this explicit in the notation. We introduce a type-based guard, remotely similar to the match-operator found in some π-calculus variants: $[TJ]\,\pi.P$ where TJ is a boolean expression based on a type judgment. The type judgment acts as a simple condition making the typing context explicit in the notation. In our situation, we could specify

$$\ldots\ .\textsc{Cfg}\ s(m_C : t_{m_C}).[\neg(\Gamma \vdash C)]\overline{C'}\langle m_C\rangle.\ \ldots$$

expressing that only if typing is not preserved, the re-establishment of a connection is necessary after a replacement.

The substitution of the type context can be formulated in two *dynamic typing rules* based on results from Section 7.1. Proposition 7.2 proves the soundness of the following **client replacement rule** [R-CRPL] for the type system:

$$\frac{\Gamma \vdash m_C :_c t_C \qquad \Gamma \vdash m_C :_p \textsc{Req}}{\Gamma[\textsc{Cfg}\ s(m'_C : t'_C)] \vdash m'_C :_p t'_C}\langle\ t_C \leq t'_C$$

where we replace $m_C : t_C$ by $m_C : t'_C$. We assume that the signature of the port m does not change. We also assume that t'_C is a contract type. The rule expresses a substitution in the type context.

Corresponding to the replacement rule for clients, we can formulate a **provider replacement rule** [R-PRPL] - based on Proposition 7.4.

$$\frac{\Gamma \vdash n_C :_c t_C \qquad \Gamma \vdash n_C :_p \textsc{Req}}{\Gamma[\textsc{Cfg}\ s(n'_C : t'_C)] \vdash n'_C :_p t'_C}\langle\ t'_C \leq t_C$$

Note that only the type requirement for the contract has changed. The execution of the CFG-action does neither affect the composition nor the component state. It only changes the type context dynamically.

Proposition 7.6. *Replacement based on the rules client replacement* [R-CRPL] *and provider replacement* [R-PRPL] *preserves well-typedness. For $\Gamma \vdash C$ and C' replaces C by one of the rules, we get $\Gamma \vdash C'$.*

PROOF. Follows from Propositions 7.2 and 7.4. □

Similar rules for dynamic replacements can be defined based on Propositions 7.3 and 7.5. Methods to analyse and reason about replacements will be addressed next.

7.3 Non-preserving Replacements

The first step in replacing components is always an analysis of types. Based on these criteria a component might be replaced consistently. In case the replacement has to take place - for example due to changes in the execution environment (technical, legal, etc.) - but does not result in a consistent replacement, then the effect of the replacement on other connected components has to be determined. This could again be done statically by looking at all potential compositions based on the overall system specification (all clients and providers composed in parallel), but should rather be limited to those components actually connected to a component in form of connectors (and components connected to those) in a composed system.

Starting point for this dynamic analysis is the network of connected components at a certain moment of time. This is the *flowgraph* of the system, which describes the structure of the system in terms of its linkages between components - the concept of flowgraphs to describe the spatial structure of connected processes is introduced in [15] Chapters 4.1 and 9.3. Flowgraphs and dependency analysis based on contract types shall now be looked at. A component export depends on the component's import; the import depends on another component's export. This dependency relation is transitive and allows us to determine which other components are potentially affected if one component has to be changed. The simple dependency relation needs to be refined. The reason is that a change in the export interface of one component (to the worse) might still satisfy the requirements of a service request. In this case there is no further effect. In case a second component is affected, then it can be tried to replace this component as well (using static analysis).

Definition 7.3. *Two kinds of graphs shall be introduced. A* **flowgraph** *is a graph where nodes are ports m_C, m_I, m_R, n_C, n_I, n_R, ... and edges are connections (m_C, n_C), (m_I, n_I), (m_R, n_R), ... The edges are directed and express dependencies. (m_I, n_I) expresses that a request m_I depends on a provided service n_I. A* **dependency graph** *is a flowgraph extended by component internal dependencies, e.g. (n_I, m_I) saying that export n_I depends on import m_I. Thus, a dependency graph has two kinds of edges: edges (m_I, n_I) are connectors between components and edges (n_I, m_I) are internal dependencies. If (p_1, p_2) and (p_2, p_3) then (p_1, p_3). Types (port type and channel type) shall be associated with each port node.*

Semantically the dependency graph is a bipartite graph defined on two different relations: the *subtype relation* between components and component-internal dependencies between service exports and services imports. A component needs to satisfy its imports in order to provide services to other components. By default, all combinations between input and output services are included in the dependency graph. A more precise account of internal dependencies can be derived from a component life cycle specification (see specification of *Comp* on page 6), which describes the extenally visible interaction behaviour of a component. Requests that occur before a service provision indicate a potential dependency. This could be refined explicitly by the component developer, but this would require to consider the actual service implementations.

The well-typedness of a composition shall be expressed by a consistency notion for dependency graphs.

Definition 7.4. *A dependency graph (or a flowgraph) is* **consistent** *if for all edges (m, n) of the connector kind:*

$$\mathcal{T}_c(n) \leq \mathcal{T}_c(m)\ \wedge\ \mathcal{T}_p(m) \simeq \mathcal{T}_p(n)$$

A graph update is an update of type associations for nodes (ports). This corresponds to the replacement of type bindings $\Gamma[\text{CFG } s(p : t_p)]$ in type context Γ.

Definition 7.5. *If a dependency graph is not consistent for a connector edge (p_2, p_1), then the* **effect** *of the inconsistency is the collection of all (p_i, p_j) with $i \geq 2, j \geq 1$ such that p_i depends on p_1.*

The effect of an inconsistency can be calculated based on the closure of the dependency relation. Classical algorithms can be used here. Therefore, this shall be neglected.

8. Related Work

A composition language for components which is also based on the π-calculus is presented in [20, 21]. A variation of the π-calculus is used to realise a composition language, called PICCOLA, which supports various forms of components, and, thus, various composition mechanisms. The basis is a formalisation of interacting objects as processes. Key concepts are glue code for component compositions and adaptation, and a scripting language to express this glue code.

Catalysis is a development approach building up on the UML incorporating formal aspects such as the pre- and postcondition technique [22]. Catalysis uses ideas from formal languages such as OBJ, CLEAR or EML. The concept of the connector that we have used here is motivated by the Catalysis approach. There, connectors allow the communication between ports of two objects. A connector defines a protocol between the ports. Several other authors also address contracts based on pre- and postconditions for the UML, see e.g. [9]. The combination of the pre- and postcondition technique and refinement calculi is explored in e.g. [23].

KobrA [24] is another approach which combines the UML with the component paradigm. The basic structuring mechanism is the *is-component-of* hierarchy, forming a tree-structured hierarchy of components, i.e. sub-components. Each component is described by a suite of UML diagrams. A component consists of a specification (an abstract export interface) and a realisation.

An early version of this paper has appeared in [25]. There we also looked at the connection between the π-calculus for component composition semantics and modal logics. In [25], we addressed the application of the framework to the Unified Modelling Language UML [26]. The connector idea is taken from [22]. In another paper [27], we have used a variant of the λ-calculus to define service requests and provisions using reduction as the mechanism for matching between services. The variant is called $\lambda\pi$-calculus [28] adding a flexible parameter (matching) concept to the λ-calculus. This $\lambda\pi$-calculus can be interpreted in object structures. We have used a π-calculus variant here, because it offers multiple (concurrent) connections and it allows to model two layers: contracts and connectors.

Walker [29] introduces object intercommunication into the π-calculus. In our approach the user (the client) is the active entity which initiates the establishment of the connections. In Walker's formalisation, the service provider also provides the communication channels. The service user acquires the contract channel, then acquires the interaction channels via the appropriate contract channels and finally uses the interaction channels to invoke methods of the service providers.

9. Conclusions

The suitability of the π-calculus for the definition of a component composition framework has recently been demonstrated, see for example the PhD-theses [30] and [31]. The idea of using a process calculus to model component composition has here been carried further by exploiting the similarity of mobility and evolution. We have developed a simple framework for the determination of effects of changes in composed systems. Our framework addresses in particular the problem of replacing single components in a system.

Mobility - the change of connection between components - is the key feature in the π-calculus. We have introduced evolution through replacement into our variant. This concept is somewhat different from mobility. Replacement is more fundamental since it is a meta-operation affecting the definition of the system (in particular the type system) under consideration. Replacement can cause mobility - the change of connections - as the result of the change in types. Replacement - as we have seen - can be introduced into the calculus, resulting in a dynamically typed calculus.

Transitions in the system based on the establishment of new connections occur in several variants. Reaction between two components is constrained by port types and channel types. Ports of different types match under different circumstances, formalised by a subtype relation. The reactions are transitions in a transition system, whose states reflect the state of composition that ports are in. Each port can pass through different composition stages - expressed by the life cycle for clients and providers.

Bisimulations and similar relations of equivalence between processes are essential concepts in the π-calculus for comparing processes. This theory could be applied in our context, if we would consider the replacement of systems of composed components. In that case, we would only be interested in the externally observable behaviour of these systems. Then, weak bisimilarity could be the tool to define and analyse replacements of compositions. However, our assumption here has been that single components are the unit of change. Consequently, we have based our replacement analysis mainly on the type system.

Some component technologies provide features for the discovery of services. This is an implicit process here, described by the contract rule. The overall model does not involve an intermediary. The CORBA framework for the interaction of remote objects is based on an object request broker. Here, this functionality is implicit. In an extension of the approach, an explicit broker could be considered. Another element of future work includes the exploration of the relationship of modal logics and the type system for our composition and replacement calculus.

References

[1] E.K. Nordhagen. *A Computational Framework for Verifying Object Component Substitutability.* PhD thesis, University of Oslo, November 1998.

[2] W. Weck. Inheritance Using Contracts & Object Composition. In *Proceedings 2nd International Workshop on Component-Oriented Programming WCOP'97.* Turku Center for Computer Science, General Publication No.5-97, Turku University, Finland, 1997.

[3] S. Cimato and P. Ciancarini. A Formal Approach to the Specification of Java Components. In B. Jacobs, G.T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Tech. Rep. 251, University of Hagen, 1999.

[4] B. Pierce and D. Sangiorgi. Typing and Subtyping in Mobile Processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[5] N. De Palma, P. Laumay, and L. Bellissard. Ensuring Dynamic Reconfiguration Consistency. In *Proceedings 6th Int. Workshop on Component-Oriented Programming WCOP2001*. http://research.microsoft.com/users/cszypers/events/, 2001.

[6] L. Tan, B. Esfandiari, and B. Pagurek. The SwapBox: A Test Container and a Framework for Hotswappable JavaBeans. In *Proceedings 6th Int. Workshop on Component-Oriented Programming WCOP2001*. http://research.microsoft.com/users/cszypers/events/, 2001.

[7] J.B. Warmer and A.G. Kleppe. *The Object Constraint Language – Precise Modeling With UML*. Addison-Wesley, 1998.

[8] G.T. Leavens and A.L. Baker. Enhancing the Pre- and Postcondition Technique for More Expressive Specifications. In R. France and B. Rumpe, editors, *Proceedings 2nd Int. Conference UML'99 - The Unified Modeling Language*. Springer Verlag, LNCS 1723, 1999.

[9] L.F. Andrade and J.L. Fiadero. Interconnecting Objects via Contracts. In R. France and B. Rumpe, editors, *Proceedings 2nd Int. Conference UML'99 - The Unified Modeling Language*. Springer Verlag, LNCS 1723, 1999.

[10] K. R. Apt. Ten Years of Hoare's Logic: A Survey – Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[11] Dexter Kozen and Jerzy Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 789–840. Elsevier Science Publishers, 1990.

[12] O. Nierstrasz and T.D. Meijler. Requirements for a Composition Language. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-based Models and Languages for Concurrent Systems*, pages 147–161. Springer-Verlag, 1995.

[13] G.T. Leavens and M. Sitamaran. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[14] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes - parts I/II. *Information and Computation*, 100(1):1–77, 1992.

[15] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[16] D. Sangiorgi and D. Walker. *The π-calculus - A Theory of Mobile Processes*. Cambridge University Press, 2001.

[17] C. Morgan. *Programming from Specifications 2e*. Addison-Wesley, 1994.

[18] R.J.R. Back and J. von Wright. *The Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[19] A. Moorman Zaremski and J.M. Wing. Specification Matching of Software Components. In Gail E. Kaiser, editor, *Proc. ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 6–17. ACM Software Engineering Notes 20(4), October 1995.

[20] M. Lumpe, J.-G. Schneider, O. Nierstrasz, and F. Achermann. Towards a Formal Composition Language. In G.T. Leavens and M. Sitamaran, editors, *Proceedings European Conference on Software Engineering ESEC'97*, pages 178–187. Springer-Verlag, 1997.

[21] M. Lumpe, F. Achermann, and O. Nierstrasz. A Formal Language for Composition. In G.T. Leavens and M. Sitamaran, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[22] D. D'Souza and A.C. Wills. *Objects, Components and Frameworks in UML: the Catalysis approach*. Addison-Wesley, 1998.

[23] M. Büchi and E. Sekerinski. Formal Methods for Component Software: The Refinement Calculus Perspective. In *Proceedings 2nd International Workshop on Component-Oriented Programming WCOP '97*. Turku Center for Computer Science, General Publication No.5-97, Turku University, Finland, 1997.

[24] C. Atkinson, J. Bayer, O. Laitenberger, and J. Zettel. Component-Based Software Engineering: The KobrA Approach. In *Proc. International Workshop on Component-Based Software Engineering, Limerick, Ireland*. 2000. ICSE (International Conference on Software Engineering) Workshop.

[25] C. Pahl. Components, Contracts and Connectors for the Unified Modelling Language. In *Proc. Symposium Formal Methods Europe 2001, Berlin, Germany*. Springer-Verlag, LNCS-Series, 2001.

[26] H.-E. Eriksson and M. Penker. *UML Toolkit*. John Wiley & Sons, 1998.

[27] C. Pahl. Modal Logics for Reasoning about Object-based Component Composition. In *Proc. 4rd Irish Workshop on Formal Methods, July 2000, Maynooth, Ireland*. 2000.

[28] L.M.G. Feijs. The Calculus $\lambda\pi$. In *Algebraic Methods: Theory, Tools and Applications*, pages 307–328. Springer-Verlag, 1989.

[29] D. Walker. Objects in the π-Calculus. *Information and Computation*, 115:253–271, 1995.

[30] M. Lumpe. *A π-Calculus Based Approach for Software Composition*. PhD thesis, Universität Bern, Institut für Informatik und angewandte Mathematik, 1999.

[31] J.-G. Schneider. *Components, Scripts, and Glue*. PhD thesis, Universität Bern, Institut für Informatik und angewandte Mathematik, 1999.

Analysis of Component-Based Systems – An Automated Theorem Proving Approach *

Murali Rangarajan
Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN 55418.
mrangara@htc.honeywell.com

Perry Alexander
EECS Dept., Info. and Telecomm. Tech. Center,
The University of Kansas
2291 Irving Hill Rd, Lawrence, KS 66044-7321.
alex@ittc.ukans.edu

ABSTRACT

As systems become increasingly complex, there is an increasing thrust towards designing systems at the requirements level. This approach enables the analysis of various system properties such as functional correctness, constraint satisfaction, et cetera at a very early stage in systems development, thus enabling faster design of systems with fewer design flaws. Analyses can be performed at various levels of rigor. For mission critical systems, analysis using formal techniques is highly preferable as it provides the highest level of rigor. But the problem with using formal analysis techniques is that they are either intractable for large designs, or require highly specialized knowledge possessed by a few select people. This prevents the majority of the design population from using such formal analyses in their design process. In this paper, we describe our approach for analysis of component-based systems for functional correctness using theorem proving techniques. The components and the design are specified using the VSPEC specification language. The model is translated into an equivalent model in the PVS specification language, and various correctness properties are automatically extracted from the model and their proofs are proofs are automatically attempted using specialized, dynamically generated, proof macros. Results of applying our technique to various modeling problems are provided, and the results are discussed.

1. INTRODUCTION

Hardware and software systems are becoming increasingly complex. Hardware systems consisting of millions of transistors and software systems requiring hundreds of thousands of lines of code are now commonplace. Along with the complexity of systems, methodologies have evolved to handle the complexity. While these methodologies provide ease of design, even for large systems, they do not have sufficient integrated support for analyzing correctness of designs. For example, top-down and bottom-up hierarchical design strategies are widely used to help design large systems. These by themselves do not ensure correctness of design. What is needed is some mechanism for analyzing functional correctness of systems during the design process.

In recent times, a number of new techniques and methodologies have been proposed to handle systems design complexity. These involve specifying requirements in a Requirements Specification Language and analyzing specifications to arrive at the correctness of designs. A number of semi-formal and formal analyses have been proposed. Semi-formal methods and formal methods employing techniques such as model-checking are easy to apply since a high degree of automation can be achieved using these techniques. But semi-formal methods do not provide the requisite rigor to be applicable for safety/mission critical systems. Model-checking based analyses are restricted by the problem of state-space explosion (even after the use of state-folding), especially at the high levels of abstraction frequently encountered at the design phases. Moreover, compositional analysis using model-checking is not straightforward. Formal methods employing theorem proving have a wide range of applicability and can provide the mathematical rigor required for analyzing safety/mission critical systems. But they are difficult to apply as they require in-depth knowledge of formal notations and theorem proving techniques.

Application of theorem proving for analyzing designs could, according to the Formal Methods community, have a great impact in the industry. However, modern designs are large and complex, and formal analysis of such designs directly is not practical. Therefore, a simplified model is analyzed for correctness, from which the correctness of the original model is assumed. But this need not be always true. Theorem proving has the capability to directly analyze large and complex designs. Such direct analysis provides more accurate information about the correctness of designs. Thereby, use of theorem proving techniques can lead to better designs.

Theorem proving is not currently practical for a number of reasons. The first problem is identifying what needs to be proved. This is a deceivingly complex task. The second problem is that of specifying what needs to be proved in

*Support for this work was provided in part by the CEENSS Technology Program, contract number F33615-93-C-4304, and the RASSP Technology Program, contract number F33615-93-C-1316.

some theorem proving language. The complex and mathematical nature of such languages makes this a daunting task for most designers. The third problem is that of modeling the domain in the same specification language as that of the theorems. The final problem is that of proving the theorems once they have been specified in the language of a theorem prover. The proof process varies wildly with the theorem being proved, the design being verified, and the underlying models on which the theorem is based.

The purpose of this work is to devise a mechanism for automated analysis of component-based designs for common classes of errors. We use the VSPEC specification language to demonstrate our approach. From VSPEC specifications, equivalent PVS theories are automatically generated. The generated theories contain verification obligations (as theorems) for interfaces and interconnections. Automated proof assistance is provided for the verification activities. This approach was applied to a variety of example problems. The results (presented in section 4) showed that it is possible to generate PVS models corresponding to VSPEC specifications, and that properties about interfaces and interconnections can be generated as theorems. They also showed that it is possible to automatically generate proof scripts for certain categories of models such that the generated theorems can be automatically proved by the PVS theorem prover.

Since the goal was to automate the proof process, a number of simplification strategies were adopted. First, the semantics of VSPEC was kept simple by removing all unnecessary extensions, while at the same time, ensuring the validity of the resulting semantics. Next, the properties to be analyzed were generated from at-most two levels in the design hierarchy of the system. This results in much simpler properties to prove than if we were to consider the design as a whole. Recursive application of this technique to all the levels in the design provides information about the overall correctness of the design. Finally, the proofs themselves are generated as sequences of LISP commands, the same format used by PVS. This enables the proofs steps to be executed automatically to check whether a certain property is applicable to a design. It is to be noted here that the proof macros only increase the possibility of a proof being completed, but they do not guarantee it. Whether the proof is actually completed or not depends upon a number of factors such as the complexity of the design, complexity of the specifications, the property being analyzed, etc.

2. VSPEC

VSPEC is a requirements specification language for VHDL. VSPEC was originally designed as a Larch Interface Language for VHDL. Therefore, VSPEC borrows a number of features from VHDL. The VHDL entities, architectures, and packages are directly used by VSPEC. The information provided by VSPEC is specified in a `specification` construct. VSPEC `specifications` are similar to VHDL architectures in that they provide additional information about an existing entity. VSPEC's declarative specification style complements the traditional VHDL operational style. Together, VSPEC and VHDL support modeling from requirements acquisition through verification and synthesis.

As a working example, a VSPEC description of a sorting component is shown in Figure 1. The entity `sort` is identical to the VHDL entity construct. This provides the interface for the VSPEC specifications. The package construct is also similar to that of VHDL, with the exception of the keyword `mutable`. This type specifier has been added in VSPEC to enable the designer to specify complex types without giving any particular implementation.

The module `sort_spec` constitutes a VSPEC specification of the `sort` entity. The `sensitive to` clause is similar to sensitivity lists and the `wait` statement in VHDL – it defines when the component is active. It is basically a boolean predicate indicating when an entity should begin executing. The functional requirements are defined using the `requires` (pre-condition) and `ensures` (post-condition) clauses. These two clauses define component function as a relationship between current and next state axiomatically. Any implementation that makes the post-condition true in the next state, given that the pre-condition is true in the current state, is a valid implementation of these requirements. The `includes` clause is used to include PVS definitions in a VSPEC description. The sorts and operators defined in the PVS theories named by the `includes` clause can be used in the VSPEC definition. In the example specification from Figure 1, the `sort` component operates correctly in any initial state whenever its input changes and produce an output that is ordered and is a permutation of the input. Note that `event` is a predefined VSPEC predicate that is true whenever its associated signal changes values in the previous state change.

In addition to allowing the designer to describe functional requirements, VSPEC also allows the designer to specify performance constraints using the `constrained by` clause. This clause defines relations over constraint variables such as power consumption, layout area (expressed as a bounding box), heat dissipation, clock speed and pin-to-pin timing. Constraint theories are written in PDL [4], and verified using the associated evaluation tool. Users may define their own constraints and theories if desired [3].

The functional semantics are modeled upon the semantics of VHDL under simulation. Therefore, each entity behaves as an independent process, interacting with the outside world using messages sent and received through its ports. Each entity is modeled as a CSP [2] *process*, and architectures are modeled using CSP's *parallel composition* operator. Component interaction is specified in terms of *events*. Events are instantaneous actions that represent some real-world occurrence of interest. The set of events considered relevant for a particular description of an object is called the object's *alphabet*. A process represents the behavior pattern of an object described in terms of events from the object's alphabet. A sequence of events that a process participates in is called a *trace* of that process. A process is fully defined by its alphabet and the set of all possible traces of that process.

The semantics for VSPEC is given in the PVS specification language [1]. This includes definitions for all the operators and types used in VSPEC, and the meanings for the various VSPEC clauses. This formal definition of all the aspects of the language in PVS enables the formal analysis of models, as detailed in the next section.

```
package sort_pkg is
    type integer_array is mutable;
end sort_pkg;

use sort_pkg;
entity sort is
    port (input: in integer_array;
          output: out integer_array);
end sort;

use sort_pkg;
specification sort_spec of sort is
    includes SortPredicates;
    sensitive to input'event;
    requires true;
    ensures
        permutation(output'post, input)
        and inorder(output'post);
    constrained by
        power <= 5mW and size <= 3um * 5um
        and heat <= 10mW and clock <= 50MHz
        and input<->output <= 5 ms;
end sort_spec;
```

Figure 1: VSPEC description of a sorting component.

3. TRANSLATION

An example VSPEC file is shown in figure 2, and its corresponding generated-PVS file is shown in figures 3 and 4. The generation process is purely syntactic and is completely automated by the VSPEC parser. The heart of the transformation to PVS involves: (i) transforming ports and state variables into Store representation; and (ii) manipulating the `requires`, `ensures` and `sensitive to` clauses. The general structure of all generated PVS theories are similar, with the basic differences being in the parameters to the theories, and in the right hand sides of the various axioms.

```
entity m3 is
port ( in1 : in integer; inout1 : inout integer;
       out1 : out integer);
end m3;

specification m3_spec of m3 is
begin
    state state1 : integer;
    sensitive to in1'event;
    modifies inout1;
    requires in1 > inout1;
    ensures out1'post = state1 and
            in1 = inout1'post;
end m3_spec;
```

Figure 2: Example VSPEC file

The state of any VSPEC system is defined using a *Store*. The store is a simple abstraction of the record structure containing all the ports and state variables. Each theory representing a specification must have access to the type `Store`. Multiple definitions of a type in PVS results in each definition being a *different* type, thereby making proofs over stores impossible. To get around this problem, the `Store` type and the constant `empty` are passed as parameters to all the entity theories as their first two parameters. For the same reason, the type `Component` is also passed as a parameter. Since there always is a 'root' theory from which the analysis starts, this process ensures that only one `Store` and `Component` are visible throughout the system.

If the original VSPEC file had imported any PVS theories, they would be imported at this point. The reason for importing theories at this point is that types defined in those theories may be needed for the remaining parameters to the theory. The locations of the included theories are specified by the corresponding `LIBRARY` declarations. In our example, since there are no included theories, no `importing` statements are generated here. The remaining parameters are the declarations of the port variables. They are declared as functions from a `Store` to their corresponding type.

The body of the theory starts by declaring a constant `comp` that represents the current component. All the properties of components are defined over their respective `comps`. This enables the linkage of various properties to specific components. Next, the `OneComponent` and the `TypeSpecificInfo` theories are included. The `OneComponent` theory specifies all aspects of the process (associated with the VSPEC entity) independently from any VSPEC component. Theories representing specific components specialize `OneComponent`. The advantage of this approach is that verification of `OneComponent` need only be performed once. The basic theorems need not be reproven each time. The theory `TypeSpecificInfo` defines some operators that are common to all types. This theory is included once for each data type used in the system.

The state variables are all declared in a manner analogous to the port variables. They are declared as constant functions from a `Store` to their corresponding types. Next, some generic variables used in the axioms and theorems are declared.

The `sensitive to`, `requires` and `ensures` clauses are each transformed into axioms over stores. The `sensitive to` clause is defined by the axiom `sensitive_ax`, the `requires` clause by the axiom `requires_ax` over the `pre` state, and the `ensures` clause by the axiom `ensures_ax` over the states `pre` and `post`. The transformation of these clauses involves two basic activities: (i) combining the various occurrences of each clause into one; and (ii) replacing variable references with functions over Store.

The `modifies_event_ax` axiom and the `input_event_ax` axioms are part of the semantics of component activation and define when the functions `modSet_event` and `input_event` are true. The former is true when there is an event on one of the `modifies` variables (including the OUT and STATE variables). The latter is true when there is an event on an IN or INOUT variable.

For the component to ever become active, its initial state must be a part of the set of active states of the process. This fact is ensured by the first theorem (`initstates_th`) in the generated theory. Since, currently, there is no mechanism

```
%% PVS representation of specification m3_spec of entity m3
m3_spec [ Store: TYPE+, empty: Store, Component: TYPE+,
  in1 : [Store -> integer], inout1 : [Store -> integer],
  out1 : [Store -> integer]] : THEORY
BEGIN
% The component corresponding to this entity
comp: Component

IMPORTING jvsp@OneComponent[Store, empty, Component, comp]
...
%% The state variables in this entity
state1 : [Store -> integer]

%% Variables used in theorems and axioms
pre, post, any: VAR Store

%% Part of definition for semantics for event
modifies_event_ax : AXIOM modSet_event(comp)(any) =
  ( event(out1, any) OR event(inout1, any)  )
...
%% Requires Clause defines I
requires_ax : AXIOM I(comp)(pre) = (   in1(pre) > inout1(pre) )

% Ensures clause defines O
ensures_ax : AXIOM O(comp)(pre,post) =
  (  out1 (post)  = state1(pre) AND in1(pre) =  inout1 (post)  )
% Sensitive to clause
sensitive_ax : AXIOM member(pre,Psi(comp)) = ( event(in1, pre))
```

Figure 3: Partial PVS translation of specification m3_spec, part 1 – Variables and axioms

to specify the initial values of the various port and state variables, the axiom `initstates_ax` asserts that the initial state is equal to `Psi`. This automatically ensures us that `InitStates` is a subset of `Psi`.

The remaining theorems represent the single-component proof obligations. The completeness proof obligation is generated as the theorem `complete_th`, the witness for incompleteness obligation as theorem `incomplete_th` and the inconsistency obligation as the theorem `inconsist_th`. This completes the generated PVS theory for the specification `m3_spec`.

Abstract architectures are defined by: (i) specifying communication paths between components; and (ii) defining activation conditions to indicate when components should process inputs. The state of an architecture is defined to be the union of its components states. Component communication is achieved when their states share objects. Activation is the VSPEC dual of VHDL's sensitivity lists and indicate when a component should process its input. Together, communication and activation define the semantics of architecture specifications.

The VSPEC architecture for the `find` component is shown in figure 5. The VSPEC architecture is identical to the VHDL architecture, except for the use of the key word `VSPEC` to denote VSPEC specifications rather than VHDL components during instantiation. The variables in the signal declaration represent internal connections between the components in the architecture. Following the signal declarations, the components in the architecture are instantiated with appropriate parameters. The parameters represent connections with other components, or with the interfaces of the architecture, based on the name of the parameter. This concludes an architecture description. In our example, the `find_arch` architecture has two components – the `sorter` component, which is an instance of the `sort` component, and the `searcher` component, which is an instance of the `bin_search` component.

The PVS representation for this architecture (Figures 6, 7) has, as usual, the parameters `Store, empty` and `Component`. This enables the architecture to be imported in other architectures. Following this, the theory `Architecture` is imported to provide semantics to the various architecture operators. The ports of the higher-level component (`find`, in our example) are then declared. These form the ports of the architecture too, and provide inputs to and obtain outputs from the components in the architecture. The signals, which provide connections between the components in the architecture, are declared next.

The specifications of the higher-level component and the components in the architecture are imported with appropriate (depending upon how the component is connected) instantiations. The sole axiom of the theory (theorem `arch_comps_Axiom` in our example) defines the set `arch_comps` to be composed of the `comps` of all the components in the architecture. This declaration is essential for the definition of the architecture process, the process representing the parallel composition of these components.

```
%% Possible initial value in all traces of entity_process.
initstates_ax: AXIOM InitStates(comp) = Psi(comp)

%% Value of outputs and state variables does not change between post &
%% any. Initstates must be a subset of Psi for the model to be valid!
initstates_th: THEOREM subset?(InitStates(comp), Psi(comp))

%% Completeness obligation
complete_th: THEOREM (member(pre, LegalStates(comp)) AND
  member(pre, Psi(comp))) => I(comp)(pre)

%% Witness for incompleteness
incomplete_th: THEOREM EXISTS (x: Store): (member(x,LegalStates(comp))
  AND member(x, Psi(comp))) => NOT I(comp)(x)

%% Inconsistency obligation
inconsistent_th: THEOREM (member(pre, LegalStates(comp)) AND
  member(pre, Psi(comp))) => NOT I(comp)(pre)

END m3_spec

%% End of PVS representation of entity
```

Figure 4: PVS translation of specification m3_spec, part 2 – Theorems

```
architecture find_arch of find is
  signal sig_out1, sig_out2, sig_out3, sig_out4: integer;
begin
  sorter: VSPEC entity sort
          port map (in1, in2, in3, in4, sig_out1, sig_out2, sig_out3,
          sig_out4);
  searcher: VSPEC entity bin_search
            port map (sig_out1, sig_out2, sig_out3, sig_out4, key,
                      output);
end architecture find_arch;
```

Figure 5: VSPEC specification of `find` component's architecture

The generated theorems represent the various proof obligations for architectures. The input consistency proof obligation is generated as the `input_interface_th` theorem, the output consistency proof obligation as the `output_interface_th` theorem, strong liveness proof obligation for `sort` component as the `strong_live_sort_th1` theorem, and weak liveness and inconsistency proof obligations of the `bin_search` component as the theorems `weak_live_bin_search_th2` and `inconsist_bin_search_ th3`. The numbers at the end of theorem names are used to disambiguate between the same proof obligations of the same component used multiple times in the architecture. The `sort` component does not have theorems corresponding to the weak liveness and inconsistency proof obligations as there are no other components providing inputs to it. All its inputs are obtained from the interface of the architecture. Similarly, the `bin_search` component does not have a strong liveness theorem as all its outputs are part of the architecture interface.

4. PROOF AUTOMATION MECHANISM

Automation is critical for the success of any new methodology, and application of theorem proving is not an exception. But automation of theorem proving, in the general case, is not possible. Our approach has been to generate proof steps that have a high probability of successful completion. This approach is facilitated by the fact that both the generated theorems, and the semantics required for their proofs, have been written by us. This enables us to fine-tune the proof steps for individual theorems.

The proof steps generated for individual theorems are dependent upon the terms in the theorem. The general approach is to `LEMMA` the relevant axioms and theorems, instantiate them with appropriate constants, perform appropriate replacements of terms, and finally, to use PVS's built-in macro `GRIND` to attempt completion of the proof. Powerful prover commands provided by PVS are used in order to generalize the generated proof. For example, the `(INST?)` command provided by PVS attempts to automatically instantiate universally quantified variables in the antecedent with appropriate skolem variables.

The proof process starts by using the VSPEC parser to parse a component specification or architecture. The parser's `--pvs` flag generates the PVS equivalent of the parsed module. While generating theorems for the PVS module, the

```
input_interface_th: THEOREM member(pre,
        Psi(find_spec[Store, empty, Component, in1, in2, in3, in4,
        key, output].comp)) =>
    (member(pre, Psi(sort_spec[Store, empty, Component, in1, in2, in3,
        in4, sig_out1, sig_out2, sig_out3, sig_out4].comp))
    or member(pre, Psi(bin_search_spec[Store, empty, Component,
        sig_out1, sig_out2, sig_out3, sig_out4, key, output].comp)))
...

strong_live_sort_th1: THEOREM
    O(sort_spec[Store, empty, Component, in1, in2, in3, in4, sig_out1,
        sig_out2, sig_out3, sig_out4].comp)(pre,post)
    => ( member(post, Psi(bin_search_spec[Store, empty, Component,
        sig_out1, sig_out2, sig_out3, sig_out4, key, output].comp)) )

weak_live_bin_search_th2: THEOREM
    (member(pre, Psi(sort_spec[Store, empty, Component, in1, in2, in3,
        in4, sig_out1, sig_out2, sig_out3, sig_out4].comp))
    and O(sort_spec[Store, empty, Component, in1, in2, in3, in4,
        sig_out1, sig_out2, sig_out3, sig_out4].comp)(pre, post))
    => member(post, Psi(bin_search_spec[Store, empty, Component,
        sig_out1, sig_out2, sig_out3, sig_out4, key, output].comp))
...
END find_arch
% End of PVS representation of architecture.
```

Figure 7: Partial PVS representation of `find` architecture, part 2

corresponding proof scripts and LISP files for use with the PVS proof checker are also simultaneously generated. After completion of the parsing, the PVS proof checker is invoked in the batch mode. Since PVS uses a LISP interface, a LISP file, containing a sequence of commands for execution by PVS, can be passed to PVS in the batch mode.

The loader file is generated by the parser, and has a sequence of commands for execution by the PVS proof checker. The `load` commands loads a file called `prfobs_fns.lisp`, which defines two main LISP functions – `mytc` and `myprove`. The `mytc` function instructs PVS to typecheck a file. The `myprove` function first installs the generated proof script, then uses it to prove a specification. The results are stored in a file called `proof_status`, which is printed out at the end of the proof process.

5. RESULTS AND EVALUATION

In this section, we present the results of analyzing model systems using our approach. The main aim of performing this evaluation was to identify the factors that affect the automatability of proofs. Towards this end, the example systems incorporate a wide variety of situations. They include control-based and data-based activation of components; linear, branching and feed-back architectures; and systems with intentional bugs that result in incompleteness (necessitating proofs over existential quantifiers).

The automated theorem-proving analysis has been applied to a number of systems. The `find` system has been described throughout this dissertation. It was chosen for illustration as it is a conceptually simple example that demonstrates most of the features of our approach. Apart from the `find` example, the `AlarmClock` system (a synthesis benchmark developed by Synopsis), the `PIP` system (a Digital Signal Processing System developed by our sponsors), and the `CruiseControl` system (a standard modeling problem) were also analyzed.

The results of our analyses are presented in table 1. The first column of the table lists the various modules that were analyzed. The second column indicates whether a module is a single component or an architecture. The third column lists the number of theorems that were automatically proved. The fourth column lists the number of theorems with *possible* proofs in that module. This is critical for statistical analysis of the effectiveness of our approach. A theorem has a possible proof if it is not preempted by some other theorem. For example, successful proof to the Completeness proof obligation automatically implies that the Incompleteness and Inconsistency proof obligations cannot be proved. At the same time, successful proof of the Incompleteness proof obligation invalidates the Completeness proof obligation, but does not necessarily mean that the Inconsistency proof obligation should hold. It may or may not, depending on the model. Such cases are handled by the eighth column, where we list the number of (invalidated) theorems that are not provable (even by hand) in a given model. We are mainly interested in seeing how many of the provable (that is, possible minus invalidated) theorems are automatically proved. The fifth through seventh columns list the various causes (activation style employed, specification style employed, or presence of existential quantifiers) for why certain provable theorems could not be proved automatically. The specification style column also includes cases in which the problems are caused by deficiencies in the source specification language. The final column lists genuine bugs identified by the

Module Name	Comp/ Arch	Auto-mated	Pos-sible	Act. Style	Spec. Style	Exist. Quant.	Not Prov.	Bug
find	Comp	2	2					
sort	Comp	2	2					
bin_search	Comp	1	2	1				
find_arch	Arch	3	4		1			
AC	Comp	2	2					
comparator	Comp	2	2					
counter	Comp	2	2					
mux	Comp	2	2					
AC_arch	Arch	1	5		4			
PIP	Comp	2	2					
PulseDetector	Comp	1	3			1	1	
InterrogatorDecoder	Comp	1	2	1				
PulseGenerator	Comp	1	2	1				
PIP_arch	Arch	2	6	4				
CruiseControl	Comp	2	2					
SystemState	Comp	2	2					
CDS	Comp	2	2					
CTS	Comp	2	2					
CruiseControl_arch	Arch	4	6				1	1

Table 1: Results of applying automated proof obligations to various systems

```
%% PVS representation of architecture find_arch of
%% entity find
find_arch [Store: TYPE+, empty: Store,
           Component: TYPE+] : THEORY
BEGIN

IMPORTING jvsp@Architecture[Store, empty, Component]

...
%% Ports of the Higher Level Component
in1: [Store -> integer]
in2: [Store -> integer]
in3: [Store -> integer]
in4: [Store -> integer]
key: [Store -> integer]
output: [Store -> integer]

sig_out1: [Store -> integer]
sig_out2: [Store -> integer]
sig_out3: [Store -> integer]
sig_out4: [Store -> integer]

...
%% Variables used in theorems and axioms
pre, post, any: VAR Store
```

Figure 6: Partial PVS representation of `find` architecture, part 1

system.

There are four main factors that affect the provability of generated theorems. The most significant of these factors is the structure of the architecture. Linear architectures are the easiest to perform automated analysis. Branching architectures increase the complexity of generated theorems and require additional proof steps, such as proof by cases, in some cases. The most complicating structure is feedback, which has its greatest impact on bisimulation proofs. Systems with feedback require induction proofs, which, in the general case, cannot be automated. Therefore, systems with feedback cannot be automatically analyzed. A special case of systems with feedback is the use of local store by components. For all practical purposes, a system with a local store is similar to that same system generating outputs to a component that feeds the same values back to the original component during the next activation period. Therefore, systems with local store cannot be automatically analyzed.

The activation style employed plays a crucial role in the provability of the proofs. When using control-based activation, it becomes necessary to do an inductive proof over all traces of the system in order to identify the states in which a component can be activated. The validity of the pre-condition is checked in all such active states. On the other hand, when using data-based activation, the pre-condition can be verified directly from the activation condition, and so is automatable.

The third factor that affects the complexity of proofs is the specification style employed by the user to specify the user-defined operators in PVS. Specifications using *conservative extension* are more amenable to automated proofs as the prover can automatically expand relevant terms, whereas other specification styles require explicit introduction of relevant axioms into the proof process.

The final factor is the presence of existential quantifiers in the theorem proved. At present, there is no automated mechanism to provide correct instantiations to existential quantifiers. Therefore, while the semantics allow such proofs to be manually performed, they cannot be automated. However, one of the new engines being implemented for the PVS theorem prover is aimed at rectifying this deficiency. This engine, once fully implemented, is expected to find suitable instantiations (in most cases) if they exist. This engine would allow automated proofs to many more proof obligations than currently possible.

6. SUMMARY AND CONCLUSIONS

In this work, we presented an approach to using theorem proving for automatic analysis of abstract, component-based,

designs. It involved the use of two different languages, one that is easy for the designers to write specifications in, and another in which it is easy to prove theorems. For abstract designs, the VSPEC specification language was used, while the PVS theorem prover was used for proving properties about the design.

The two-language approach necessitated the writing of the semantics of the specification language in the theorem proving language and translating specifications into that semantics. A number of interesting properties generic to component-based hardware-like systems were identified. These analyze the interfaces and interconnections of an architecture with respect to a specification. Since interfaces and interconnections were identified as the sources of most errors in systems design and implementation, these were expected to have the most impact. Translation is a straightforward process, with one PVS theory being generated for each VSPEC entity. The various clauses are generated as axioms within the theory. The proof obligations for the analysis of interfaces and interconnections within the model are generated as theorems within the theory.

The final part of our approach involved the generation of proof steps to attempt automated proofs to the generated theorems. The proof steps were generated simultaneously with the generation of theorems. The proof steps make use of powerful proof macros provided by the PVS proof checker so as to make the proofs generic.

Our methodology was evaluated with the goal of identifying the conditions under which it is or is not effective. Four models were analyzed using our approach – the `Find` model, the `AlarmClock` model, the `PIP` model and the `CruiseControl` model. Most of the provable theorems were shown to be automatically proved using the generated proof scripts. A number of factors affecting the automatability of the proofs were also identified. These include structure of the architecture, activation style, specification style and presence of existential quantifiers.

In conclusion, our approach to the problem is promising. The use of theorem proving facilitates appropriate handling of abstraction during the early design stages, while the automation makes this approach easily applicable. Our use of design abstraction and small theorems makes our approach tractable.

7. REFERENCES

[1] Judy Crow, John Rushby, Natarajan Shankar, and Mandayan Srivas. *A Tutorial Introduction to PVS*. SRI International, Menlo Park, CA, June 1995. Presented at WIFT'95.

[2] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–77, 1978.

[3] Amitvikram Rajkhowa. Vspec constraints modeling, evaluation and verification. Master's thesis, University of Cincinnati, 1999.

[4] Ranga Vemuri, Ram Mandayam, and Vijay Meduri. Performance modeling using PDL. *Computer*, 29(4):44–53, April 1996.

A Component Oriented Notation for Behavioral Specification and Validation

Isabelle Ryl and Mireille Clerbout and Arnaud Bailly
L.I.F.L. CNRS UPRESA 8022
Bât M3, Cité Scientifique
59655 Villeneuve d'Ascq cedex
France
{ryl, clerbout, bailly}@lifl.fr

ABSTRACT

Component software development is definitely on a high trend in the sofware engineering field. However, integrating components which the producer does not have complete control over increases the risk of getting unexpected software behavior. So developing components for reuse by third-party integrators is a challenging task that one can make easier if the behavior of these software components is precisely specified.

In this paper, we introduce a specification language complementing the interface definition language IDL3 proposed by OMG to describe CORBA Component Model compliant components. This specification language is based on communication history : the sequence of observable events - method calls, return of method calls, events, exceptions – that occurred since the system has been started. It allows us to characterize the functional behavior of components by way of invariants : an interface invariant specifies a contract between a component that provides it and each of its clients, whereas a component invariant constraints the whole communication between one component and all its clients and servers. We propose a procedure based on this notation for generating specification-based test cases adapted to unit testing and discuss how to use this notation for validation purposes.

1. INTRODUCTION

The development of component based applications is clearly a necessity for software industry in a world of large scale distributed applications. Furthermore, the growth of a market of reusable components, the famous *Components off-the-shelf* – COTS – seems to be the only way to reduce production costs in a domain where constantly evolving technology checks productivity. Nevertheless, this approach makes sense if components are high-quality and really "composable". This can be achieved by strict development methods and interoperability norms. This last point is addressed by platforms like CORBA, EJB or DCOM.

The well-known advantages of a "component oriented" approach are the following :

- *reusability.* Components are "bricks" that are available for designers,
- *maintainability* . Functionalities can be added or modified just by insulating responsible components,
- *interoperabilty.* Components implementing a given specification can be searched amongst existing components offered by various providers on the COTS' market.

To reach these goals, the specification of components has to be precise enough to allow searching, testing and composing components which are potentially designed, implemented, integrated, and used by different people. The mere existence of a market itself depends on the confidence providers and consumers may put in the functional correctness of a product.

In "Component Software" [22], Clemens Szypersky proposes the following definition of components: "A Software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." This definition emphasizes the use of well specified interfaces to describe components because the interfaces have to be the only link between a component and its outer-world i.e. its clients and environment.

One way to correctly specify interfaces is to use specific interface description languages, like the OMG-IDL for CORBA objects and the new OMG-IDL3 for CORBA Component Model compliant components. IDL typically defines contracts between components using method signatures. Unfortunately IDL contracts are mostly syntactic and do not capture the semantic aspects of the contract. This leads to under-specified components and breaks the initial interoperability goal. As an example, we can mention the work of Ousmane Sy [21] who showed that five different implementations of the COS event service by CORBA framework

providers were neither inter-operable nor substitutable because of diverging interpretations of the OMG-IDL specification.

During the last few years, the interest in formal software development has grown steadily. Approaches based on formal methods provide the advantages of a precise and non-ambiguous specification language with a formal semantic. And, even if formal proofs are out of their scope, formal verification is possible and generally supported by tools.

Sofware quality should benefit from formal method based component development a various stages:

1. a formal specification language has to be rich enough to express the full semantics of components,
2. powerful tools have to be able to check the consistency of specifications, using for example type-checking. Higher level properties must be verified: it may be possible to control that a component respect the contracts of its required environment, to control that the implementation of a component conforms to its specification and that connected components are compatible.

To fulfill this requirements, we propose to increase the specification power of OMG-IDL3 from the OMG CORBA Component Model (CCM) with formal specifications. This increased specification will follow the component through its life-cycle and will serve as *lingua franca* for all the actors – designers or users – to describe the behavior of the component. We only focus on describing functional aspects and do not address non-functional requirements like quality of service or overall reliability.

To us, a component is a black box that offers services, requires a specified environment, and can emit or receive asynchronous events. Interfaces are seen as contracts between two components or, in a more practical point of view as communication channels (like the event channels) between components. A system is then a set of components which are connected to each other by interfaces and event channels, and we focus on communications between components. Our specification language allows us to express two kinds of property. The first one is the notion of protocol, that is to say the partial order in which the input and output communications have to occurred. The second one is the data level, i.e. the possibility to specify properties on the value of an input or output parameter.

To meet these requirements, we propose a specification language based on communication traces.

The basic events of the system we consider are the messages components exchange, which are supposed to be instantaneous and asynchronous:

- method calls between interfaces of components,
- return of method calls,
- exceptions between components,
- event sending,
- event receiving.

Note that internal method calls are not considered here.

The trace of a running component, system or sub-system is the sequence of all the events involving it that occurred since its creation. The purpose of the specification is to capture the set of all possible communication traces of a given system. Note that a lot of works deals with the trace notion in different contexts (for example CSP [10], trace theory [8], ...).

After a short introduction to the CCM abstract model and IDL3, Section 2 describes our specification language. Section 3, gives some ideas of validation activities and proposes a way to test components. After a short review of related works, we conclude with some possible further work.

2. HOW TO SPECIFY COMPONENTS?

As already said, we build upon the CORBA Component model. We insisted on the fact that the specification could increase the reliability of the component at several stages in its life. Therefore, the specification ought to be present every time it might be needed and this explains our choice of IDL3 as our ground language. It offers two advantages:

1. IDL3 already exists, is used by several tools, and is embedded in the component package,
2. the IDL3 language contains all the necessary syntactic definitions of components and interfaces that can be exploited for formal specification purposes.

In order to respect the IDL3 syntax, we just insert some formal properties of components as IDL3 comments. That way, the specification is embedded in the packaged component and do not interfere with existing tools. To enlighten our methodology, we will develop in the remainder of this article a small example: an electronic ballot system.

Example informal specification. *Each voter uses an electronic individual polling device that communicates with a central office which registers the votes. To simplify the writing, we just consider a referendum and some rules:*

- *voters are allowed to vote at most once,*
- *voters are allowed to read the results when the ballot is over,*
- *the central accepts votes until the closure, then rejects them,*
- *the central refuses to provide partial results before the closure,*
- *the central must announce the correct results !!!*

The first subsection presents the general CCM model, the second one briefly introduces the IDL3 syntax. The last one presents our formalism on the example.

2.1 Model

To formally specify software components, we first have to agree on the definition of a component. We try in this subsection to present the abstract model we use that is part of the CCM model. We are just a little bit more restrictive

than the CCM ; CCM components may support interfaces and components may communicate with other components outside the scope of interfaces: this is not allowed in our model where all the components must declare ports (interfaces) and communicate through these ports only.

Interfaces are used by developers as implementation contracts for components and by clients to interact with components at runtime. Interface definition contains attributes and method signatures and may be defined using multiple inheritance. An interface groups a – hopefully coherent – set of methods providing some services and, according to the connection mechanism of the CCM model, it becomes a communication channel for data exchange between components.

A **component** definition groups attributes and ports. Attributes are properties of components set at deployment time for configuration purpose. Attributes misuse may raise exceptions. Ports come in four flavors:

- *facets* are interfaces provided by the component and synchronously used by the clients,
- *receptacles* are interfaces used by the component in a synchronous mode,
- *event sinks* are interfaces provided by the component and asynchronously used by the clients,
- *event sources* are interfaces used by the component in an asynchronous mode.

Ports are used at deployment time and runtime to connect components together. According to this model, a component may be seen as an element of a system that offers a collection of services under the assumption that another collection of services is available. This approach makes component exchange easier. Note that components may be defined using single inheritance (in a Java style). Figure 1 shows the diagram (drawn from [15]) used to represent a component and its ports.

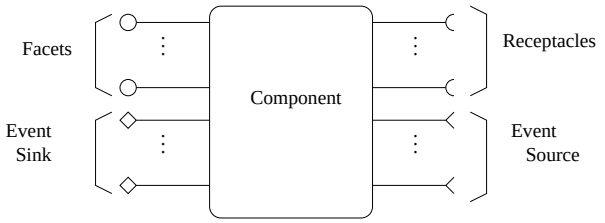


Figure 1: Component.

A **system** is a set of components. The current set of connections between components represents the system's configuration. Figure 2 presents our proposed model for the electronic vote example. Two types[1] of components are used, one for the votes collecting central office, **Center**, and one for the individual polling devices, **Electronic_box**. An **Electronic_box** component is used by each voter, it offers an interface *Electronic_vote* that allows the voter to send his vote or get the results. **Electronic_box** components are distributed and connected to the *Vote_Center* interface of a **Center** component that centralizes information. **Center**'s main role is to count votes, it also offers an interface *Vote_Admin* used to close the ballot. When the ballot is closed, the **Center** sends an event to inform connected components of the closure and transmits them results. The event sinks of **Electronic_box** components are connected to the event source of **Center**: the events sent via this source will be received by all connected components.

[1] For lack of space, we do not address the problem of identification of voters which can be solved using a classical "login/password" protocol.

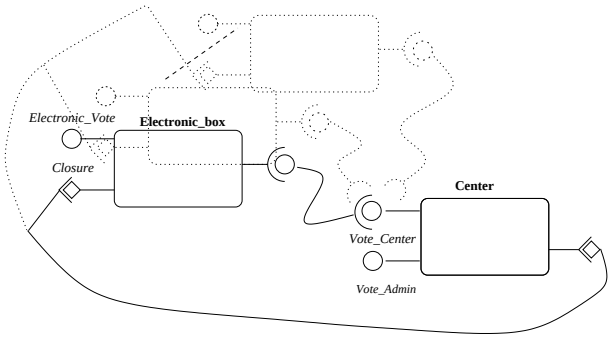


Figure 2: The example model.

2.2 IDL3 short presentation

The OMG IDL3 language extends IDL2 with some syntactic constructs to take into account the CORBA Component abstract model. This section shall introduce IDL3 concepts relevant to our example and is by no mean a full description of the CORBA Component Model. We refer the reader to [15] for more details about the CCM.

A part of the syntax is inherited from IDL2. Declarations are gathered in modules that delimit a domain name. Interfaces declare attributes and methods. Interfaces may be defined using multiple inheritance. Declarations are in a C/Java style, methods may have `in`, `out`, and `inout` parameters of simple types (`Short`, `Long`, `Float`, `Char`, `String`, `Boolean`, ...), complex types (`Enum`, `Struct`, `Array`, ...) or object types. Methods may also throw exceptions which may be defined by the user.

Figure 3 shows the IDL3 definitions of our example's interfaces. The module `Vote` starts by defining four exceptions (an exception may contain fields) followed by interface definitions. `Electronic_Vote` is the interface used by the voters, it offers two methods, one to vote and one to read the results. The first one has an in-parameter which represents the value of the vote (`true` for "yes" and `false` for "no") and it may throw two exceptions: `too_late` if somebody tries to vote after the closure and `already_voted` if somebody tries to vote twice. The method `read_results` has two out-parameters respectively returning the number of "yes" and "no" votes of the ballot. This method may also throw an exception when used before the closure (the results are not available). The interfaces `Vote_Center` and `Vote_Admin` are facets of the `Center`, they are respectively used by the `Electronic_Box` components and by an administrator. The `vote` method is used to transmit the elector's vote to the

center, it returns a boolean indicating if the vote has been taken into account or not (in case of closure for example). The method `close` allows an administrator to close the ballot and throws an exception if already closed. Comments are between `/**` and `*/` or `//` and end of line.

```
module Vote {
   exception too_late{};
   exception already_voted{};
   exception already_closed {};
   exception not_closed {};

   interface Electronic_Vote {
      void vote (in boolean choice)
               raises (too_late, already_voted);
      void read_results (out long yes,
                         out long no)
               raises (not_closed);
   /**
      invariant
         (h;_<-this.vote(_) |- H)
               => (!_<-this.vote(_) in h)
         &&
         (h;_<-this.vote<already_voted> |- H)
               => (_<-this.vote(_) in h)
         &&
         ((_<-this.vote<too_late> |
         _<-this.read_results (_,_));h in H) =>
               h/<- |- (_<-this.vote<too_late> |
               _<-this.read_results (_,_)))*
      */
   };

   interface Vote_Center {
      boolean vote (in boolean choice);
   };

   interface Vote_Admin {
      void close() raises (already_closed);
      /**
      invariant
         (h;_<-this.close() |- H) =>
               (!_<-this.close() in h)
      */
   };
   ...// insert here component declarations
};
```

Figure 3: Interface Specifications.

Components are introduced by the keyword `component`. As already said, components may contain attributes and ports definitions. Several keywords allow us to distinguish different kinds of ports: `provides` and `uses` for facets and receptacles, `consumes` for event sinks and `publishes` or `emits` respectively for *1-to-n* or *1-to-1* event sources.

Figures 4 and 5 complete our example's definition. Component `Center` has two facets and one *1-to-n* event source while component `Electronic_box` has one facet, one receptacle and one event sink. Each producing or consuming event port specifies a type of event: events are simply valuetypes (that is to say objects by value) that inherit from the `Components::EventBase` interface. In our example, a closure event has two attributes encoding results of the ballot: `yes_number` for count of "yes" and `no_number` for count of "no". Thus, an event has a meaning of its own (here indicating ballot's closure) but may also carry values (here results of the vote).

One can notice that all ports of a component are named and that a component may offer several ports of the same type. The (de-)connection operations are relevant facts in the component life, they consist in the assignment of real references to receptacles and sources of the component. Connection and deconnection operations are implicit methods of any component.

2.3 Specification language presentation

We have previously expressed our goals, and said that we use IDL3 as our components' definition language. It is now time to describe the formal specification language inserted as comments. As said earlier, we only focus (at least at this stage of our work) on the functional aspect of components, aiming to answer to these questions :

- How to describe a component's behavior ?
- How to ascertain that two components are interchangeable *i.e* they offer the same services?
- How to be sure that several components can be connected?

A system evolves from its creation to its destruction by way of components interactions. Thus, the "state" of a system (or component) may be seen as the result of its past interactions with its environment. So, the specification language we use is based on the concept of communication "history". This idea comes from an object oriented formal notation developed at Oslo University: OUN (Oslo University Notation)[17].

The communication history of a system is a sequence of observable events (a trace) that records all communications between components that occurred since the system has been started. The notion of "observable" event depends on the viewpoint we consider:

- observable events of a system are all the communications between components (method calls, returns of method calls, events, exceptions),
- observable events of a component are the events of the system that can be "seen" from the viewpoint of the component, that is to say all the communications whose sender or receiver is the component,
- observable events of an interface of a component are the communications involving the component through this interface.

Events of the system are supposed to be instantaneous, they may fall into four categories:

```
component Center {
   provides Vote_Center v;
   provides Vote_Admin a;
   publishes Closure c;
   /**
   invariant
         (h;_<-a.close() |- H) => (!_<-a.close() in h)
      && (h;_<-a.close<already_closed> |- H) => (_<-a.close() in h)
      && (h;_<-v.vote(_:true) |- H) => (!this->c[x,y] in h)
      && (h;_<-v.vote(_:false) |- H) => (this->c[x,y] in h)
      && (h;_<-a.close() |- H) => (h1;this->c[x,y] |- h && x=result(h1,true) && y=result(h1,false))

   functions
      result :  Trace, boolean -> int;
      result(empty,_) -> 0,
      result(h;_<-v.vote(x),x) -> result(h,x) +1,
      result(h;_, x) -> result(h).
   */
};
```

Figure 4: Component `Center`.

1. $c \rightarrow (c' : i).m(\bar{x})$ denotes a call of method m with the tuple of parameter values $\bar{x}$ initiated by component c, m is a method of facet i of component c'. Events belonging to this category are termed initiation events,
2. $c \leftarrow (c' : i).m(\bar{y} : z)$ denotes "normal" return of a previous event, the parameter values $\bar{y}$ may be different from $\bar{x}$ since out and inout parameter modes are allowed. The optional $: z$ value denotes possible return value of the method. Events belonging to this category are termed termination events,
3. $c \leftarrow (c' : i).m < e >$ denotes termination of a method call by an exception e. Those events are termed exception events,
4. $c \rightarrow \{(c_1 : s_1), \ldots, (c_n : s_n)\}[\bar{x}]$ denotes an asynchronous message, an event sent by component c to sinks $s_1, \ldots, s_n$ of components $c_1, \ldots, c_n$ respectively with tuple of values $\bar{x}$. Those events are termed asynchronous events.

Note that we do not characterize attributes and (de-)connection events: we consider attributes as pairs of set/get methods and (de-)connection events as method calls.

The set of possible events of the system is called the **alphabet of the system** (it depends on interfaces and components comprising the system). We can define in the same way alphabets for various elements of the system depending on what they can observe.

The **alphabet of a component** is the subset of events of the system whose receiver or sender is this component.

The **alphabet of a component seen through an interface** is the subset of the component alphabet containing events that are initiation, termination or exception of method calls defined in this interface.

A **trace** of a component (resp. a system) is a sequence of events of this component's (resp. system) alphabet in which a termination or exception event responds to a past initiation event. Notice that asynchronous events have no corresponding terminations. One may think of a trace as a sequence of events of a running system registered in the order they appeared since the start of the system until an arbitrary observation instant. Clearly, the system may run after the observation – a trace does not represent complete execution – and observation may occur at any time – a trace's prefix is also a trace.

A system element's specification describes its behavior in terms of possible communication traces. A component's specification is expressed as an invariant on traces (in first order logic). This invariant characterizes a subset of all possible traces over the component's alphabet hence capturing this component's semantic.

An **invariant** may be added to each interface or component. Informally, it is a first order formula with variables, constants, functions, predicates and constraints whose models are the valid traces of the component.

Our purpose is to convince the reader of expressiveness of such a notation and not to detail the syntactic sugar it offers, so we give an idea of the notation on the example.

Interface invariants in our example are shown in Figure 3. In our model, interface invariants are seen as *1-to-1* contracts, so they specify communications between a component implementing the interface, represented by the keyword `this`, and one of its clients, represented by the symbol `_` which denotes any value in the corresponding domain (here the set of components). This does not inhibit one-to-many communication schemes: a component may receive method calls from different clients through one of its interfaces but

```
component Electronic_Box{
   provides Electronic_Vote e;
   uses Vote_center v;
   consumes Closure c;
   /**
   invariant
         (h;_<-e.vote(x) |- H) => (this->v.vote(x);this<-v.vote(x:true) -| h )
      && (h;_<-e.vote<too_late> |- H) => (_->c[_,_] in h)
      && (h;_<-e.read_results(x,y) |- H ) => (_->c[x,y] in h)
      && (h;_<-e.read_results<not_closed> |- H) => (!_->c[_,_] in h))
   */
};
```

Figure 5: Component `Electronic_Box`.

the invariant of the interface specifies the communication pattern with each client. Clients are independent from each others thus the contract declared in each interface must ensure that clients can use some services whatever other clients may do. For example, if a file must be opened by a client before being read, a client following this rule must not be affected by another client trying to read this file without opening it.

Let us first detail the `Electronic_Vote` interface. The symbol `H` denotes the history, that is to say a solution to the formula. This invariant is a conjunction (denoted by `&&`) of three implications. The first one says that a vote ends correctly if it is the first one: if a sequence (denoted by `;`) of a trace `h` and the termination of a `vote` is a prefix (denoted by `|-`) of the history then `h` does not contain any termination of `vote` (`!` denotes the negation and `in` denotes "is a factor of"). The second part of the invariant describes the converse situation: a vote ends by exception `already_voted` if it is not the first one. The third part addresses another problem: votes are accepted before the closure and results are available after the closure. The closure may occur at any time, the client of the interface cannot detect it except if he receives a `too_late` exception or the results. This part of the invariant says that any part `h` of the history following an exception `too_late` in response to a vote or (denoted by `|`) a termination of `read_results`, only contains responses to method calls that are exceptions `too_late` or terminations of `read_results` (`/<-` denotes the projection[2] onto termination and exception events only).

Interfaces `Vote_Center` and `Vote_admin` have simpler specifications. There is no invariant in the first one since there is only one method whose use does not depend on anything observable in this interface. We could have said that as soon as the return value has been `false`, it remains `false` until the end (the vote is closed) but the user is free to specify properties or not if they seem not to be useful. The invariant of `Vote_Admin` just says that a closure operation ends normally if it is the first one: it is not possible to close the ballot twice. The invariant does not specify at what point the exception `already_closed` may occur because it is not decidable in the interface: a client cannot predict if another client has closed the vote before him.

In this example, we do not speak about initiation events: the component implementing the interface is not responsible for input events, it may only ensure its own outputs. Inputs events are often used to specify several situations of the kind: "if a client sends me this event before this one then this will happen ...".

Component invariants follows same syntax than interface invariants, but a component invariant specifies the whole communication pattern between this component and all other components, clients or servers: at this stage it is possible to specify the interleaving of its communications with several other components. The trace set of a component is described by its own invariant and the invariants of the interfaces it provides.

Definition. (Trace set) *Let us denote by Σ_c the alphabet of a component c and $f_1, \ldots, f_n$ its facets. The invariants of the component and the interfaces are respectively denoted by $\varphi, \varphi_1, \ldots, \varphi_n$. Then, the trace set of the component c is defined by:*

$$\{t \in \Sigma_c^* \mid \varphi(t) \wedge \forall i \in \{1, \ldots, n\}, \forall c' \bullet \varphi_i(t/f_i/c')\}$$

where c' denotes another component and $t/f_i/c'$ the projection of t over the events that are communications through the facet f_i with c'.

In other words, the trace set of c is the set of words defined on Σ_c that satisfy the invariant of c and the invariant of each facet of c with respect to a projection over communications with another component through this facet[2]. Note that there is a subtle difference between the alphabet's definition and the notation used: as soon as two components are connected, the receptacle contains the reference of the interface of the other component, so it may be used directly. Thus, we use the notation `c->r.m(`$\bar{x}$`)` instead of `c->(c':i).m(`$\bar{x}$`)` when the receptacle `r` of `c` is connected to the facet `i` of `c'`. Similarly, we do not denote the current component by `this` but by the name of the port involved in the communication, this allows us to easily distinguish communications on different ports.

[2]The projection of a trace t onto a set of events E can be seen as the operation that deletes in t all the events that do not belong to E.

Figure 4 gives the specification of the `Center` component, a conjunction of five implications. The first and the second ones say that the ballot may be closed at most once, otherwise an `already_closed` exception is thrown. This invariant differs from `Vote_Admin` interfaces' invariant because of the possible instantiations of mute symbol _: in the component invariant, it may represent any other component each time it appears. Thus we can say that a client calling the `close` method receives an exception if the termination of close has already occurred, whichever client has received this termination. The two following implications say that votes are accepted while the closure has not been announced and that they are rejected (return value `false`) as soon as the closure is announced. The last implication is a little bit different since it uses a function. The "`functions` part" of the specification is an auxiliary part that allows the user to define its own functions for specification purposes only. The definition of a function must start by the name of the function, and the parameter and return value types. The function itself is defined in a Prolog style by several clauses: the clause that will be used is chosen by unification on the heads of clauses in the order they are declared. Note that functions do not have side effects on traces or values passed to them. The `result` function calculates the result of the vote on a trace of the component: it takes two parameters, a trace and a boolean and it returns number of votes that appear in the trace with the boolean as parameter value. If the trace is empty, the result is 0 whichever boolean value is given as input. If the trace ends by a vote termination event whose parameter is the boolean which is currently counted, the result is the function applied to the beginning of the trace plus one. If the trace ends by any other event, this event does not affect the result. Let us return to the last part of the invariant. If a close event succeeds, then an asynchronous event has been sent with values `x` and `y` that are the correct results of the vote. This example shows how powerful is the specification language: it is possible to describe when calls or exception occur but also to precisely calculate parameters' values.

The form of the specification of the `Electronic_Box` component is very similar to the previous one, we only detail the first part of the conjunction in which three components are involved. It says that voters' choices are correctly transmitted to the center: each termination event of vote is immediately preceded (-| is read as "is a suffix of") in the trace by center's method `vote` initiation event (*i.e.* vote) with the same value of the vote `x` and matching termination with return value `true`. Thus, the vote has been transmitted and, since the three events are consecutive, we can deduce that there is one transmission for each vote.

To conclude on this example, the language allows us to express properties like protocols between several components as well as precise descriptions of parameter values and case when exceptions are thrown. The example of the function `result` shows that it is possible to calculate, using a function, an abstract state of a component from the trace. Our model supports synchronous (method calls) as well as asynchronous (events) communication. Components may be implemented using multi-threading, re-entering code and so on. Anything concerning the implementation is out of our scope so for example we cannot express the fact that a component must be multi-threaded, but our model supports it.

3. V&V ACTIVITIES

As said in the introduction, our project aims at providing tools to exploit the specification at different stages of the component's life. A formal specification provides a strong reasoning basis to deal with the system's properties. The specification is all the more useful as automated tools are provided. We propose to use the specification for different purposes: to validate the specification, to ensure that the component is conforming to the specification by way of testing methods, to check the composability and the substitutability of components. We expose in this section the different ideas, emphasizing the test phase which is the most advanced part of the work.

3.1 Validation

Specification Validation. The first step is to ensure that the specification satisfies the user's requirements. A specification written by a user is not systematically valid and may contains two kinds of errors:

- errors using the language concepts that may lead to inconsistency — *e.g.* an empty trace set or a trace set not closed under prefix,
- design errors that may lead to under specification.

In order to detect such errors, validation tools must be provided to check the specification. Clearly, the general problem of the validation of specifications is undecidable so any tool we provide will not be completely automatic.

To address this problem, we benefit from the work initiated in Oslo for the OUN notation [17]. Even if the two notations are not exactly the same (there are objects in OUN and components in our notation and it is syntactically more restrictive) the basic concepts are very close so we can use the same approach. The solution adopted in OUN is to use the tools offered by the PVS toolkit [18, 6]: PVS provides (among other things) a model-checker and a powerful theorem prover. The particularity of the prover is to allow the user to include proof strategies adapted to its own problems: this increases the automation of the proofs. The idea is to define the semantics of OUN in PVS in order to directly use the PVS tools, the work of [13] may be adapted to our notation.

Testing. As soon as the specification satisfies user's requirements, the problem is to obtain an implementation of the component that conforms to this specification. One approach is to generate code from specification, this produces a safe code but we rejected it for several reasons. First, the goal of the "component approach" is to free the developer from technical stuff to make him concentrate on business logic: specialists are recognized to write efficient code adapted to their domains. Second, our approach considers components as interchangeable "black boxes" and peculiarities of the code are out of our scope. As most of the software production lines do, we propose to use testing to check a component's correctness with respect to its specification. The test process is detailed in the next subsection.

Another problem is **composition**: how to be sure that two components are compatible ? Once components are shown to conform to their specifications, the compatibility of two components depends on the compatibility of their specifications. The effective connections of components are dynamical but port types may be used to statically check the correctness of the possible connections during the validation phase of the component (if there is one) or later during the assembly phase. It suffices to check that each component respects the contract of interfaces it uses and that each used interface provides the expected services[3]. For that, we have to check the compatibility of trace sets. A component c will behave correctly when assembled if (1) its traces relative to the viewpoint we consider (here by the way of projection) satisfy the invariant of the interfaces it uses and (2) if the used interfaces do not provide unexpected outputs (non-deterministic components may have several outputs for the same inputs). In other words:

Definition. (Connectable components) *Let c be a component and $\mathcal{T}_c$ be its trace set. The component c is "connectable" if and only if for each type I of receptacles or object parameters of c, for all component c' providing a facet of type I, and for all trace t of $\mathcal{T}_c/I/c'$:*

$$\varphi(t), \qquad (1)$$

$$\forall h \in \mathcal{T}_{c'}/I/c \bullet (h/ \rightarrow = t/ \rightarrow) \Rightarrow (h \in \mathcal{T}_c/I/c'). \qquad (2)$$

This definition is supposed to ensure compatibility of components. This is the case when receptacles and parameters used by a component are exactly of the declared type. The following question concerns sub-typing: what happens if we connect a receptacle of type I to a facet whose type is a sub-type of I? We expect all static verifications to remain valid whatever dynamic connections are made. For that, we use behavioral sub-typing: a subtype will behave like any of its super-types in the same context. Thus, the definition of connectable components we gave is valid even if sub-typing is used and we have a strong inheritance relation. "Behave like super-type" means for us that if we give the inputs of the super-type to the subtype, the subtype produces outputs that could have been produced by the super-type:

Definition.[4] **(Behavioral Interface Inheritance)** *Let I be an interface that inherits from $J_1, \ldots, J_n$. Let the formulas $\varphi_1, \ldots, \varphi_n$ be the invariants of $J_1, \ldots, J_n$ respectively. The inheritance relation of I is correct if and only if for all component c providing I and for all component c':*

$$\forall t \in \mathcal{T}_c/I/c', \forall i \in \{1, \ldots, n\}[(t/ \rightarrow = t/J_i/ \rightarrow) \Rightarrow \varphi_i(t)].$$

The correctness of the inheritance relations of interfaces should be proved during the validation of the specification. Note that component inheritance does not affect our verifications so it is not constrained.

The specification may also be useful at other stages in a system's life. We can for example evoke the maintenance: when can we say that two components are interchangeable ? Since our model pays a large attention to interfaces, we can assert that two components having the same ports are strictly **equivalent** (we still speak about functionalities). This relation may be too strong from a practical point of view: it is interesting to replace a component by another one, different but providing at least the same services in the same context, we say that they may be substituted:

Definition. (Substitutability) *A component c' may be substituted for a component c if:*

- *the services provided by c may be provided by c'* i.e. *for each facet of c, either c' provides a facet of the same type or it provides a facet of a behavioral subtype,*
- *the services required by c' are available* i.e. *receptacles of c' are of the same types or behavioral super-types as receptacles of c,*
- *c and c' have the same sources and sink.*

This definition works because of the behavioral inheritance relation we defined earlier on interfaces. In some sense it defines a kind of behavioral sub-typing for components.

3.2 Component testing

We have already explained the reasons which lead us to chose a testing method to verify correctness of a component towards its specification. In this subsection, we give some more details about the testing process. Defining a testing procedure requires to:

- select a representative set of test cases since exhaustive testing is not tractable,
- have an oracle, that is to say something (another program, a human, a specification, ...) able to decide if the program gives the right answer when executed with a test case,
- be able to execute the tests, that is to say run the tested program with the chosen test cases (inputs) and check (using the oracle) the program's outputs correcteness.

Our proposition takes place in functional, unitary, and specification-based black-box testing. The use of formal specifications for testing has several well-known advantages, and because of its form, the notation we propose has the following ones:

- we use it to generate test cases which are traces, thus we test a complete behavior involving several other components, several methods, client and server aspects of the component, and not a simple stimulus/response protocol,
- the specification is the oracle because traces contain inputs as well as outputs. Note that the specification language form is very different from programming languages, thus avoiding redundancy errors.

Test-case generation. As a first step, we have to generate a representative subset of a component trace set. The form of the specification introduces two kinds of problems: first

[3] As a matter of fact the same checks are necessary for each object parameter, the process is identical.

[4] Using this definition, specifications are not inherited. This choice gives some freedom in the sub-typing relations but may lead to excessive writing work. We could add a stronger inheritance relation using projection that would add more constraints on sub-typing but offers specification inheritance.

we have to generate sequences of events and secondly we have to find parameter values that satisfy the constraints.

The invariant is a formula: terms and logical connectives. First, we consider the formula from the viewpoint of propositional calculus where trace predicates are the atoms of the formula. By finding all combinations of predicates that satisfy this formula we are defining all possible behaviors of this component, which is close to the classic disjunctive normal form partitioning techniques from the test literature [7]. Each solution gives us formulas to obtain sets of test cases. The problem is then to generate these test cases. The terms generate languages that contain variables constrained by predicates. For example, the invariant of `Electronic_Box` is a conjunction of implications. The resolution gives us the disjunctive normal form we could have obtained by replacing each $A \Rightarrow B$ by $(A \wedge B) \vee (\neg A \wedge B) \vee (\neg A \wedge \neg B)$ and distributing. So, for example one term of the disjunction is given by Figure 6.

```
   (h;_<-e.vote(x) |- H)
∧ (this->v.vote(x);this<-v.vote(x:true) -| h )
∧ ¬ (h;_<-e.vote<too_late> |- H)
∧ ¬ (_->c[_,_] in h)
∧ ¬ (h;_<-e.read_results(x,y) |- H )
∧ ¬ (_->c[x,y] in h)
∧ ¬ (h;_<-e.read_results<not_closed> |- H)
∧ (!_->c[_,_] in h))
```

Figure 6: A sample conjunction of Invariant terms.

The solution we adopted to obtain the traces is to use PROLOG as introduced in [14]: operators on trace languages, predicates, functions defined by users (which are already in a PROLOG-like form), ... may be defined as PROLOG clauses. The goal of the program has one free variable which is `H` thus, it enumerates the traces. To obtain a tractable test case set, we have to define when to stop. We use the regularity hypothesis introduced in [3] which formalizes the following idea: "if we test all the test cases whose complexity is lower than the complexity of the formula, then we can consider that the formula is valid". The problem is to define the complexity to use. For the moment, we consider the number of operators allowing to generate the traces which is not fully satisfying. When the sequences are built, the last operation consists in instantiating parameter variables. Constraints on parameters are given by predicates and we use uniformity hypothesis introduced in [3] to select one representative value for each domain.

Test-case execution. We obtain a set of test cases which are traces, the problem is now to execute the traces. The abstraction level of our model gives us traces that are not directly executable. During the process development of components, the IDL3 description is projected in IDL2 and then some rules define projections onto different programming languages. So we have to apply the same projections rules to our traces to obtain the real traces that may be observed during the component execution.

The development of a test platform aimed at EJB is in progress. The basic idea is to use a test container. A component under test is placed in the test container which monitors the test executions: other components are represented by stubs that transmit messages to the container.

4. RELATED WORKS

There is now widespread acceptance over the necessity to specify software system and one acknowledges that having a precise and unambiguous formal specification available is a prerequisite in order to automate black-box testing.

Today there exist some proposals for specification languages designed for Interfaces Definition Languages : for example Larch/Corba [20] which is rather data-oriented due to its roots in Abstract Data Type or Borneo [19] where constraints do not consider data. Note that these two proposals only take into account server aspects of components.

In [5, 4] the authors extend general IDLs and then CORBA-IDL with protocol information concerning supported and required services using Milner's polyadic π-calculus which seems to be a more low level syntax language than ours and [2] uses a formalism based on Petri Net to specify CORBA component. The specification conformance with testing is not addressed in these works.

The generalized use of UML notation and its various behavioral formalisms (state-charts, collaboration and sequence diagrams) brings out several approaches (and tools) for test-case generation in object-oriented software. These works are based on techniques closed to finite state machines or finite labeled transition systems [9, 16, 12, 11].

Another class of approaches for test generation uses partition testing techniques [3, 7, 1]. Even if these techniques generally lean on model-based or algebraic specifications our project will benefit from these works.

5. CONCLUSIONS

We have presented in this paper a component oriented formal notation and some of the general definitions that can be exploited to validate specifications and to prove component properties. This work takes sense if tools are provided to specify and validate implemented components. The work is in progress, especially the test platform for EJB. Since Einar Broch Johnsen recently finished the definition of OUN semantics in PVS [13], we are now ready to exploit it to deal with validation aspects. Some other aspects could impact our work in the future. The first one is to address the general composition problem: how to build a component from components? In our context, the problem is to find a trace set for the assembly from the component trace sets, it relates to the problem of formal language reconstruction. The second one is to describe a component which an application needs and find it. This requires a specification of the application and to be able to extract from it a component specification (which is not the most difficult part). Clearly, the major problem would be to search a component and the automation of the search seems not to be tractable for the moment. Anyway, if a candidate component is found by a user, it is possible to compare its specification – if it has one – to the specification deduced from the application's or to test it – if it has no specification, in order to be sure that it is convenient for the application.

6. ACKNOWLEDGMENTS

We are grateful for feedback from discussions with the GOAL team, in particular Raphaël Marvie and Philippe Merle. Jean-Marc Geib has provided valuable detailed advice concerning this manuscript.

7. REFERENCES

[1] B. K. Aichernig. *Systematic Black-Box Testing of Computer-Based Systems through Formal Abstraction Techniques*. PhD thesis, Technischen Universität Graz, Germany, 2001.

[2] R. Bastide, O. Sy, and P. Palanque. Formal specification and prototyping of CORBA systems. In *Proc. ECOOP'99, Lisbon Portugal*, vol. 1628 of *LNCS*, pp. 474–494, 1999.

[3] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *IEEE Software Engineering Journal*, 6(6):387–405, 1991.

[4] C. Canal, L. Fuentes, J. Troya, and A. Vallecillo. Extending CORBA interfaces with pi-calculus for protocol compatibility. In *Proc. TOOLS Europe'2000), Mont Saint-Michel, France*, pp. 208–225. IEEE Computer Society Press, 2000.

[5] C. Canal, L. Fuentes, and A. Vallecillo. Extending IDLs with pi-calculus for protocol compatibility. In *Proc. ECOOP'99 Workshop Reader, ECOOP'99 Workshops, Panels, and Posters*, vol. 1743 of *LNCS*, pp. 5–6, 1999.

[6] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *Proc. Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, 1995.

[7] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proc. FME'93: Industrial-Strength Formal Methods*, vol. 670 of *LNCS*, pp. 268–284, 1993.

[8] V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, Singapore, 1995.

[9] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. In *Proc. ISSTA 2000*, pp. 60–70, Portland, Oregon, 2000.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[11] T. Jéron, J.-M. Jézéquel, and A. Le Guennec. Validation and test generation for object-oriented distributed software. In *Proc. PDSE'98, Kyoto, Japan*, 1998.

[12] J.-M. Jézéquel, A. L. Guennec, and F. Pennaneac'h. Validating distributed software modeled with the Unified Modeling Language. In *Proc. UML'98 - Beyond the Notation, Mulhouse, France.*, vol. 1618 of *LNCS*, pp. 365–377, 1998.

[13] E. B. Johnsen and O. Owe. A PVS proof environment for OUN. Research Report 295, Department of Informatics, University of Oslo, june 2001.

[14] B. Marre. *Une méthode et un outil d'assistance à la sélection de jeux de tests à partir de spécifications algébriques*. PhD thesis, Université de Paris-Sud – Orsay, 1991.

[15] R. Marvie and P. Merle. Corba Component Model: Discussion and use with OpenCCM. *Informatica*, submitted.

[16] A. J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proc. UML99, Fort Collins, CO*, pp. 416–429. IEEE Computer Society Press, 1999.

[17] O. Owe and I. Ryl. A notation for combining formal reasoning, object orientation and openness. RR 278, Department of Informatics, University of Oslo, 1999.

[18] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[19] S. Sankar. Introducing formal method to software engineers through OMG's CORBA environment and interface definition language. In *Proc. AMAST'96 Munich, Germany*, vol. 1101 of *LNCS*, pp. 52–61, 1996.

[20] G. Sivaprasad. Larch/CORBA: Specifying the behavior of CORBA-IDL interfaces. TR 95-27a, Department of Computer Science, Iowa State University, 1995.

[21] O. Sy. *Spécification comportementale de composants CORBA*. PhD thesis, Université de Toulouse I, 2001.

[22] C. Szyperski. *Component Software – Beyong Object Oriented Programming*. Addison-Wesley, 1998.

ACOEL on CORAL
A COmponent Requirement and Abstraction Language

An Extended Abstract

Vugranam C. Sreedhar
IBM TJ Watson Research Center
Hawthorne, NY 10532
sreedhar@watson.ibm.com

ABSTRACT

CORAL is a language for specifying properties of ACOEL, a component-oriented extensional language. The design of CORAL is based on input/output automata and type state. The properties of ACOEL components that need to be verified are specified using CORAL. A verification engine will then crawl through CORAL and verify whether ACOEL can be safely executed or not. In this paper we focus on CORAL, and show how to specify properties of ACOEL. We will also briefly discuss the concurrent modification problem that is commonly encountered in the iterator design pattern.

1. INTRODUCTION

The Internet has revolutionized the kinds of software applications that are currently being developed. These days people are talking about applications as services just as electricity and telephone services. When software are treated as services, it is important to ensure that they are properly packaged as components that can be easily connected to other software components, and it is even more important that (1) software components be certified that it will not do any harm to other components or the environment in which it is deployed, and (2) the clients will properly use the components. ACOEL is a component-oriented extensional language for creating and plugging components together [22, 23].[1] In ACOEL, a component developer can specify and abstract properties and requirements of components using CORAL (a COmponent Requirement and Abstraction Language). Depending on the context in which a component is used, a certification tool will try to certify that the component is well-behaved and is safe for plugging into the system.

[1] ACOEL was initially called as York.

Copyright 2000 ACM 0-89791-88-6/97/05 ..$5.00

In this paper we will mostly focus on CORAL.

There are two aspects to CORAL: abstraction and requirements. Abstraction essentially suppresses the irrelevant details of a component so that one can focus just on those properties that we wish to verify. There is definitely a compromise between abstraction and the level of details that one is interested in verifying. Requirements are constraints that are necessary for proper functioning of components. There are many different kinds of requirements that a component will want to enforce. For instance, a square root function $sqrt(x)$ will require that x is not a negative number. For proper functioning of a FTP component, it is required that a client first connects to a file server before getting files from the server. Some of the popular modeling and specification languages and tools in the literature include UML/OCL (Unified Modeling Language/Object Constraint Language) [10], JML (Java Modeling Language) [15], Larch [12], SMV [17], etc. Once the requirements of a component are specified using one of these languages, the underlying system will then encode the specification into a mathematical structure and then prove the required properties.

To ensure usability of an abstraction and specification language, it is important to maintain a close correspondence between the component concrete language and the language used for specifying abstraction/requirement of components. JML, for instance, is tailored to Java [15]. CORAL is a requirement and an abstraction language for expressing and proving properties of ACOEL components. A component in ACOEL consists of a set of typed input ports and output ports. The input ports of a component consists of all the services that the component will provide, while the output ports are all the services that the component require for correct functioning. A port type can be either an interface type or a delegate type. An interface type consists of a set of methods and named constants, whereas a delegate type is an encapsulated signature of a method. The internal implementation of a component in ACOEL is completely hidden from the clients (i.e., a black-box component). In CORAL, the set of input ports of a component are abstracted as a set of *input actions*, the set of output ports of a component are abstracted as a set of *output actions* and the internal implementations of a component

are abstracted as a set of *internal actions*. The states of a component, and of the environment are encoded using state variables and data types. The above actions when performed on a state will transform the state to another state. In CORAL such state transitions are expressed using a state transition relations. The CORAL model of an ACOEL component is very to close to an *input/output automaton* (IOA) [13].

The rest of the paper is organized as follows: Section 2 gives a brief introduction to ACOEL. Section 3 discusses IOA modeling of CORAL. Section 4 introduces CORAL using a simple example called the concurrent modification problem. Section 5 discusses some of the related work. Finally, Section 6 gives our conclusion and also projects some of the future research direction.

2. ACOEL

The design of ACOEL was motivated by the following component design principles.

- *Pluggable Units* A component is a unit of abstraction with clearly defined external contracts and the internal implementation should be encapsulated. The external contract should consist of both the services it provides and the requirements it needs when it is plugged or (re-)used in a system.
- *Late and Explicit Composition*. For a component to be composable by a third-party with other components, it must support *late* or *dynamic composition*. During the development phase, requirements of a component should only be constrained by some external contract. Then, at runtime, an explicit connection is made with other "compatible" components (i.e., one that satisfy the constraints) to effect late composition.
- *Types for Composition*. Typing essentially restricts the kinds of services (i.e., operations or messages) that can be requested from a component.
- *Restricted Inheritance*. In OO programming, it is well-known that one cannot achieve both true encapsulation and *unrestricted* class inheritance with overriding capabilities [21]. In ACOEL, classes (which support inheritance) are second-class citizens, and are not visible to the external clients.
- *No Global State*. In ACOEL, there are no global variables and public methods that are visible to the entire system.

Let us briefly illustrate ACOEL by implementing the Iterator design pattern [11]. An Iterator pattern consists of an aggregate (e.g., set, list, array, etc.) and an iterator that traverses the aggregate. The main construct in ACOEL is `component`. A `component` consists of a set of *typed* input ports and output ports. A `List` component, defined below, consists of two input ports: one port is used by the client code to add / get / remove list elements and for creating an iterator, and the other port is used by the iterator to add / remove / get list elements. A client uses the following type to access services from the `List` component.

```
interface CLIntf {
  void add(int index, Elem e) ;
  void remove(int index, Elem e) ;
  Elem get(int index) ;
  ListIter iterator() ;
}
```

An iterator component interacts with the `List` component using the following interface.

```
interface ILIntf {
  void remove(int index, Elem e) ;
  Elem get(int index) ;
  void start() ; // start of the iterator
  void end() ; // end of the iterator.
}
```

The `start()` method and `end()` are basically used to start and end an iteration, and `iterator()` is a factory method that returns an iterator component.

Next we define the `List` component.

```
component List {
  in CLIntf clin ;
  in ILIntf ilin ;
  ListNode head = null ;
  int count = 0 ;
  List(){head = null ; count =0 ;}
  class ListNode {
     Elem e ;
     ListNode n ;
     ListNode(){} ;

  }
  class CLCls implements CLIntf, ILIntf {
    void add(int index, Elem e) { ...};
    void remove(int index, Elem e) {...};
    Elem get(int index) {...} ;
    int count(){return count ;}
    ListIter iterator() {
      return new ListIter(This) ;
    }
    void start() { ...}
    void end() { ...}

  }
  attach clin to CLCls ;
  attach ilin to CLCls ;
}
```

The `attach` statement essentially attaches an input port to a particular implementation class inside the component. Any messages that arrive at an input port is forwarded to the instance of the class that is attached to the input port. The class instance will either process the message or it will delegate to another class instance inside the component.

Next we define the `ListIter` component. It consists of one input port and one output port. The output port `ilout` is used to connect to the input port `ilin` of `List`. A client component uses the input port `clin` for accessing services of the `ListIter`. First let us define the type `CIIntf` of input port `clin`.

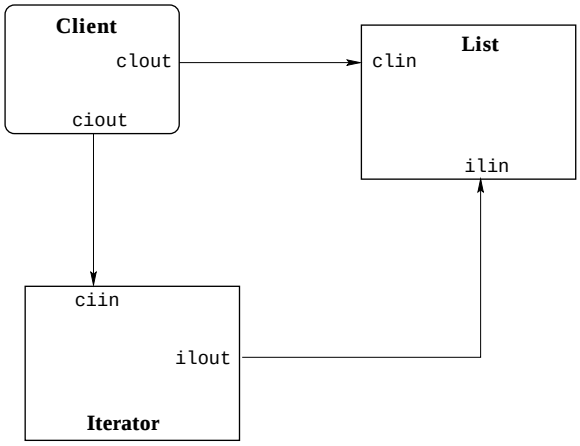


Figure 1: Various components in Iterator pattern

```
interface CIIntf {
  Elem next() ;
  Elem start() ;
  boolean hasNext() ;
  void remove() ;
}
```

A client component uses the `start()` method to start a new iteration. Here is the `ListIter` component for iterating over the elements of a list.

```
component ListIter {
  in CIIntf clin ;
  out ILIntf ilout ;
  ListIter(List l ) {
    connect ilout to l.ilin ;
    pos = 0 ;
  }
  int pos = 0 ;
  class ILCls implements ILIntf {
    Elem next() {
      Elem e = ilout.get(pos) ;
      pos++ ;
      return e ;
    } ;
    Elem start(){
      ilout.start() ;
      pos = 0 ;
    }
    boolean hasNext() {
       if (pos < ilout.count())
         return true ;
       ilout.end() ;
       return false ;
    } ;
    void remove() {...} ;
  }
  attach clin to ILCls ;
```

A client component has to first explicitly `connect` to a component before obtaining the services. Notice that `ListIterator` can invoke services of `List` component via its own output port `ilout`, and this output port is connected to input port `ilin` of `List` component instance.

Finally, here is a client code that wants to access the `List` component and the `ListIterator` component.

```
component Client {
  out CIIntf ciout ;
  out CLIntf clout ;
  main() {
    List l = new List() ;
    connect clout to l.clin ;
    // add a bunch  of elements ...
    ListIterator li = l.iterator() ;
    connect ciout to li.ciin ;
    ciout.start() ;
    while(ciout.hasNext() {
      Elem e = ciout.next() ;
    }
  }
```

Figure 2 shows the overall structure of the iterator pattern. There can be more than one iterator that is simultaneously active. A client will typically use the iterator created by the list component (through the factory method `iterator()`). The list component can ensure that it will only interact with iterators that it created for a client. Whenever there are multiple simultaneous iterators, there is a potential for concurrent modification of the list by multiple iterators (which will lead to inconsistent states). We will discuss this problem later in the paper.

3. MODELING COMPONENTS

A component in ACOEL consists of (1) an external contract made of typed input and output ports, and (2) an internal implementation consisting of classes, methods, and data fields. A client can only see the external contract and the internal implementation is completely encapsulated. A component provides services via its input ports, and specifies the services its requires via its output ports. In ACOEL, a `connect` statement makes an explicit connection between an output port of a component to a "compatible" input port of another component. Let $connect\ \breve{c}_1\langle p_o\rangle\ to\ \breve{c}_2\langle q_i\rangle$ be a connect statement. For this connection to be *compatible*, it is necessary that $q_i <: p_o$. The sub-type relation ensures that any message sent over the connection by $\breve{c}_1$ can be processed by $\breve{c}_2$. But the sub-type relation is not sufficient to ensure port compatibility. In ACOEL, we enforce other kinds of constraints using CORAL.

We use a framework that is similar to input/output automaton (IOA) to model ACOEL components. Abstractly, a component automaton (CA) consists of a set of actions, a set of state, and a set of transitions. The set of actions are classified as either *input actions* $in(A)$ (corresponding to messages arriving at input ports), *output actions* $out(A)$ (corresponding to the requirements at output ports), and *internal actions* $int(A)$ (corresponding to internal calls). Let $acts(A) = in(A) \cup out(A) \cup int(A)$.

Similar to IOA, a CA A consists of the following four components:

- $sig(A)$, a signature
- $states(A)$, a set of states (not necessarily finite)
- $start(A) \subseteq states(A)$, a set of start or initial states
- $trans(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$, a state-transition relation, such that for every state s and every input action π, $(s, \pi, s') \in trans(A)$.

An action π is enabled in a state s if $(s, \pi, s') \in trans(A)$. Input actions are enabled in every state (i.e., a component cannot block messages arriving at its input ports). This is not a big restriction, since almost always we can throw an error condition for messages that a component cannot handle (also, we can use the type system to ensure that no arbitrary message arrives at input ports of a component).

There are few differences between a regular IOA and the kinds of programs that we are dealing with in ACOEL. First, components in ACOEL can be dynamically created and destroyed. Also, each component has its own state. In ACOEL there are no global variables and methods. Since component instances are dynamically created and destroyed, an IOA model should include actions for creation and destruction of automaton and for modeling system of automaton. To model dynamic creation of components, we introduce a *create action* $crt(A)$ that corresponds to creation of an automaton A. The $crt(A)$ will also invoke the constructor function that modifies the state of A. The create action $crt(A)$ can be thought of as an input action to the newly created automaton A, and the input action will invoke the constructor methods of the corresponding component. The create action will be executed by another automaton for creating a new automaton. At any instance, only a finite set of automaton exists. We can think of a configuration $\mathcal{C}$ as a finite set $\{\langle A_1, s_1\rangle, \ldots \langle A_n, s_n\rangle\}$, where A_i, for $1 \leq i \leq n$, is automaton identifier and s_i is the state of A_i. An action π essentially changes a configuration $\mathcal{C}$ to a new configuration $\mathcal{C}'$, a create action will add a new automaton to $\mathcal{C}$, and all other action will simply change the states of existing automaton.

4. THE CORAL LANGUAGE

In this section we will briefly introduce CORAL using the iterator pattern example. Our intention is only to expose the core ideas behind CORAL. A component automaton (CA) consists of two main parts: (1) `states` and (2) `transitions`. The `states` part consists of a set of state variables, whose types can be either primitive or composite data types. Primitive data types include `char`, `string`, `int`, `float`, and reference type. Composite data types can be either in-built types or user-defined types. In a `types` part one can define new data types (see Figure 2). The `transitions` part consists of a set of state transition written in the style of *pre-condition-effect-error* for each action. This is illustrated in `ListAutomaton`, a CORAL automaton for the `List` component (see Figure 2). For each action, we list the `pre`-condition part, the `eff`ect part, and the `error` part. Whenever the pre-condition part is satisfied, the `eff` part is executed otherwise the `error` part (if defined) is

```
coral ListAutomaton {
  types:
    Iter {
       int id ;
       enum st = {active, passive} ;
    }
  states:
    int srcId ;
    List l ; // list type
    Iter iter[] ; // a hash of iterators.
  transitions:
    input void CLIntf.add (int index, Elem e) {
      pre:
        (forall i iter[i].st==passive) &&
          (l.length < index)
      eff:
         l.insert(index, e) ;
      error:
          throw AddException ;
    }
    input void CLIntf.remove(int index, Elem e) {
      pre:
        (forall i iter[i].st==passive) &&
          (l.length < index)
      eff:
        l.remove(index, e) ;
      error:
        throw RemoveException ;
    }

    input Elem CLIntf.get (int index) {
      pre:
          (l.length < index)
      eff:
      error:
        throw GetException ;
    }
    input CLIntf.iterator () {
      pre:
      eff:
        // add a new iterator to iter
 int newsrcId = create ListIterator
         iter.add(newsrcId) ;
 return newsrcId ;
    }
    input ILIntf.start() {
      pre:
      eff:
        iter[srcId].st = active ;
    }
    input ILIntf.end() {
      pre:
      eff:
        iter[srcId].st = passive ;
    }
    input Elem ILIntf.get (int index) {
      pre:
         (iter[srcId] == active)
         (l.length < index)
      eff:
      error:
        throw GetException ;

    }
    input ILIntf.remove() {
       pre:
         (iter[srcId] == active) &&
         (forall i  and i!=srcId
           {iter[i].st==passive} ) &&
         (l.length < index)
       eff:
         l.remove(index, e) ;
       error:
         throw RemoveException ;
    }
}
```

Figure 2: CORAL for `List` component.

executed. The `eff` part essentially performs state transformations.

For the example in Figure 2, the `states` part consists of three states: `srcId` is the identity of the source component that is invoking the input action. `l` is a list with operations such `insert`, `remove`, etc. An `insert` operation will add an element to `l` and changes the state of `l` to a new `l`. The `iter` state keeps track of all iterators that a client created. A `create` operation will essentially create a new iterator identity and saves it in `iter`. Consider the input action `CLIntf.add()`, the `eff` part will be executed only if all the iterators in `iter[]` are passive and the `index` is less than the length of the list. Otherwise the `error` part is executed.

We essentially translate a CA to an IOA, and then verify properties in IOA. An input action in CA also returns a value (which can either a normal value or an error condition). So an input action in CA is translated into a input action followed by an output action in IOA. The purpose of the output action is to return a value or an error condition back to source component. We do the same for an output action in CA (i.e., it is also broken into an output action followed by an input action).

Unlike in IOA, in CA we typically do not perform composition operation explicitly—we typically verify whether a composition is a valid composition, and the actual composition is effected by subtype relation between ports via `connect` statement. There are two kinds of verification we are interested: *invariance* and *reaching an error state*. An invariance is a property that is true in all reachable states. Reaching an error state means that a pre-condition fails and an "error" state is reached. An execution of an automaton is a finite sequence of $s_o, \pi_1, \ldots, \pi_n, s_n$, with s_0 being a start state of the automaton. A state is reachable if it occurs in some execution. Our main goal is verification of safety properties (rather than liveness or fairness properties).

Let us briefly illustrate one kind of verification problem, called concurrent modification problem (CMP). This problem was motivated from Ramalingam et al. [18]. We have simplified the problem from what is described in Ramalingam et al. [18]. The main problem with CMP is that when an iterator is active a modification to the underlying aggregate structure can cause an inconsistency between the iterator and aggregate structure. Most implementation of an iterator pattern will allow modification to an aggregate structure only through the iterator (especially when an iterator is active). Let us slightly modify the client code given in Section 2 and include the statement `l.add(0,e)` in the while-loop.

```
component Client {
  out CIIntf ciout ;
  out CLIntf clout ;
  main() {
    List l = new List() ;
    connect clout to l.clin ;
    // add a bunch  of elements ...
    ListIterator li = l.iterator() ;
    connect ciout to li.ciin ;
    while(ciout.hasNext() {
      Elem e = ciout.next() ;
      l.add(0,e) ;
    }
  }
```

In the `ListAutomaton` the `pre`condition for `l.add` will fail since an iterator in `iter` state *may* still be active. Although the above example looks trivial there are many non-trivial phases that one has to go through before coming to the conclusion. For instance, we need alias analysis information to disambiguate different iterators. We need to use theorem proving techniques to verify invariants defined in the pre-conditions. We have used `end()` method to explicitly terminate an iterator. Compared to Ramalingam et al., our approach gives very conservative result. It is to be noted that our intention in using CMP is only to illustrate the use of IOA for verifying this, albeit simplified, problem.

5. DISCUSSION AND RELATED WORK

Verifying software system is an age-old, but certainly not a solved problem. Many specification and verification techniques have been proposed in the literature for ensuring that software systems are safe and well-behaved [2, 14, 24, 3, 12, 7, 20]. With the advent of the Internet-based applications it is even more important to ensure safety and security of software system. In this paper we presented CORAL for abstracting and specifying requirements of ACOEL components. We used IOA for modeling ACOEL components. Typically in the past, IOA has been used to model distributed system. In CORAL we use IOA to verify whether a component when plugged into a system will behave correctly, and also whether a client of the component will use the component correctly or not. CORAL can be used to verify other kinds of constraints such a protocol verification [25]. We can simply encode the correct sequences of method calls using an automaton.

This paper presents a preliminary experience of using IOA for software verification. There are many open-ended problems that needs to be resolved. Handling aliasing, sub-type polymorphism, etc. presents some interesting challenges. Recently Attie and Lynch proposed dynamic IOA that can handle dynamic creation and destruction of automaton. Rather than thinking in terms of single automaton, dynamic IOA goes one step further and defines a configuration of interacting automata [4]. We are currently exploring on how to use the full potential of dynamic IOA in CORAL. For verification purposes we have to deal with practical programming languages which typically include aliasing and polymorphism. Unlike IOA, our main goal is verification of components. One component can be connected to another component through their ports if the corresponding port types have a sub-type relation (i.e., the input port should be a subtype of the output port). We use CORAL to go beyond subtype relation and verify other kinds of constraints [16].

Model checking is a classical approach to verification of software systems [8]. Bandera is a collection of tools for model-checking concurrent Java programs [9]. It takes Java source code, compiles them, and generates code for verification tools like SMV and SPIN. SLAM

project is very similar to Bandera project, except that SLAM also uses predicate abstraction and discovery to point errors in C code [5]. Strix is specification language for expressing business process and a Strix compiler once generates code for SMV model checker [6]. CANVAS uses EASL specification and translate them to a 3-valued logic for verifying program properties [18]. JML is a Java Modeling Language and it uses design-by-contract and Larch theorem prover to verify program properties [15]. There are several other projects related to software verification.

Another important, but related, area is the Architecture Description Language (ADL) [19]. A software system is typically starts off with a requirement and a design phase. During this phase, the implementation details are typically ignored and the focus is on understanding and developing software architecture. ADLs are typically used at this phase to specify the structure and the requirements of a software system. ArchJava is an example of integrating ADL with Java [1]. CORAL can be used as a ADL. One can express the requirements of ACOEL components, even before implementing them using CORAL. To use as an ADL, we need a way to compose component automaton. For this we rely on IOA theory of composing automaton.

6. CONCLUSION

In this paper we briefly introduced CORAL as a language for abstracting and specifying ACOEL components. CORAL is based on IOA. Unlike classical IOA, our intention in using the theory of IOA is for verification of software components. We are currently working on three aspects of CORAL. First we are refining on the syntax and semantics of CORAL. Second, we are focusing on the dynamic IOA model for CORAL. Finally, we are looking at ways to model aliasing, sub-typing, classes, and other states within IOA. Both ACOEL and CORAL are at design stages, and we are at initial stages of implementation. We expect to publish more details of CORAL in the near future.

Acknowledgement

I thank Deepak Goyal for valuable discussions and comments on an earlier draft of the paper.

7. REFERENCES

[1] Jonathan Aldrich and Craig Chambers. ArchJava: connecting software architecture to implmentation. Technical Report UW-CSE-01-08-01, Univ. of Washington, August 2001.

[2] Dean Allemang. Extending the applicability of formal verification techniques. In Gary T. Leavens and Murali Sitaraman, editors, *Proceedings of the First Workshop on the Foundations of Component-Based Systems, Zurich, Switzerland, September 26 1997*, pages 1–10, September 1997.

[3] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, June 1997.

[4] Paul C. Attie and Nancy A. Lynch. Dynamic input/output automata: a formal model for dynamic systems. In *CONCUR'01: 12th International Conference on Concurrency Theory*, LNCS. Springler-Verlag, 2001.

[5] T. Ball and S. Rajamani. Checking temporal properties of software with boolean programs. In *Proceedings of the Workshop on Advances in Verification*, 2000.

[6] B. Bloom. Seeing by owl-light:Symbolic model checking of business application requirements. Technical Report ????, IBM T.J. Watson Research Center, 2001.

[7] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[8] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[9] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[10] Martin Fowler and Kendall Scot. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, New York, NY, 1995.

[12] S. Garland, J. Guttag, and J. Horning. An overview of Larch. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 329–348. Springer-Verlag Lecture Notes in Computer Science 693, 1993.

[13] S. Garland and N. Lynch. Using i/o automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.

[14] M. Goedicke, H. Schumann, and J. Cramer. On the specification of software components. In Jean-Pierre Finance, editor, *Proceedings of the 6th International Workshop on Software Specification and Design*, pages 166–174, Como, Italy, October 1991. IEEE Computer Society Press.

[15] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, 1998.

[16] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. and Sys.*, 16(1):1811–1841, November 1994.

[17] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts,

1993.

[18] G. Ramalingam, A. Warshavsky, J. Field, and M. Sagiv. Deriving specialized heap analyses for verifying component-client conformance. Technical Report RC22145, IBM T.J. Watson Research Center, August 2001.

[19] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[20] Murali Sitaraman, Lonnie R. Welch, and Douglas E. Harms. On specification of reusable software components. *International Journal of Software Engineering and Knowledege Engineering*, 3(2):207–229, 1993.

[21] Alan Snyder. Inheritance and the development of encapsulated software components. In Bruce Shriver and Peter Wegner, editors, *Workshop on Object-Oriented Programming*, pages 165–188, Camebridge, MA, 1987. MIT Press.

[22] Vugranam C. Sreedhar. ACOEL: A component-oriented extensional language. Technical report, IBM T.J. Watson Research Center, 2001.

[23] Vugranam C. Sreedhar. York: Programming software components. In *Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2001. Poster session.

[24] D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

[25] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.

Non-Functional Requirements in a Component Model for Embedded Systems

Position Paper

Roel Wuyts
Software Composition Group
Institut für Informatik
Universität Bern, Switzerland
roel.wuyts@iam.unibe.ch

Stéphane Ducasse
Software Composition Group
Institut für Informatik
Universität Bern, Switzerland
ducasse@iam.unibe.ch

ABSTRACT

In this paper we describe an interesting context to study formal methods for component systems: embedded devices. The context of embedded devices is highly constrained by the physical requirements the devices have to adhere to. As a result, component models for embedded devices are not general purpose but geared towards these constrained contexts. In this paper we give the concrete setting of the Pecos project (a project with as goal component engineering for embedded devices). We describe the Pecos component model, and show possibilities where we think formal verification could be useful. We would like to use this as a very concrete example to discuss formal verification techniques.

1. INTRODUCTION

Software for embedded systems is typically monolithic and platform-dependent. These systems are hard to maintain, upgrade and customise, and they are almost impossible to port to other platforms. Component-based software engineering would bring a number of advantages to the embedded systems world such as fast development times, the ability to secure investments through re-use of existing components, and the ability for domain experts to interactively compose sophisticated embedded systems software [7].

The goal of the PECOS (PErvasive COmponent Systems) project (a European Esprit project) is to find solutions for component oriented development (COD) for embedded systems . In this context we are developping Comes (a general Component Meta-Model) and the Pecos Model (a specialization of Comes targeted towards embedded systems in the context of the project).

In Comes, components are black-box encapsulations of behavior. They have interfaces that consist of *properties* and *ports*, can contain subcomponents, and have consistency rules that express structural integration internal to the component (for example, to check dependencies between properties). The ports of components are connected by explicit *connectors*. Consistency rules of the composite component can reason about the properties of the composite, but also on the connectors and the properties of the sub components.

The Pecos Model is a specialization of Comes to explicitly support components for embedded devices in the context of the Pecos project. Interesting is that this puts a lot of extra constraints on the component model. We firmly believe that this will allow us to use formal (mathematical) techniques to verify non-functional requirements of the modeled components. More specifically, we want support regarding *timing and scheduling* and *memory consumption*.

For this workshop we see ourselves as the providers of an interesting problem and problem context. We believe that the extra constraints imposed by the context of our component model make it a good example to use and assess the functionality of formal techniques. Coming from a practical, less formal discipline of software engineering and programming language design, we want to discuss with the more mathematically enclined researchers on how to come to formal support for specifying and checking components.

In the rest of the paper we introduce the specific problem context of embedded systems in more detail. Then we show the current version of the component model. Finally we enumerate the places where we think that formal techniques could help us, and discuss techniques we think are of interest to us.

2. THE EMBEDDED SYSTEMS CONTEXT

A massive shift is going from desktop applications to embedded systems, where intelligent devices taking over roles that are currently done in desktop applications. Moreover, the capabilities of embedded devices augment rapidly, and their responsibilities increase likewise. Distributed embedded devices (intelligent field devices, smart sensors) not only acquire but also pre-process data and run more and more sophisticated application programs (control functions, self-diagnostics, etc.).

The drawback of this evolution is that the software needs

to follow. Here the story is less positive: the software engineering techniques that are typically employed are lacking far behind software engineering techniques for mainstream applications. Currently software for embedded devices is written in assembly or C, in a monolithic fashion, with a typical development time of two to three years. The reasons for this are two-fold. The first reason is the specific context of embedded devices (with all the constraints of power comsumption and simple hardware as a result of this). The second reason is that, up until a couple of years ago, the market for embedded devices was relatively small, and was thus neglected by the big players from desktop applications. For example, operating systems or development environments are hard to find for embedded systems.

The goal of the PECOS (PErvasive COmponent Systems) project is to apply solutions for component oriented development (COD) in the context of embedded systems. As in desktop applications, the overall goal is to have more reuse, higher quality and reduced development time. Key factor in the project is the component model to support components for embedded systems. Before we have a look at this model, we first introduce the Pecos component development process, and field devices, the embedded systems the Pecos model should support.

2.1 Pecos Process

Part of the solution of the Pecos project is a component development process. In this section we give a quick overview of this process, as this will help to introduce some choices made in the Pecos Component Model. The process consists of two main phases: the *component construction phase* and the *field device assembly* phase.

The component construction phase defines what is needed to develop a single component (that possibly contains subcomponents), instrument it (to provide information about runtime aspects of the component), and put it in the component repository. It specifies the following workflow:

- the component is created. This means defining the basic properties and the interface of the component.
- the subcomponents are filled in. If the component has subcomponents, then these subcomponents need to be selected from the repository and added to the componenent. They also need to be connected with each other.
- the component is checked. In this phase, a structural check is performed to make sure that everything is specified according to the model, and that the given information follows the rules in the model. For example, when the model specifies that a component should have a name, then this is checked at this moment. Also, when type information needs to be given it is checked that the given types exist. Or, if there are subcomponents, their connections are checked.
- generating skeleton code. When the check succeeds, meaning that the component's structure is verified, skeleton code can be generated.
- filling in the skeleton: the skeleton code has to be extended into a full working implementation.
- instrumenting the component: the component is then ready to be instrumented. In this phase it is deployed in a standard environment so that certain runtime information can be gathered. What information depends on the model. Since in the Pecos model we want to check scheduling information and memory consumption, basic figures need to be extracted. Note that we need the instrumentation because we see components as black-box abstractions where we have no idea about their internals. If this constraint is lifted, the instrumentation phase could be made simpler or even omitted. We discuss this in more detail when we discuss the non-functional checks.
- the instrumented component is then added to the component repository.

A second activity is to assemble components into field devices (the actual embedded systems that need to be modeled in the context of Pecos). This activity consists of the following steps:

- select a template for the field device that needs to be created
- select the components that need to be filled in to instantiate the field device
- connect the components
- perform structural checks on the instantiated field device
- perform non-functional checks using the information provided by the components. For example, make sure that the total power consumption of the chosen components does not exceed the limit of the Field Device, or that a schedule can be found to schedule the components.
- generate the code for the field device
- deploy the component on the actual hardware

In the next section we have a look at Field Devices, the actuall embedded systems used in the project. Then we introduce the model to support the specification and checking of these devices.

2.2 Field Devices

Field devices are embedded reactive systems. A field device can analyze temperature, pressure, and flow, and control some actuators, positioners of valves or other motors. Field devices impose certain specific physical constraints For example a TZID (a pneumatic positioner) works under the following very hard constraint: the available power is only 100 mW for the whole device. This limits severely the available CPU and memory resources. The TZID uses a 16 bit micro-controller with 256k ROM and 20k RAM (on-chip), and communicates using fieldbus communication stacks (an

interoperability standard for communication between field devices). The device has a static software configuration, i.e., the firmware is updated/replaced completely, and there is no dynamic loadable functionality.

As a result from the physical constraints (especially the very harsh power consumption requirements), the runtime environment and the software are subject to the following constraints:

- One processor: all the components composing a field device are running on a single processor, that is very slow when compared to mainstream processors.
- One monolithic piece of code: after assembling the different components that compose a field device, the software for the field device forms one single piece that is deployed.
- No dynamic change: At run-time (after the field device is initialized) there is no memory allocation, nor dynamic reconfiguration.
- Single language per application: a component is created in a single language like C or C++.
- Multi-threading: field device components can be running on different threads. The scheduling is carried out by either the OS or by an explicit scheduler. However, most of the components are passive and scheduled by a central scheduler. Components that are active (that have their own thread) are typically the ones close to the hardware. They are responsible for regularly reading values from this hardware, such as the current motor position or speed.
- Components communicate by sharing data contained in a blackboard-like structure. Components read and write data they want to communicate to this central memory location.
- Some components are described by state automata. Some components have state, others are stateless because they are only representing algorithms.
- Components only offer interfaces in terms of in/out ports. The component state automata definition, or other behavorial descriptions, are not available. This is a very hard requirement, as this means that a lot of existing formal verification techniques are not usable.
- A field device architecture is fixed. It is composed by an *Analog component* controlling the overall workings of the device, a *Transducer component* that interfaces to the hardware, a *HMI component* for the Human-Machine interaction and an *EEPROM component* to store data in non-volatile memory.

3. THE PECOS COMPONENT MODEL

The Pecos Component Model is the foundation of the Pecos project. Its goal is to allow to specify and check components and Field Devices, given the constraints given above. In this section we iterate over the requirements for the model, introduce its main aspects. In the next section we then look at how formal techniques could be applied in this context.

3.1 Requirements

The goal of the Pecos Component Model is to be able to model and check a field device. More specifically, it has to allow:

- to specify individual components (that can contain subcomponents);
- to connect components;
- to assemble components into Field Devices;
- to check the structure and well-formedness of component compositions and Field Devices;
- to check non-functional requirements of Field Devices. More specifically, *timing and scheduling* of components, and their *memory consumption*;

3.2 Model Overview

In the constraints imposed by the context of embedded systems on field devices we already saw that Field Devices follow a blackboard-like achitecture. Hence, there is a central block of memory (called the *Object Manager*, or OM for short) that holds all the values that need to be passed between components in a field device. The OM is filled when the field device is initializad. At runtime, its structure does not change (as there is no allocation at runtime after the initialization). Components that need to share data to do by writing and reading from the OM.

Normally, when components would all be running in their own thread and hence in parallel, locking and synchronization of the OM would certainly be needed. However, in the specific context of a field device such a solution,(typical solution for desktop applications), is not possible. The reason is that it's too expensive in both processing power and memory consumption, and that OS facilities to support locking and synchronization are not always available or very costly. Field devices solve the problem by providing one central scheduler that sequentially schedules all components. Hence, at any moment in time, only one component has access to the OM and thus no locking is needed. Of course, this introduces other problems as well, that we will discuss in detail later on when we talk about supporting (checking) non-functional requirements.

The Pecos model builds on our experiences with supporting Software Architectures using logic programming languages [6]. The main constituents are components, ports and connectors:

- *component*: a Pecos component has a name, contains information regarding scheduling and memory consumption (see further), has a list of data ports and possibly has a list of subcomponents and connectors for these subcomponents;
- *data ports*: a data port indicates that the component provides or needs data for other components. It contains a type (of the data that will be passed, such as Float), a direction (in, out or inout),

- *connectors* connect data ports of components, and hence model a data dependency between two ports. Connectors contain the names of the component and the ports they connect

Besides this structural information, we also check some Pecos specific constraints, such as type and range information on ports. Table 1 lists all the structural checks that can be performed.

Besides the components and connectors, the Pecos model also offers a Field Device template. This is a template component that has to be instantiated with 4 concrete components. The Field Device component specifies the structure and the behaviour of a field device in such a way that its structure and semantics can be checked, and that code can be generated from it. To instantiate the field devoce, four components and their connections that have to be specified:

- *Human Machine Interface Component*: a field-device can be equipped with displays and other devices so that users can inspect or modify the behaviour from the device itself
- *Non-volatile memory Component*: the state of the component needs to be written to certain kinds of memory
- *Input-Output-Controller Component*: the data from the device component typically consist of raw values that are immediately related to the hardware contained. The function of this component is to provide an interface to the other non-hardware related components that is not hardware specific. For example, it can scale raw data from the hardware so that the display can show the value of a temperature controller in degrees Celsius.
- *Device Component*: all components that deal with the hardware are encapsulated by this component.

The result is a Field Device that can be checked for well-formedness (making sure that everytthing conforms to the structural rules) and for non-functional requirements. These last checks are the topic of the following section.

4. CHECKING OF NON-FUNCTIONAL REQUIREMENTS

The previous sections described the context of field devices and the Pecos component model to model components for field devices. However, it didn't give much information about the checking of non-functional requirements. In this section we describe what we would like to support, and what we are currently doing. We also give information about related formal work that we think could be useful (but that we not use at the moment of writing).

In the Pecos project we want to support two issues, that we have already touched upon throughout the paper: scheduling of components and memory consumption. We explain these two issues in more detail, and then have a look at opportunities we see for formal verification.

4.1 Component Scheduling

We already explained that in field devices we do not want to use regular locking of data, but instead want to schedule the components sequentially such that this is not needed. Hence, a very important aspect that needs to be checked when a field device component is instantiated is the scheduler.

More specifically, we currently instrument every individual component with information regarding its *execution time* (the time it takes to execute its behaviour once) and with information about its *cycletime*(the number of times it needs to be executed in one scheduler cycle). Using this information (combined with the information of the data dependency provided by the connectors) we are now investigating whether it is possible to derive or check a scheduler. The hardest thing to solve is that we currently identified three kinds of components: passive components, active components and event components. Passive components are straightforward to handle: they just need to be scheduled by the scheduler such that their execution and cycling information is met. Active components are more difficult. The reason is that they have their own thread that is running inside of the component. This thread is typically used to read-out values directly from hardware, such as the current speed of a motor. In the current implementation used in field devices, these values write to internal fields in the component, and when the component is scheduled the values in the internal fields are copied to the OM. Hence, active components are scheduled and handled exactly as passive components, even though they have their own thread. We are currently debating whether this is a good solution, and what would be alternatives. Event components pose the same problems as active components. They do not have their own thread, but act as event sinks that have to capture and react to events sent by certain pieces of hardware. Just as with active components, they capture an event, wait until they are scheduled by the scheduler and then handle the event.

At the moment of writing we are still investigating possible solutions to check and generate the scheduler, with probably the most interesting option to express all the scheduler constraints using Constraint Logic Programming over Real Numbers (CLP(R)), and calculate possible schedules. By the time of the workshop we will have a concrete solution for this problem, as this is currently under full development.

4.2 Memory consumption

Due to the minimal memory available in field devices, the memory occupied by a component is a crucial information. The model should support the computation of the component size and checks for component substituability.

To perform the checks, every component is instrumented with the size it needs for its code and for its data. This should then be summed and combined with the information from the blackboard.

4.3 Possibilities for Formal Verification

We are thinking to lift the constraint that components are completely black-box, and adding and using state charts as a way to describe the behavior of components. When we

Table 1: Structural Checks in the Pecos Component Model

Port	The type of the property can only be one in a fixed set (Float, Tfloat, Tscale, ...);
	The direction should be in, out or inout;
	The location of a port has to be 'static', 'dynamic', or 'nv';
	The minimum in the range is smaller than the maximum.
Component	The State can only be active, passive or 'event';
	All the numbers regarding timing and code sizes should be positive or 0..
Connector	Connectors can only connect out and in; ports;
	The types of the ports should be compatible;
	The ranges of ports should be compatible;

do this, we can think of using synchronous languages such as Esterel [1], Argo/Argonaute [5], Lustre [3], CRP [2] and combined approaches [4].

Especially Esterel seems a natural candidate to use in the context of embedded systems. It is a synchronous and imperative concurrent language dedicated to control-dominated reactive programs which are found in real-time process control, embedded systems, supervision of complex systems, communication protocols and HMI. In Esterel, programs are abstractions that manipulate input signals and generate output signals. Once programs are expressed in Esterel they can be formally proved (i.e., non-reachability of state, timing constraints), compiled to C in a compact form, and simulated. In the context of Pecos, Esterel seems particularly interesting because the size generated is suitable for field devices and, more important, timing issues and memory consumption can be verified:

- it allows the verification that given an input, the output of a program is comprised in a certain amount of cycles of the input. This means that component substitution could be verified.
- it allows different code generation schemas. The first one is boolean generation. By counting the number of instructions the exact size of a component and its exact execution time can be counted. The second is condition-based and can provide maximum execution time for a component.

Another possibility would be to look at the formalism of timed state automata, to take timing information into account.

5. CONCLUSION

In this paper we describe the context of embedded systems, for which we made a component model to specify and check Field Devices (a particular kind of embedded system). Due to the physical constraints imposed on embedded systems, a component model for embedded devices has very specific constraints: no runtime allocation, no locking or synchronization, and a simple scheduler. We describe the Pecos Component Model that we are developing, and that allows to specify and check Field Devices and their components. The most interesting aspect of the model is that we want to check certain non-functional requirements before the software for the field device is deployed in the hardware. This is still under ful development. We showed the current status of the checks, and where we suspect that formal techniques could be welcomed. In the workshop we want to discuss with people from the formal community, using our context as a test case.

6. REFERENCES

[1] G. Berry. *The foundations of Esterel*. MIT Press, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[2] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In ACM, editor, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1993*, pages 85–98. ACM Press, 1993.

[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. In *Proceedings of the IEEE*, September 1991.

[4] F. M. M. Jourdan, F. Lagnier and P. Raymond. A multiparadigm language for reactive systems. In *Proceedings of the IEEE Internal Conference on Computer Languages*, 1994.

[5] F. Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. In *Proceedings of the IEEE Internal Conference on Visual Languages*, 1991.

[6] K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pages 33–45, June 1999.

[7] C. A. Szyperski. *Component Software*. Addison-Wesley, 1998.