Behavioral Substitutability in Component Frameworks: a Formal Approach

Sabine Moisan
INRIA Sophia Antipolis
2004, route des Lucioles
06902 Sophia Antipolis, France
Sabine.Moisan@inria.fr

Annie Ressouche
INRIA Sophia Antipolis
2004, route des Lucioles
06902 Sophia Antipolis, France
Annie.Ressouche@inria.fr

Jean-Paul Rigault
I3S Laboratory
UNSA/CNRS, UMR 6070
06902 Sophia Antipolis, France
jpr@essi.fr

ABSTRACT

When using a component framework, developers need to respect the behavior implemented by the components. Dynamic information such as the description of valid sequences of operations is required. In this paper we propose a mathematical model and a formal language to describe the knowledge about behavior. We rely on a hierarchical model of deterministic finite state-machines. The communication between the machines follows the Synchronous Paradigm. We focus on extension of components, owing to the notion of behavioral substitutability. Our approach relies on compositionality properties to ease automatic verification. From the language and the model, we can draw practical design rules that preserve safety properties. Associated tools may ensure correct and safe reuse of components, as well as automatic simulation and verification, code generation, and run-time checks.

Keywords

framework, components, behavioral substitutability, synchronous reactive systems, model checking

1. INTRODUCTION

A current trend in Software Engineering is to favor reusability of code and also of analysis and design models. This is mandatory to improve product time to market, software quality, maintenance, and to decrease development cost. The notion of frameworks was introduced as a possible answer to these needs. Basically, a framework is a well-defined architecture composed of generic classes and their relationships. As reusable entities, classes rapidly appeared as too fine grained. Hence, the notion of component frameworks emerged. According to Szyperski [21] a component is "a unit of [software] composition with contractually specified interfaces and explicit context dependencies...". In the object-oriented approach a component usually corresponds to a collection of interrelated classes and objects providing a logically consistent set of services.

Using a component framework involves selecting, adapting, and assembling components to build a customized application. Thus *reusing* existing components is a major task. Building on reusability is not straightforward, though; it implies to understand the nature of the contract between the client (i.e., the framework user) and the component. This contract may be the mere specification of a static interface (list of operation signatures), which is clearly not sufficient since it misses the information regarding the component *behavior*. Adding pre- and post-conditions to operations is an interesting improvement. However, contracts express only behavior local to an operation, making it difficult to comprehend the global valid sequences of operations. The description of such a valid sequence is the essential part of what we call the *protocol of use* of the framework. This protocol is often more complex than for using, e.g., a simple library. Hence, it is important to provide models and tools to formalize it, reason about it, and manipulate it.

Our work on formalizing component protocols relies on our experience with a framework for knowledge-based system (KBS) inference engines, named BLOCKS [17]. BLOCKS's objective is to help designers create new engines and reuse or modify existing ones, without extensive code rewriting. It is a set of C++ classes, each one coming with a behavioral description of the valid sequences of operations, in the form of state-transition diagrams. Such a description allowed us to prove invariant properties of the framework, using model-checking techniques. As with other frameworks, the developer adapts BLOCKS classes essentially through subtyping (more exactly, class derivation used as subtyping). The least that can be expected is that the derived classes respect the behavioral protocol that the base classes implement and guarantee. In particular, we want to ensure that an invariant property at the framework base level also holds at the developer's class level. Thus the notion of *behavioral substitutability* is central to such a safe use of the framework. To this end we chose to elaborate a formal model of behavioral substitutability so that we may lay design rules on top of it. In this model safety properties are preserved during subtyping. Our aim is to propose a verification algorithm as well as practical design rules to ensure sound framework adaptation.

The paper is organized as follows. The next section details our notion of components and defines their protocols of use. Section 3 presents the mathematical model and the formal language to describe the behavioral part of the protocol. Section 4 illustrates practical design rules drawn from the model. Section 5 briefly compares our approach with other techniques and methods. The conclusion draws perspectives about the development of supporting tools.

2. TARGET FRAMEWORK CHARACTERISTICS

2.1 Notion of Components

In the object-oriented community a component framework is usually composed of hierarchies of classes that the framework user may compose or extend. The root class of each hierarchy corresponds to an important concept in the target domain. In this context, a component can be viewed as the realization of a sub-tree of the class hierarchy: this complies with one of Szyperski's definitions for components [21].

As a matter of example, let us examine the problem of history management in an object-oriented environment. In our framework (BLOCKS) a *history* is composed of several successive *snapshots*, each one gathering the modifications (or *deltas*) to object attributes that have happened since the previous snapshot (that is during an execution step). It is a rather general view of history management and any framework with a similar purpose is likely to provide classes such as `History`, `Snapshot` and `Delta`, as shown in the UML class diagram of figure 1. Class `Snapshot` memorizes the modifications of objects during an execution step in its attached `Delta` set; it displays several operations: memorize the deltas and other contextual information, add a new delta, and add a child snapshot (i.e., close the current step and start a new one).

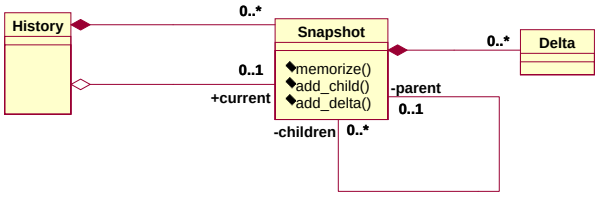


Figure 1: Simplified UML diagram of class `Snapshot`

2.2 Using a Framework

Framework users both adapt the components and write some glue code.They will (non-exclusively) use these components directly, or specialize the classes they contain by inheritance, or compose the classes together, or instantiate new classes from predefined generic[1] ones. Among all these possibilities, class derivation is frequent. It is also the one that may raise the trickiest problems, that is why we shall concentrate on it in the rest of this paper.

When deriving a class the user may either introduce new attributes and/or operations or redefine inherited operations. These specializations should be "semantically acceptable", i.e., they should respect the framework invariants.

In our example, the `Snapshot` class originally does not take into account a possible "backtrack" (the "linear" history of `Snapshot` becomes a "branching" one). This feature is necessary in simulation activities to try different actions or to modify some contextual information and see what happens. To cope with such requirements, the user can derive a `BSnapshot` class from `Snapshot` (figure 2). In this example, the inherited operations need no redefinition[2]. `BSnapshot` defines two new operations: `regenerate` that reestablishes the memorized values and `search` that checks whether a condition was true in a previous state. The regeneration feature implies that deltas have the ability to redo and undo their changes; hence the new class `BDelta` has to be substituted to `Delta`. Relying on static information in the class diagram of `Snapshot` (signatures of operations and associations), the framework user obtains the inheritance graph shown on figure 2.

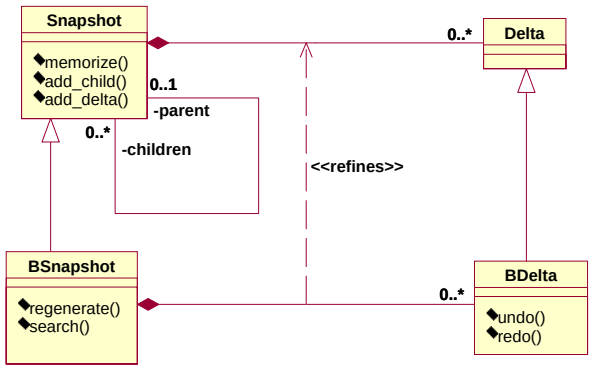


Figure 2: UML class diagram of `BSnapshot`; above, the original classes, below, the derived ones

2.3 Protocol(s) of Use

Static information is not sufficient to ensure a safe and correct use of a framework: specifying a *protocol of use* is required. This protocol is defined by two sets of constraints. First, a *static* set enforces the internal consistency of class structures. UML-like class diagrams provide a part of this information: input interfaces of classes (list of operation signatures), specializations, associations, indication of operation redefinitions, and even constraints on the operations that a component expects from other components (a sort of *required interface*, something that will likely find its way into UML 2.0). We do not focus on this part of the protocol since its static nature makes it easy to generate the necessary information at compile-time. A second set of constraints describes *dynamic* requirements: (1) the legal sequences of operation calls, (2) the specification of internal behavior of operations and of sequences of messages these operations send to other components, and (3) the behavioral specification of valid redefinition of operations in derived classes. These dynamic aspects are more complicated to express than static ones and there is no tool (as natural as compiler-like tools for the static case) to handle and check them. While item (1) and partially item (2) are addressed by classical UML state-transition models, the complete treatment of the last two items is more challenging. We shall propose a solution in section 3.

3. BEHAVIOR DESCRIPTION AND BEHAVIOR REFINEMENT

Our approach is threefold. First, we define a mathematical model providing consistent description of *behavioral entities*. In the model, behavioral entities are whole components, sub-components, single operations, or any assembly of these. Hence, the whole system is a hierarchical composition of communicating behavioral entities. Such a model complements the UML approach and allows to specify the class and operation behavior with respect to class derivation. Second, we propose a *hierarchical* behavioral specification *language* to describe the dynamic aspect of components. In the third place, we define a *semantic* mapping to bridge the gap

[1] class templates in C++

[2] In the general case, there would be new operations as well as redefined operations. Our approach is able to cope with both cases.

between the specification language and its meaning in the mathematical model.

As already mentioned, our primary intent is to formalize the behavior side of class derivation, in the sense of *subtyping*[3]. In the object-oriented approach, subtyping usually obeys the classical Substitutability Principle [13]. This principle has a static interpretation which leads to, for instance, the well-known covariant and contravariant issues for parameters and return types. But it may also be given a dynamic interpretation, leading to behavioral subtyping, or *behavioral substitutability* [10]. This is the kind of interpretation we need to enforce the dynamic aspect of framework protocols, since it provides a notion of behaviorwise safe derivation.

To deal with behavioral substitutability, we need behavior representation formalisms: we propose to rely on the family of *synchronous* models [9]. These models are dedicated to specify event-driven and discrete time systems. Such systems interact with their environment, reacting to input events by sending output events. Furthermore, they obey the *synchrony hypothesis*: the corresponding reaction is *atomic*; during a reaction, the input events are frozen, all events are considered as *simultaneous*, events are broadcast and available to any part of the system that listens to them. A reaction is also called an *instant*. The succession of instants defines a logical time. The major interest of synchronous models is that their verification exhibits a lower computational complexity than asynchronous ones, which is the main reason for our choice.

3.1 Mathematical Model of Behavior

Labeled transition systems are usual mathematical models for synchronous languages. These systems are a special kind of finite deterministic state machines (automata) and we shall denote them LFSM for short. In our model, we use LFSMs to represent the state behavior of *behavioral entities* (classes as well as their operations). Each transition has a *label* representing an elementary execution step of the entity, consisting of a *trigger* (input condition) and an *action* to be executed when the transition is fired. In our case an action corresponds to emitting events, such as calling an operation of some component whereas a trigger corresponds to receiving events such as calling an operation.

A LFSM is a tuple $M = (S, s_0, T, A)$ where S is a finite set of states, $s_0 \in S$ is the initial state, A is the *alphabet* of events from which the set of labels L is built, and T is the transition relation $T \subseteq S \times L \times S$. We introduce the set I of input events $I \subseteq A$ and the set $O \subseteq A$ of output events (or actions).

Labels. L, the set of *labels*, has elements of the form i/o, where i is the trigger set and $o \subseteq O$ the action or output events set; i has the form (i^+, i^-) where i^+, the positive (input event) set of a label (resp. i^-, the negative (input event) set), consists of the events tested for their presence (resp. for their absence) in the trigger at a given instant.

A trigger contains the information about all the input events, be they present or absent at a given instant. Obviously, an event cannot be tested for both absence and presence at the same instant. Thus (i^+, i^-) constitutes a partition of I. Moreover, as a consequence of the previous definition of an *instant* in the synchronous model, an event cannot be tested for absence while being emitted in the same instant. Hence, the following well-formedness conditions on labels:

$$\begin{cases} i^+ \cap i^- = \emptyset & \text{(trigger consistency)} \\ i^+ \cup i^- = I & \text{(trigger completeness)} \\ i^- \cap o = \emptyset & \text{(synchrony hypothesis)} \end{cases}$$

Transitions. Each transition $s \xrightarrow{l} s'$ has three parts: a source state s, a label l, and a target state s'. There cannot be two transitions leaving the same state and bearing the same trigger. Formally, if there are two transitions from the same state s such that $s \xrightarrow{i_1/o_1} s_1$ and $s \xrightarrow{i_2/o_2} s_2$, with $s_1 \neq s_2$, then $i_1 \neq i_2$. This rule, together with the label well-formedness conditions, ensure that LFSMs are *deterministic*. This requirement for determinism constitutes one of the foundations of the synchronous approach and is mandatory for all models and proofs that follow.

Behavioral Substitutability. The substitutability principle should apply to the dynamic semantics of a behavioral entity–such as either a whole class or one of its (redefined) operations [10, 18]. If M and M' are LFSMs denoting respectively some behavior in a base class and its redefinition in a derivative, we seek for a relation $M' \preceq M$ stating that "M' safely extends M". To comply with inheritance, this relation must be a preorder.

Following the substitutability principle, we say that M' is a correct extension of M, iff the alphabet of M' (A'_M) is a superset of the alphabet of M (A_M) and every sequence of inputs that is valid [4] for M is also valid for M' and produces the same outputs (once restricted to the alphabet of M). Thus, the behavior of M' restricted to the alphabet of M is identical to the one of M. Formally,

$$M' \preceq M \Leftrightarrow A_M \subseteq A_{M'} \wedge M \, \mathcal{R}_{sim} \, (M' \backslash A_M)$$

where $M' \backslash A_M$ is the *restriction* of M' to the alphabet of M and $\mathcal{R}_{sim}$ the behavioral simulation relation. Both are defined below.

First, we define the restriction ($l \backslash A$) of a label (l) over an alphabet (A) as follows: let $l = i/o$,

$$l \backslash A = \begin{cases} (i \cap A)/(o \cap A) & \text{if } i^+ \subseteq A \\ undef & \text{otherwise} \end{cases}$$

Intuitively, this corresponds to consider as undefined all the transitions bearing a positive trigger not in A, and to strip the events not in A from the outputs.

The restriction of M to the alphabet A (generally with $A \subseteq A_M$) is obtained by restricting all the labels of M to A, then discarding the resulting undefined transitions. Formally, let $M = (S, s_0, T, A_M)$ be a LFSM, $M \backslash A = (S, s_0, T \backslash A, A_M \cap A)$ where $T \backslash A$ is defined as follows:

$$s \xrightarrow{l'} s' \in T \backslash A \Leftrightarrow \exists \, s \xrightarrow{l} s' \in T \wedge l' = l \backslash A \neq undef$$

Second, we adopt a behavioral simulation relation similar to Milner's classical simulation [16]. Let M_1 and M_2 be two LFSMs with the same alphabet: $M_1 = (S_{M_1}, s_0^{M_1}, T_{M_1}, A)$ and $M_2 =$

[3]Note that, in this paper, derivation, inheritance, specialization all refer to the *subtyping* interpretation. In particular, we do not consider the other uses or interpretations of inheritance that some programming languages may offer.

[4]A *path* in a LFSM M is a (possibly infinite) sequence of transitions $\pi = s_0 \xrightarrow{i_0/o_0} s_1 \xrightarrow{i_1/o_1} s_2 ...$ such that $\forall i (s_i, i_i/o_i, s_{i+1}) \in T$. The sequence $i_0/o_0, i_1/o_1 ...$ is called the *trace* associated with the path. When such a path exists, the corresponding trigger sequence $i_0, i_1, ...$ is said to be a *valid* sequence of M.

$(S_{M_2}, s_0^{M_2}, T_{M_2}, A)$. A relation $\mathcal{R}_{sim} \subseteq S_{M_1} \times S_{M_2}$ is called a *simulation* iff $(s_0^{M_1}, s_0^{M_2}) \in \mathcal{R}_{sim}$ and

$$\forall (s_1, s_2) \in \mathcal{R}_{sim} :$$
$$s_1 \xrightarrow{l} s_1' \in T_{M_1} \Rightarrow \exists s_2 \xrightarrow{l} s_2' \in T_{M_2} \wedge (s_1', s_2') \in \mathcal{R}_{sim}$$

Simulation is local, since the relation between two states is based only on their successors. As a result, it can be checked in polynomial time and it is widely used as an efficient computable condition for trace-containment. Moreover, the simulation relation can be computed using a symbolic fixed point procedure [11], allowing to tackle large-sized state spaces.

We say that M' *simulates* M iff $M' \preceq M$. Thus, M' simulates M iff there exists a relation binding each state of M to a state of the restriction of M' to the alphabet of M. Any valid sequence of M is also a valid sequence of M' and the output traces are identical, once restricted to A_M. As a consequence, if M' simulates M, M' can be substituted for M, for all purposes of M.

Milner's simulation relation ($\mathcal{R}_{sim}$) is a preorder and preserves satisfaction of the formulae of a subset of temporal logic, expressive enough for most verification tasks (namely $\forall CTL^*$ [12]). Moreover, this subset has efficient model checking algorithms. Obviously, relation $\preceq$ is also a preorder over LFSMs and any formula that holds for M holds also for M'.

The notion of correct extension can be extended to components. We can represent the *protocol of use* of a class C (see section 2.3) by a LFSM $\mathcal{P}(C)$. If C and C' are two classes, $C' \preceq C$ iff (1) C' derives from C (according to footnote [3], this means "is a subtype of"), (2) the protocol of use of C' simulates the one of C, that is $\mathcal{P}(C') \preceq \mathcal{P}(C)$. As indicated in 2.3, we assume that the protocol of use of a class describes not only the way the other objects may call the class operations, but also the way the operations of the class invoke operations on (other) objects.

With such a model, the description of behavior matches the class hierarchy. Hence, class and operation refinements are compatible and consistent with the static description: checking dynamic behavior may benefit from the static hierarchical organization.

3.2 Behavior Description Language

We need a language that makes it possible to describe complex behavioral entities in a structured way, particularly by means of scoping and composition. Our language is very similar to *Argos* [15]. It offers a graphical notation close to UML StateCharts with some restrictions, but with a different semantics based on the Synchronous Paradigm [9]. The language is easily compiled into LFSMs. Programs written in this language operationally describe behavioral entities; we call them *behavioral programs*. The semantics of this language should be expressible in terms of the mathematical model, permitting an easy translation into LFSMs.

The primitive elements from which programs are constructed are called *flat automata*, since they cannot be decomposed (they contain no application of any operators). They are the direct representation of LFSMs, with the following simplified notation: only positive (i.e., present) events appear in triggers, all other events are considered as absent.

The language is generated by the following grammar (where A is a flat automaton, s a state name and Y a set of events):

$$P ::= A \mid A[P/s] \mid P \| P \mid P < Y >$$

Parallel composition ($P \| Q$) is a symmetric operator which behaves as the synchronous product of its operands where labels are unioned. *Hierarchical composition* ($A[P/s]$) corresponds to the possibility for a state in an automaton to be refined by a behavioral (sub) program. This operation is able to express preemption, exceptions, and normal termination of sub-programs. *Scoping* ($P < Y >$) where P is a program and Y a set of *local* events, makes it possible to restrict the scope of some events. Indeed, when refining a state by combining hierarchical and parallel composition, it may be useful to send events from one branch of the parallel composition to the other(s) without these events being globally visible. This operation can be seen as encapsulation: local events that fired a transition must be emitted in their scope; they cannot come from the surrounding environment.

The language offers syntactic means to build programs that reflect the behavior of components. Nevertheless, the soundness of the approach requires a clear definition of the relationship between behavioral programs and their mathematical representation as LFSMs (section 3.1). Let $\mathcal{B}$ denote the set of behavioral programs and $\mathcal{M}$ the set of LFSMs. We define a *semantic* function $\mathcal{S} : \mathcal{B} \longrightarrow \mathcal{M}$ that is stable with respect to the previously defined operators (parallel composition, hierarchical composition, and scoping).

$\mathcal{S}$ is *structurally* defined over the syntax of the language. Because of lack of space, we just give here the flavor of the definition of this semantics. For a more complete description, see [20]. A flat automaton constitutes its own semantics. The semantics of parallel composition $P \| Q$ is the *synchronous product* [9] of the semantics of P and Q: each reaction (instant) is considered atomic; within an instant, input and output events are matched by name, providing instantaneous communication. The semantics of hierarchical composition $P[Q/s]$ is basically the one of P where state s has been replaced by the semantics of Q whose transitions have been modified to respect the outgoing transitions (preemptions) of s. More specifically, in the absence of preemption, the semantics of Q remains the same; otherwise, the preemptions of s have priority, which may lead to unioning internal and preemption actions. For scoping, the semantics of $P < Y >$ is basically the one of P where transitions triggered by local events that are not emitted are discarded and where the occurrences of local events are removed from the labels of the remaining transitions. Thus all events in Y (be they triggers or actions) are encapsulated within $P < Y >$ and invisible from the outside, as is invisible the internal communication they support.

The following theorem expresses that relation $\preceq$ is a congruence with respect to the language operators. The proof [20] is out of the scope of the paper, and is obtained by explicit construction of the preorder relation.

THEOREM 1. *Let P, Q_1 and Q_2 be behavioral programs such that $\mathcal{S}(Q_1) \preceq \mathcal{S}(Q_2)$ and both P, Q_1 and P, Q_2 are outputs disjoint; the following holds:*
$\mathcal{S}(P[Q_1/s]) \preceq \mathcal{S}(P[Q_2/s])$
$\mathcal{S}(P \| Q_1) \preceq \mathcal{S}(P \| Q_2)$
$\mathcal{S}(Q_1 < Y >) \preceq \mathcal{S}(Q_2 < Y >)$

This *compositionality property* is fundamental to our approach. It

gives a modular and incremental way to verify behavioral programs using their natural structure: properties of a whole program can be deduced from properties of its sub-programs. This helps to push back the bounds of state explosion, the major drawback of model checking.

3.3 Modular Verification

The compositionality property is very useful, since one can deal with highly complex global behaviors provided that they result from composing elementary behaviors that can be verified, modified, and understood incrementally. In particular it makes it possible to perform modular verification using some temporal logic.

Temporal logics are formalisms for describing sequences of transitions between states in a finite state machine model. They are formal languages where assertions related to behavior are easily expressed. The logic we consider ($\forall CTL^*$) [12] is based on first-order logic. This logic, to be efficient when deciding whether a formula is true, does not introduce the existential path quantifier. It offers *temporal operators* that make it possible to express properties holding for a given state, for the next state (operator **X**), eventually for a future state (**F**), for all future states (**G**), or that a property remains true until some condition becomes true (**U**). One can also express that a property holds for all the paths starting in a given state ($\forall$). These operators can be combined with boolean connectors and nested.

The logic may be interpreted over LFSMs. One can express model-checking algorithms and satisfaction of a formula is defined in a natural inductive way. We say that a LFSM M satisfies a state formula ψ ($M \models \psi$) if property ψ is true for the initial state of M.

In the same line as Clarke et al. [12], the main result of our approach is the following theorem (the proof [20] is by structural induction on formulae and from the translation of LFSMs into Kripke structures).

THEOREM 2. *Let P and Q two behavioral programs with disjoint output sets and ψ a $\forall CTL^*$ formula:*
if $\mathcal{S}(P) \models \psi$ then $\mathcal{S}(P[Q/s]) \models \psi$.
if $\mathcal{S}(P) \models \psi$ or $\mathcal{S}(Q) \models \psi$ then $\mathcal{S}(P \| Q) \models \psi$.

This result complements theorem 1; it expresses the compositional stability of proofs with respect to the composition operators. This property provides a hierarchical and incremental means to verify properties and is the key to simplify model checking.

The properties that are preserved by our operators include substitutability. This is an immediate consequence of theorems 1 and 2. For instance, if we have proved that $P_1 \preceq P_2$, then we can infer that $P_1 \| Q \preceq P_2 \| Q$, for any possible Q, provided that it is output disjoint with P_1 and P_2.

4. PRACTICAL ISSUES

4.1 Design Rules

To guarantee a safe use of the components, we state some practical design rules that we can draw from our model and that can be applied at the behavioral language level. When a behavioral program P (called the base program) is extended by another behavioral program P', respecting these rules ensures that we obtain a new deterministic automaton for which behavioral substitutability holds ($P' \preceq P$). These rules correspond to *sufficient* conditions that save us the trouble of a formal proof for each derived program.

At this time we have identified eight such practical rules. A formal description of these rules can be found in [20]. We briefly list them here: (1) no modification of the base program structure (no deletion nor modification of transitions or states); (2) possibility of adding trigger-disjoint transitions for a given state; possibility of parallel composition with a program with (3) disjoint actions or (4) different initial trigger or (5) with a substitutable program; possibility of hierarchical composition with a program (6) without auto-preemption or (7) with disjoint triggers and actions; (8) no localization of global events.

4.2 Application to Components

To illustrate our purpose, let us consider the previously mentioned history mechanism (section 2.1). We present on figure 3(a) the behavioral program for the whole `Snapshot` class. This program specifies the valid sequences of operations that can be applied to `Snapshot` instances. Two states correspond to execution of operations (`memorize` and `add_child`); they are to be refined by behavioral programs describing these operations. Figure 3(b) presents the expected behavioral program for `BSnapshot` which derives from class `Snapshot`. In particular, `BSnapshot` necessitates a new operation, `regenerate`, called when backtracking the history (i.e., when `search` returns `success`). It is clear that the new class sports a behavior significantly different from its base class: it has the extra possibilities to search inside a sleeping snapshot and to call `regenerate` when `success` occurs.

The behavioral program of `BSnapshot` has been obtained from the one of `Snapshot` after applying a combination of our design rules. Obviously no state nor transition have been deleted from `Snapshot` (rule 1). The new transition from `inactive` to `regeneration` bears a completely new trigger (rule 2). The program that refines state `inactive` has no trigger belonging to the preemption trigger set of this state (rule 7). Finally, the local event `success` was not part of the `Snapshot` program (rule 8). Thus, by construction, `BSnapshot` is substitutable for class `Snapshot`; no other verification is necessary to assert that `BSnapshot` $\preceq$ `Snapshot`.

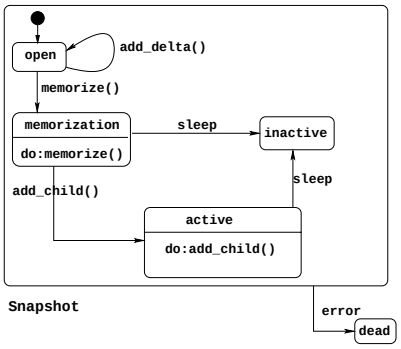
Therefore, even though `BSnapshot` extends `Snapshot` behavior, the extension has no influence when a `BSnapshot` is used as a `Snapshot`. As a result, every trace of `Snapshot` is also a trace of `BSnapshot`.

4.3 Stability of Properties

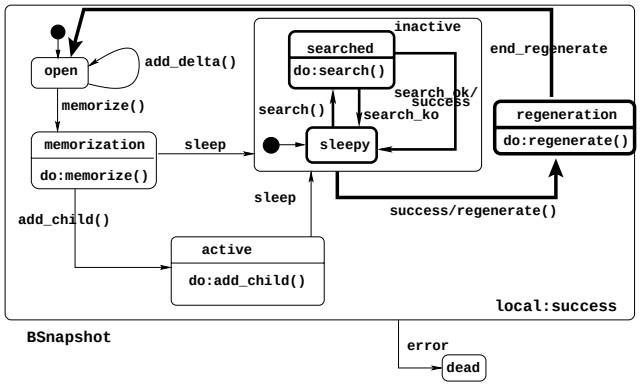
Continuing with the previous example, `BSnapshot` $\preceq$ `Snapshot` implies that every temporal property in $\forall CTL^*$ true for `Snapshot` is also true for its extension `BSnapshot`. For instance, suppose we wish to prove the following property: "It is possible to add a child to a snapshot (i.e., to call the `add_child()` operation) only after memorization has been properly done". Looking at the behavioral program (figure 3(a)), we can decompose P_{child} into two specifications:

$\forall \mathbf{G}(\texttt{add\_child}() \& \forall \mathbf{G}(\neg \texttt{error})) \Rightarrow \forall \mathbf{F} \texttt{state} = \texttt{inactive}$
$\forall \mathbf{G}(\texttt{error} \Rightarrow \forall \mathbf{G}(\neg \texttt{state} = \texttt{inactive}))$

Intuitively, the first formula corresponds to memorization success: if `add_child()` is received and if no `error` occurs, then state `inactive` is reached. The second formula corresponds to memorization failure: `error` occurred, and state `inactive` will never be reached. A model-checker can verify automatically that these



(a) Behavioral program of class `Snapshot`.



(b) Behavioral program of class `BSnapshot`. It is similar to `Snapshot` with a refined `inactive` state, a local event `success`, and the possibility of launching `regenerate` from the `inactive` state. Restriction `BSnapshot` $\setminus A_{\texttt{Snapshot}}$ is obtained by removing states and transitions displayed with thick lines.

Figure 3: Behavioral programs of classes `Snapshot` and `BSnapshot`.

two formulae are true. Conversely, if a formula is false, the model checker usually gives a counter-example.

5. RELATED WORK AND DISCUSSION

Modeling component behavior and protocols and ensuring correct use of component frameworks through a proof system is a recent research line. Most approaches concentrate on the composition problem [14, 1, 8] whereas we are focusing this paper on the substitutability issue.

Most works in the field of Software Architecture for modeling behavior [3] address component compatibility and adaptation in a distributed environment and are often based on process calculi [18, 22, 19]. Some authors put a specific emphasis on the substitutability problem [13]. For instance [4] proposes static subtype checking relying on Nierstrasz's notion of regular types [18]. As another example, in [5], the authors focus on inheritance and extension of behavior, using the π-calculus as their formal model. These works also consider a distributed environment. The problems of compatibility and substitutability are also significant in fields other than Software Engineering, such as hardware modeling and design. As a matter of example, [6] proposes a "game view" of (hardware) components, relying on deterministic automata.

As far as the objectives (well-formedness, verification, compatibility, and refinement) and models (deterministic automata, non-distributed environment) are concerned, our work is close to the one in [6], although our target applications are similar to the Software Architecture community ones. Another approach introduces behavioral compatibility relying on type-theory [2]. It is more general than ours in its objectives, although quite similar as far as behavioral description is concerned; it is also more general theoretically speaking, while we focus on providing operational tools. In contrast with these works, we restrict to the problem of substitutability in a non-distributed world. Indeed this is what we needed for BLOCKS. Again, this restriction allows us to adopt models more familiar to software developers (UML StateCharts-like), easier to handle (deterministic systems), efficient for formal analysis (model-checking and simulation), and for which there exist effective algorithms and tools. The Synchronous Paradigm [9] offers good properties and tools in such a context. This is why we could use it as the foundation of our model.

As already mentioned our notion of substitutability guarantees the stability of interesting (safety) properties during the derivation process. Hence, at the user level as well as at the framework one, it may be necessary to automatically verify these properties. To this end, we have chosen model checking techniques. Indeed, model checkers rely on verification algorithms based on the exploration of a state space and they can be made automatic since tools are available. They are robust and can be made transparent to framework users. The problem with model checkers is the possible explosion of the state space. Fortunately, this problem has become less limiting over the last decade owing to symbolic algorithms. Furthermore, taking advantage of the structural decomposition of the system allows modular proofs on smaller (sub-)systems. This requires a formal model that exhibits the *compositionality property*, which is the case for our model (theorems 1 and 2).

6. CONCLUSION AND PERSPECTIVES

The work described in this paper is derived from our experience and aims at simplifying the correct use of a framework. We have adapted framework technology to the design of knowledge-based system engines and observed a significant gain in development time. For instance, once the analysis completed, the design of a new planning engine based the BLOCKS framework took only two months (instead of about two years for a similar former project started from scratch) and more than 90 % of the code reused existing components [17]. While performing these extensions, we realized the need to formalize and verify component protocols, especially when dealing with subtyping. The corresponding formalism, the topic of this paper, has been developed in parallel with the KBS engines. As a consequence of this initial work, developing formal description of BLOCKS components led us to a better organization of the framework, with an architecture that not only satisfies our design rules but also makes the job easier for the framework user to commit to these rules.

Our behavioral formalism relies on a mathematical model, a specification language, and a semantic mapping from the language to the model. The model supports multiple levels of abstraction, from highly symbolic (just labels) to merely operational (pieces of code).

Moreover, this model is original in the sense that it can cover both static and dynamic behavioral properties of components. To use our formalism, the framework user has only to describe behavioral programs, by drawing simple StateCharts-like graphs with a provided graphic interface. The user may be to a large extend oblivious of the theoretical foundations of the underlying models and their complexity. The model has also a pragmatic outcome: it allows simulation of resulting applications and generation of code, of run-time traces, and of run-time assertions.

Our aim is to accompany frameworks with several kinds of dedicated tools. We are working on tools for manipulating behavioral programs. Currently, we provide a graphic interface to display existing descriptions and modify them. In the future, the interface will watch the user activity and warn about possible violation of the design rules. Since these rules are just sufficient, it is possible for the user not to apply them or to apply them in such a way that they cannot be clearly identified. To cope with this situation, we shall also provide a static substitutability analyzer, based on our model (section 3.1) and a usual partitioning simulation algorithm.

At the present time we have designed a complete interface with *NuSMV* [7], in both directions. First, our description language can be translated into *NuSMV* specifications, and our tool provides also a user friendly way to express the properties the users may want to prove. Second, *NuSMV* diagnosis and return messages are displayed in a readable form: users can browse the hierarchies of behavioral derivations and follow the steps of the proofs. It took us a few weeks to connect our behavioral description language to the *NuSMV* model-checker. The next step is to implement the substitutability analysis tool.

Another interesting feature would be to provide an automatic code generation facility as well as run-time checks. Indeed the behavioral description is rather abstract and may be interpreted in a variety of ways. In particular, automata and associated labels can be given a code interpretation. The generated code would provide skeletal implementations of operations. This code will be correct, by construction–at least with respect to those properties which have been previously checked. Furthermore, the generated code can also be instrumented to provide run-time traces and assertions built in the components.

Developing such tools is a heavy task. Yet, as frameworks are becoming more popular but also more complex, one cannot hope using them without some kind of active assistance, based on formal modeling of component features and automated support.

7. REFERENCES

[1] F. Achermann and O. Nierstrasz. Applications = Components + Scripts - A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[2] S. Alagic and S. Kouznetsova. Behavioral Compatibility of Self-Typed Theory. In B. Magnusson, editor, *ECOOP 2002*, number 2374 in LNCS, pages 585–608, Malaga, Spain, 2002. Springer.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans, on Software Engineering and Methodology*, 6(3):213–249, 1997.

[4] S. Butkevich, M. Renedo, G. Baumgartner, and M. Young. Compiler and Tool Support for Debugging Object Protocols. In *Proc. 8th ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, pages 50–59, San Diego, CA, USA, 2000. ACM Press.

[5] C. Canal, E. Pimentel, and J. M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, (41):105–138, 2001.

[6] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and Freddy Y. C. Mang. Synchronous and Bidirectional Component Interfaces. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proc. CAV*, number 2404 in LNCS, pages 214–227. Springer, 2002.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: an OpenSource Tool for Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proc. CAV*, number 2404 in LNCS, pages 359–364. Springer, 2002.

[8] J. Costa Seco and L. Caires. A Basic Model of Typed Components. In Elisa Bertino, editor, *ECOOP 2000*, number 1850 in LNCS, pages 108–128. Springer, 2000.

[9] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic, 1993.

[10] D. Harel and O. Kupferman. On object systems and behavioral inheritance. *IEEE Trans. Software Engineering*, 28(9):889–903, 2002.

[11] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulation on finite and infinite graphs. *Proc. IEEE Symp. Foundations of Computer Science*, pages 453–462, 1995.

[12] E. M. Clarke Jr., O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[13] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[14] K. Mani Chandy and M. Charpentier. An experiment in program composition and proof. *Formal Methods in System Design*, 20(1):7–21, January 2002.

[15] F. Maraninchi. Operational and Compositional Semantics of Synchronous Automaton Composition. In *Proc. Concur. 1992*, number 630 in LNCS. Springer, 1992.

[16] R. Milner. An algebraic definition of simulation between programs. *Proc. IJCAI*, pages 481–489, 1971.

[17] S. Moisan, A. Ressouche, and J-P. Rigault. BLOCKS, a Component Framework with Checking Facilities for Knowledge-Based Systems. *Informatica*, 25:501–507, 2001.

[18] Nierstrasz O. *Object-Oriented Software Composition*, chapter Regular Types for Active Objects, pages 99–121. Prentice-Hall, 1995.

[19] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. on Software Engineering*, 28(11), Nov 2002.

[20] A. Ressouche and S. Moisan. A Behavior Model of Component Frameworks. Technical report, INRIA, August 2003. to appear.

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

[22] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, March 1997.