FOAL '09

Proceedings of the Eighth Workshop on Foundations of Aspect-Oriented Languages

held at the
Eighth International Conference on
Aspect-Oriented Software Development

March 2, Charlottesville, Virginia

Workshop Organizers: Curtis Clifton, Shmuel Katz, Gary T. Leavens, and Mira Mezini

Contents

Preface

Aspect-oriented programming is a paradigm in software engineering and programming languages that promises better support for separation of concerns. The seventh Foundations of Aspect-Oriented Languages (FOAL) workshop was held at the Seventh International Conference on Aspect-Oriented Software Development in Charlottesville, Virginia, on March 2, 2009. This workshop was designed to be a forum for research in formal foundations of aspect-oriented programming languages. The call for papers announced the areas of interest for FOAL as including: semantics of aspect-oriented languages, specification and verification for such languages, type systems, static analysis, theory of testing, theory of aspect composition, and theory of aspect translation (compilation) and rewriting. The call for papers welcomed all theoretical and foundational studies of foundations of aspect-oriented languages.

The goals of this FOAL workshop were to:

- Make progress on the foundations of aspect-oriented programming languages.
- Exchange ideas about semantics and formal methods for aspect-oriented programming languages.
- Foster interest within the programming language theory and types communities in aspect-oriented programming languages.
- Foster interest within the formal methods community in aspect-oriented programming and the problems of reasoning about aspect-oriented programs.

FOAL logos courtesy of Luca Cardelli

The workshop was organized by Curtis Clifton (Rose-Hulman Institute of Technology, USA), Shmuel Katz (Technion–Israel Institute of Technology, Israel), Gary T. Leavens (University of Central Florida, USA), and Mira Mezini (Darmstadt University of Technology, Germany). The program committee was chaired by Mario Südholt.

We thank the organizers of AOSD 2009 for hosting the workshop, and in particular workshops chair Doug Schmidt, the AOSD general chair Kevin Sullivan, and the organizing chair Jeff Gray for their help.

Message from the Program Committee Chair

The eighth FOAL workshop maintains the high bar for quality set by previous instances. FOAL is one of the primary forums for foundational work on aspect-oriented software development. As in the past, each paper was subjected to full review by at least three reviewers. Papers co-authored by program committee members or organizers were reviewed by four or five reviewers and were held to a higher standard. I am grateful to the program committee members for their dedication, insightful comments, attention to detail, and the service they provided to the community and the individual authors.

The members of the program committee were: Curtis Clifton (Rose-Hulman Institute of Technology), Erik Ernst (University of Aarhus), Pascal Fradet (INRIA), Shmuel Katz (Technion-Israel Institute of Technology), Karl Lieberherr (Northeastern University), David Lorenz (The Open University of Israel), Hidehiko Masuhara (University of Tokyo), Mira Mezini (Darmstadt University of Technology) Klaus Ostermann (Darmstadt University of Technology), James Riely (DePaul University), and Damien Sereni (Oxford)

I am also grateful to the authors of submitted works. Twelve papers were submitted for review this year. Of these, the program committee selected six for presentation at the workshop and publication in the proceedings. One of these has been included specifically for a demo-based presentation because of corresponding tool support to animate and illustrate semantics of aspect based languages

The program was rounded out with an invited talk by Mehmet Ak sit of the University of Twente in the Netherlands, and an open session on "New Ideas, Open Questions, and Work in Progress," organized by Shmuel Katz.

Finally, I would like to thank the other members of the organizing committee of FOAL—Shmuel Katz, Gary Leavens, and Mira Mezini—for their work in guiding us toward another inspiring workshop.

Mario Südholt
FOAL '09 Program Chair
École des Mines de Nantes, France

A Type System for Functional Traversal-Based Aspects

Bryan Chadwick
College of Computer & Information Science
Northeastern University, 360 Huntington Avenue
Boston, Massachusetts 02115 USA
chadwick@ccs.neu.edu

Karl Lieberherr
College of Computer & Information Science
Northeastern University, 360 Huntington Avenue
Boston, Massachusetts 02115 USA
lieber@ccs.neu.edu

ABSTRACT

We present a programming language model of the ideas behind Functional Adaptive Programming (AP-F) and our Java implementation, DemeterF. Computation in AP-F is encapsulated in sets of functions that implement a fold over a data structure with the help of a generic traversal. In this paper we define the syntax, semantics, and typing rules of a simple AP-F model, together with a proof of soundness that guarantees that traversal expressions result in a value of the expected type. Applying a function set to a different structure can then be statically checked to eliminate some runtime tests and sources of program errors.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms

Languages,Theory

Keywords

Adaptive Programming, Functional Aspects, Traversals, Type Soundness

1. INTRODUCTION

Aspect Oriented Languages provide an enormous amount of flexibility to programmers, which comes from the specification of aspects over a join point model using pointcuts and advice. In [9] the authors discuss different models that fall under this view, one of which is data structure traversal specifications in DemeterJ [12], called Adaptive Programming (AP) [8]. In AP, join points (traversal *entry* and *exit* points) are selected using a *strategy*, which directs traversal, while advice is encapsulated in *visitors* with `before` and `after` methods. Computation remains adaptable to data structure changes, but computing via side effects (`void` methods) means that adaptability goes unchecked, since structural assumptions are implicit in order dependencies between advice.

We have recently developed a functional formulation of AP that maintains the separate traversal, control (strategies), and computation of DemeterJ, but with traversals that return values. Computation in Functional Adaptive Programming [3] (AP-F) and our implementation, DemeterF [2], is encapsulated in sets of functions (or *function objects*) that, together with a generic traversal function, implement a fold over a data structure. This limits a program's adaptability, but provides more structural information about the flow of data, through argument and return types. Function objects can then be checked statically to ensure safety. In this paper we present a limited model of AP-F, describing its syntax, semantics, and typing rules. We prove the type system sound, meaning that function objects always contain applicable functions (advice is *complete*) and that the resulting program returns a value of the expected type. Applying a function object to a different structure can then be statically checked to eliminate some runtime tests and sources of program errors.

FOAL'09, March 2, 2009, Charlottesville, Virginia, USA.

2. SYNTAX

Figure 1 shows our model syntax including definitions, functions, and simple expressions: variable references, instance construction, and traversals. A program begins with a number of definitions. The types in concrete definitions represent constructor arguments, while each T_i in the definition of an abstract type A becomes a subtype of A.

$$
\begin{array}{rcl}
x & ::= & \text{variable names} \\
C & ::= & \text{concrete type names} \\
A & ::= & \text{abstract type names} \\
\\
P & ::= & D_1 \ldots D_n \;\; e \\
D & ::= & \texttt{concrete}\ C\ (T_1, \ldots, T_n) \\
 & \mid & \texttt{abstract}\ A\ (T_1, \ldots, T_n) \\
T & ::= & C \mid A \\
\\
e & ::= & x \mid \texttt{new}\ C\ (e_1, \ldots, e_n) \mid \texttt{traverse}(e_0,\ F) \\
F & ::= & \texttt{funcset}(f_1 \ldots f_n) \\
f & ::= & (T_0\ x_0, \ldots, T_n\ x_n)\{\ \texttt{return}\ e;\ \}
\end{array}
$$

Figure 1: AP-F Model Language Syntax

For simplicity there are no local variables, fields in ab-

```
// Double a given number representation
abstract  Int (Succ, Zero)
concrete Succ (Int)
concrete Zero ()

traverse(new Succ(new Succ(new Succ(new Zero()))),
         funcset((Succ s, Int i)
                   { return new Succ(new Succ(i)); }
                 (Zero z)
                   { return z; }))
```

Figure 2: Example Program : `Double`

stract types, non-traversal functions, or traversal control, since these do not add anything new to the type system or proofs. Function sets correspond to DemeterF function objects (sets of methods) and are used to compute over traversals of constructed values. Each function provides its arguments with their types and a single body expression, which becomes the function's result. A function set is similar to a list of `lambda` expressions (anonymous functions), though we require that all functions have no free variables.

A simple example program is given in Figure 2 with a traversal that doubles a given number representation; in this case calculating $3 * 2$. For each successor object that is traversed we return a nested double successor with the same inner integer, bottom up. The function for `Zero` is applied first, and the result is subsequently passed to the `Succ` function three times, along with each original nested `Succ` instance. This looks very similar to `fold` in functional languages, but the `traversal` function expression implicitly adapts to different data structure shapes.

2.1 Well Formed Rules

We introduce rules similar to [5] and [4] to ensure that a given program is well formed, before type checking and/or evaluation. The rules are shown below with informal descriptions; the formal definitions are elided for space reasons. With a well formed program, we can now define evaluation of expressions (the program body) in the context of a program's definitions.

TypesOnce(P): Each type is only defined once

SingleSuper(P): Each type is used in at most one abstract definition

InductiveTypes(P): Objects of concrete types are *constructible* without mutation, *i.e.*, abstract types include at least one non-recursive subtype, and data structure cycles contain at least one abstract type.

CompleteTypes(P): Each type in an abstract definition is itself defined

ClosedFunctions(P): All functions in P contain no free variables

3. SEMANTICS

Our semantics describes object creation and a depth-first (bottom up) traversal scheme that applies a function from the given set when applicable. Figure 3 shows the syntax of runtime expressions, values, and evaluation contexts. For recursive traversals and function dispatch we add two expression forms not in the surface syntax. The first gives a

$$
\begin{array}{rcl}
e & ::= & \cdots \\
 & | & \texttt{recur(}\, F,\, v_0,\, e_1, \ldots, e_n \,\texttt{)} \\
 & | & \texttt{apply(}\, f,\, v_0,\, e_1, \ldots, e_n \,\texttt{)} \\
\\
v & ::= & \texttt{new}\, C\,\texttt{(}\, v_1, \ldots,\, v_n \,\texttt{)} \\
\\
E & ::= & [\,] \\
 & | & \texttt{new}\, C\,\texttt{(}\, v \ldots, E, e \ldots \,\texttt{)} \\
 & | & \texttt{traverse(}\, E,\, F \,\texttt{)} \\
 & | & \texttt{recur(}\, F,\, v_0, v \ldots, E, e \ldots \,\texttt{)}
\end{array}
$$

Figure 3: Runtime Expressions, Values, and Evaluation Contexts

reduction context for the recursive step when traversing a value and the second separates the recursive traversal from function application. Values, v, are defined as a subset of the expression forms, including only constructed objects. Rather than congruence rules, we present *evaluation contexts*, E, under which reduction is permitted. A context is not used for `apply(...)` since reduction proceeds directly from `apply` to argument substitution.

We define the operational semantics as a single step reduction relation between contexts, $\rightarrow$, which is shown in Figure 4. Traversal of a constructed value proceeds by recurring on each field. Once all recursive results are completed (*i.e.*, reduce to values), a function is chosen based on the original object's type. For a simplified presentation and proof, a function is selected based only on the type of the originally traversed value (single dispatch)[1].

The meta-functions *type*, *types*, and *choose* are defined along with substitution in Figure 5. The *type* function simply returns the type name used in value construction (*i.e.*, reflection), while *types* returns the declared argument types of the given function. The implementation of *choose* selects the function in a set with a first argument that matches the given type. Substitution is the typical replacement of e' for all free occurrences of x in e. Note that variables are bound in functions, but functions are not the same as typical functional closures, since substitution does not occur inside function sets. This simplifies the traversal typing rules and proof by eliminating the need for a type environment in the traversal judgment, providing symmetry between runtime traversals and static traversal typings.

4. TYPE SYSTEM

For ease of presentation, our type system is divided into three separate judgments: expressions ($\vdash_e$), functions ($\vdash_F$), and traversals ($\vdash_{\mathcal{T}}$). All judgments are made in the context of a program's definitions, which provide a basis for the subtype relation ($\leq$).

4.1 Expressions : $\vdash_e$

Our type system is typical for non-traversal expressions, shown in Figure 6. Variables are looked up in a type environment, Γ, which is a list of variable/type pairs. The construction of *objects* requires each field to be a subtype of

[1]In later formulations (future work) *choose* will use the types of all recursive results, in addition to the original value's type (multiple dispatch).

the declared type. For traversal expressions we delegate to our traversal judgment, which determines the return type of a traversal of an instance of type T with function set F, notated $\langle T_0, F\rangle$. It begins with no *recursive types* ($\emptyset$) and the typing derivation must discharge all recursive *traversal constraints*.

4.2 Functions: $\vdash_F$

Functions are typed in the normal manner, typing the body expression with the argument names bound to their assumed types. The result type of a function is inferred from the argument types and body expression, though substitution could cause a subtype to be returned at runtime (considered in Section 5).

4.3 Traversals : $\vdash_{\mathcal{T}}$

The traversal typing judgment uses a set, $\mathcal{X}$, of recursed types (*i.e.*, a *stack*) to identify recursive type *uses* in concrete type definitions. A set of pairs, Φ, represents *traversal constraints* collected from recursive type uses. A constraint of (T, T') means the traversal of a value of type T must result in a subtype of T'.

We read the judgment $\mathcal{X} \vdash_{\mathcal{T}} \langle T, F\rangle : T'; \Phi$ as:

> *With recurred types $\mathcal{X}$, the traversal of a value of type T with function set F, returns a value of type T' with traversal constraints Φ.*

It is split into two rules shown in Figure 8; one each for concrete and abstract types. Essentially we connect the traversal of values to the static structural definitions in the program.

For traversal of a concrete type, C, we select the parameter types of the matching function in the given set, F. The meta-function $choose(F, C)$ selects a function in the set F which has C as the type of its first argument, and *types* returns the sequence of argument types. The type of the function eventually becomes the type of the traversal, but there are several conditions to be checked.

The recursive traversal of each non-recursive field with type T_i is typed by including C in the recursive type set. The result types, T_i', are required to be subtypes of the function's argument types, T_i''. If the recursive traversals generate constraints on C then we require the function's result type to be a subtype of the constrained type(s). New constraints are created for field types that exist in the set of recursed types, $\mathcal{X}$. Since the types of traversals of recursed types are unknown, we assume that the function argument types are the correct sub-traversal result types. The final constraint set is constructed from the union of field constraints by *discharging* those that involve C; the underscore (_) stands for *any* type.

The typing of the traversal of an abstract type, A, is slightly simpler since the traversal depends only on the results of the subtypes of A. Subtype traversals are typed by including A in the set of recursive types. The final return type is a common supertype, T, of the subtype traversal results. Constraints on A are checked the same as in the concrete case and new constraints are generated from recursive subtypes in $\mathcal{X}$, requiring a subtype of T as a traversal result. Constraints on A are likewise discharged.

5. TYPE SOUNDNESS

To prove type soundness we use the standard technique from [15] of proving preservation and progress. Most rules are similar to those in [5]. We begin with required lemmas.

LEMMA 1 (SUBSTITUTION PRESERVES TYPE). *If* $(\Gamma, x{:}T_x) \vdash_e e : T$, $\vdash_e e' : T'_x$, *and* $T'_x \leq T_x$ *then* $\Gamma \vdash_e e[e'/x] : T'$ *and* $T' \leq T$.

PROOF: By straight-forward induction on the structure of e, using the definition of substitution (Figure 5) and LEMMA 3. For a traversal expression we require that the traversal of a subtype return a subtype of the originally assigned type.

When we apply a function during traversal, substitution into the body always results in a subtype of the expected return type. The proof extends to the runtime expressions (`recur` and `apply`).

LEMMA 2 (COMPLETE FUNCTIONS). *For any well typed traversal expression* $e =$ `traverse(` e_0, F `)`, *the call* $choose(F, C)$ *will not fail.*

PROOF: By straight-forward induction on the typing derivation of e, using the traversal judgment ($\vdash_{\mathcal{T}}$) rules, T-CTRAV and T-ATRAV (Figure 8).

In our implementation, DemeterF, the main concern is actually LEMMA 2, since at runtime we must be able to select an applicable function (*i.e.*, advice) from a given function object during traversal. Other lemmas/theorems give the stronger result that the type of the value returned from traversal is predictable.

LEMMA 3 (SUBTYPE TRAVERSALS RETURN SUBTYPES). *For any well typed traversal expression* $e =$ `traverse(` e_0, F `)` *with* $\Gamma \vdash_e e_0 : T_0$ *and result type* T, *the traversal of* e'_0, *where* $\Gamma \vdash_e e'_0 : T'_0$ *and* $T'_0 \leq T_0$, *has result type* T' *with* $T' \leq T$.

PROOF: By induction on the typing derivation of e, using the traversal judgment rules, T-CTRAV and T-ATRAV.

Note that by our syntax and well-formed rules, there are no subtypes of a concrete type C, so the type of a construction expression will not change during evaluation. Our function selection ($choose(F, C)$) is deterministic and complete by LEMMA 2, and as such will return the same type for a given concrete traversal. The T-ATRAV rule also requires that the result of all subtype traversals be a subtype of the result type.

LEMMA 4 (WELL TYPED CONTEXTS). *For any closed expressions* e, e', *and context* E, *if* $\vdash_e e : T$, $\vdash_e e' : T'$ *with* $T' \leq T$, *and* $\Gamma \vdash_e E[e] : T_0$, *then* $\Gamma \vdash_e E[e'] : T'_0$ *and* $T'_0 \leq T_0$.

PROOF: By induction on the structure of the context E and the typing derivation of $E[e]$, using LEMMA 3.

This aids the preservation proof below, since each reduction occurs on limited expression forms and contexts.

THEOREM 1 (PRESERVATION). *If* $\vdash_e E[e] : T$ *and* $E[e] \rightarrow E[e']$ *then* $\vdash_e E[e'] : T'$ *with* $T' \leq T$.

PROOF: By straight-forward induction on the structure and typing derivation of $E[e]$, using LEMMAS 1, 3, and 4.

This gives the first half of soundness: reduction preserves type, also referred to as *subject reduction*.

THEOREM 2 (PROGRESS). *Suppose that e is a closed expression. If $\vdash_e e : T$ then either e is a value, or $e = E[e_0]$ and $E[e_0] \rightarrow E[e'_0]$.*

PROOF: By straight-forward induction on the structure and typing derivation of e.

We now have all requirements for the full theorem: well typed terms reduce to values of the expected type.

THEOREM 3 (TYPE SOUNDNESS). *Suppose that e is a closed expression and $\vdash_e e : T$, then either e is a value of type T, or $e \rightarrow e'$ and $\vdash_e e : T'$, with $T' \leq T$.*

PROOF: By PROGRESS and PRESERVATION theorems: e is either a value or can be reduced. If e reduces to e', then the type of e' (T') must be a subtype of T.

By PROGRESS and PRESERVATION we conclude that a well formed, well typed program will reduce to a value of predicted type, which allows us to precalculate the selection of certain functions from a set, or eliminate error checking from our dispatch implementation.

6. RELATED WORK

Most related work on semantics of Aspect Oriented Programming (AOP) Languages differs from our approach in that we do not describe a weaving semantics in order to provide a cleaner soundness proof. What we do share is a notion of dynamically selected advice (*i.e.*, *choose*), which is sometimes referred to as *advice lookup* [1, 6], implemented as *match-pcd* in [14].

In [7] the authors discuss the formulation of type safety for an AOP language in the theorem prover Coq, developing a more typical model. Our approach is specific to adaptive programming using traversals, which simplifies our soundness proof, but reduces the overall power of our model.

Similarly, in [13] the authors discuss an aspect extension to ML [10]. Labels are used to provide explicit join points, with first-class advice and side effects, providing most, if not all, of the flexibility of mainstream AOP languages. Around advice is similar to our function dispatch, though our syntax has been simplified as a first step in modeling our AP-F implementation.

Object Oriented type inference [11] has been used to provide a sound type system for a pure OO language using constraints. Constraints are generated and solved to establish that a *message-not-understood* error cannot occur. We have adapted typical typing rules based on their work and our experience with traversals. Ultimately their approach might lead to a simpler type checker, but full investigation is an item of future work.

7. CONCLUSION

We have presented the syntax, semantics, and type system of a restricted model of Functional Adaptive Programming (AP-F) and proven it type sound. AP-F provides a limited form of safe adaptive programming by way of functional, traversal-based aspects. The complication in the approach comes from the generalization of function set dispatch (*choose*), which delays function selection until recursive values are computed. This is done in order to later support a simple extension to unrestricted dispatch, as exists in our implementation.

7.1 Future Work

We are currently working on an unrestricted proof of type soundness with a full version of *choose* that selects a function based on all function arguments. With these results we hope to develop an approach for static dispatch of function objects during traversal, eliminating some of the overhead of reflection.

8. REFERENCES

[1] G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. μabc: A minimal aspect calculus. In *Proceedings of the 2004 International Conference on Concurrency Theory*, pages 209–224. Springer-Verlag, 2004.

[2] B. Chadwick. DemeterF: The functional adaptive programming library. Website, 2008. `http://www.ccs.neu.edu/home/chadwick/demeterf/`.

[3] B. Chadwick and K. Lieberherr. Functional Adaptive Programming. Technical Report NU-CCIS-08-75, CCIS/PRL, Northeastern University, Boston, October 2008.

[4] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *In POPL*, pages 171–183. ACM Press, 1998.

[5] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. In *TOPLAS*, pages 132–146, 1999.

[6] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *ECOOP*, pages 54–73, 2003.

[7] F. Kammüller and M. Voesgen. Towards type safety of aspect-oriented languages. In *AOSD 2006, FOAL Workshop*, 2009.

[8] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, ISBN 0-534-94602-X.

[9] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP*, pages 2–28, 2003.

[10] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[11] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *OOPSLA*, pages 146–161, New York, NY, USA, 1991. ACM.

[12] The Demeter Group. The DemeterJ website. `http://www.ccs.neu.edu/research/demeter`, 2007.

[13] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ICFP*, pages 127–139, New York, NY, USA, 2003. ACM.

[14] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5):890–910, 2004.

[15] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

$$E[\,\texttt{traverse(}\ \texttt{new}\ C\ \texttt{(}\ v_1,\ \ldots,\ v_n\ \texttt{)},\ F\ \texttt{)}\,] \\ \quad \rightarrow E[\,\texttt{recur(}\ F,\ \texttt{new}\ C\ \texttt{(}\ v_1,\ \ldots,\ v_n\ \texttt{)},\ \texttt{traverse(}\ v_1,\ F\ \texttt{)},\ \ldots,\ \texttt{traverse(}\ v_n,\ F\ \texttt{)}\ \texttt{)}\,]$$

$$E[\,\texttt{recur(}\ F,\ v_0,\ v_1,\ \ldots,\ v_n\ \texttt{)}\,] \rightarrow E[\,\texttt{apply(}\ \mathit{choose}(F,\ \mathit{type}(\,v_0\texttt{))},\ v_0,\ v_1,\ \ldots,\ v_n\ \texttt{)}\,]$$

$$E[\,\texttt{apply(}\ \texttt{(}T_0\ x_0,\ \ldots,\ T_n\ x_n\texttt{)\{ return}\ e\texttt{; }\texttt{\}},\ v_0,\ v_1,\ \ldots,\ v_n\ \texttt{)}\,] \rightarrow E[\ e[\overline{v_i/x_i}]\ ]$$

Figure 4: Reduction Rules

$$\mathit{type}\,(\texttt{new}\ C\ \texttt{(}\ v_1,\ \ldots,\ v_n\ \texttt{)}) \ =\ C$$

$$\mathit{types}\,(\texttt{(}T_0\ x_0,\ \ldots,\ T_n\ x_n\texttt{)\{ return}\ e\texttt{; }\texttt{\}}) \ =\ (T_0,\ \ldots,\ T_n)$$

$$\mathit{choose}\,(\texttt{funcset(}f\ldots\texttt{, (}C\ x_0,\ \ldots\texttt{)\{ return}\ e\texttt{; }\texttt{\}, }f\ldots\texttt{)},\ C) \ =\ \texttt{(}C\ x_0,\ \ldots\texttt{)\{ return}\ e\texttt{; }\texttt{\}}$$

$$\begin{array}{lcl} x[e'/x] & = & e' \\ x'[e'/x] & = & x' \quad \text{if } x' \neq x \\ \texttt{new}\ C\ \texttt{(}\ e_1,\ \ldots,\ e_n\ \texttt{)}[e'/x] & = & \texttt{new}\ C\ \texttt{(}\ e_1[e'/x],\ \ldots,\ e_n[e'/x]\ \texttt{)} \\ \texttt{recur(}\ F,\ v_0,\ e_1,\ \ldots,\ e_n\ \texttt{)}[e'/x] & = & \texttt{recur(}\ F,\ v_0,\ e_1[e'/x],\ \ldots,\ e_n[e'/x]\ \texttt{)} \\ \texttt{apply(}\ f,\ v_0,\ e_1,\ \ldots,\ e_n\ \texttt{)}[e'/x] & = & \texttt{apply(}\ f,\ v_0,\ e_1[e'/x],\ \ldots,\ e_n[e'/x]\ \texttt{)} \\ \texttt{traverse(}\ e_0,\ F\ \texttt{)}[e'/x] & = & \texttt{traverse(}\ e_0[e'/x],\ F\ \texttt{)} \end{array}$$

Figure 5: Reflection, Function Selection and Substitution Definitions

[T-Var]
$$\frac{x : T \in \Gamma}{\Gamma \vdash_e x : T}$$

[T-New]
$$\frac{\texttt{concrete}\ C\ \texttt{(}\ T_1, \ldots,\ T_n\ \texttt{)} \in P \qquad \Gamma \vdash_e e_i : T_i' \quad T_i' \leq T_i \ \text{ for all } i \in 1..n}{\Gamma \vdash_e \texttt{new}\ C\ \texttt{(}\ e_1,\ \ldots,\ e_n\ \texttt{)} : C}$$

[T-Trav]
$$\frac{\Gamma \vdash_e e_0 : T_0 \quad \emptyset \vdash_{\mathcal{T}} \langle T_0,\, F\rangle : T;\, \emptyset}{\texttt{traverse(}\ e_0\ ,\ F\ \texttt{)} : T}$$

[T-Recur]
$$\frac{\vdash_e v_0 : C \quad \Gamma \vdash_e \texttt{apply(}\ \mathit{choose}(F,\, C),\ v_0,\ e_1,\ \ldots,\ e_n\ \texttt{)} : T}{\Gamma \vdash_e \texttt{recur(}\ F,\ v_0,\ e_1,\ \ldots,\ e_n\ \texttt{)} : T}$$

[T-Apply]
$$\frac{\begin{array}{c}\vdash_e v_0 : C \quad \mathit{types}(f) = (C,\, T_1'',\, \ldots\, ,T_n'') \quad \vdash_F f : T \\ \text{for all } i \in 1..n \ \ \Gamma \vdash_e e_i : T_i' \ \wedge\ T_i' \leq T_i''\end{array}}{\Gamma \vdash_e \texttt{apply(}\ f,\ v_0,\ e_1,\ \ldots,\ e_n\ \texttt{)} : T}$$

Figure 6: Expression Typing Rules

[T-Func]
$$\frac{\overline{x_i : T_i} \vdash_e e_0 : T}{\vdash_F \texttt{(}T_0\ x_0,\ \ldots,\ T_n\ x_n\texttt{)\{ return}\ e_0\texttt{; }\texttt{\}} : T}$$

Figure 7: Function Typing Rule

[T-CTrav]
$$\frac{\begin{array}{c}\texttt{concrete}\ C\ \texttt{(}\ T_1,\ \ldots,\ T_n\ \texttt{)} \in P \quad \mathit{types}(\mathit{choose}(F,C)) = (C,\, T_1'',\, \ldots\, ,T_n'') \quad \vdash_F \mathit{choose}(F,C) : T \\ \text{for all } i \in 1..n \ \ T_i \notin \mathcal{X} \Rightarrow\ \mathcal{X} \cup \{C\} \vdash_{\mathcal{T}} \langle T_i, F\rangle : T_i'; \Phi_i \ \wedge\ T_i' \leq T_i'' \\ (C,\, T') \in (\Phi_1 \cup \cdots \cup \Phi_n) \ \Rightarrow\ T \leq T' \qquad \Phi = \{\ (T_j,\, T_j'')\ \mid\ j \in 1..n\ \wedge\ T_j \in \mathcal{X}\ \} \qquad \Phi' = \Phi \cup (\Phi_1 \cup \cdots \cup \Phi_n) \backslash (C,\ \_\,)\end{array}}{\mathcal{X} \vdash_{\mathcal{T}} \langle C, F\rangle : T; \Phi'}$$

[T-ATrav]
$$\frac{\begin{array}{c}\texttt{abstract}\ A\ \texttt{(}\ T_1,\ \ldots,\ T_n\ \texttt{)} \in P \\ \text{for all } i \in 1..n \ \ T_i \notin \mathcal{X} \Rightarrow\ \ \mathcal{X} \cup \{A\} \vdash_{\mathcal{T}} \langle T_i, F\rangle : T_i'; \Phi_i \ \wedge\ T_i' \leq T \\ (A,\, T') \in (\Phi_1 \cup \cdots \cup \Phi_n) \ \Rightarrow\ T \leq T' \qquad \Phi = \{\ (T_j,\, T)\ \mid\ j \in 1..n\ \wedge\ T_j \in \mathcal{X}\ \} \qquad \Phi' = \Phi \cup (\Phi_1 \cup \cdots \cup \Phi_n) \backslash (A,\ \_\,)\end{array}}{\mathcal{X} \vdash_{\mathcal{T}} \langle A, F\rangle : T; \Phi'}$$

Figure 8: Traversal Typing Rules

Modular Verification of Strongly Invasive Aspects

Emilia Katz Shmuel Katz
Computer Science Department
Technion – Israel Institute of Technology
{emika, katz}@cs.technion.ac.il

ABSTRACT

An extended specification for aspects, and a new verification method based on model checking are used to establish the correctness of strongly-invasive aspects, independently of any particular base program to which they may be woven. Such aspects can change the underlying base program variables to new states, and after the aspect advice has completed, the base program code continues from states that were previously unreachable. The needed changes in the MAVEN model checker are described, and the soundness of the verification method is proven. An example is shown of its application to aspects that provide various bonus points to student grading programs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model Checking*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Verification, languages

Keywords

Aspects, model-checking, specification, modularity

1. INTRODUCTION

Several works have dealt with model checking of aspect systems [8, 12, 5, 4, 9, 6]. These works either treat a system with aspects woven in, or try to deal with the aspects modularly, relative to a specification. In the later case, the motivation is either to reduce the size of the models, or to allow convenient reuse of aspects in a library. Such an approach requires that the aspect itself have an independent specification that can be shown to hold. In one form or another, the specification of an aspect describes an *assumption* about any base system to which the aspect can be woven, and a *guarantee* about the resultant system after the aspect is woven. The aspects are shown correct relative to their specification, and not to interfere with each other [6], and then, for each system to be constructed with the aspects, the base system is shown to satisfy the assumptions of the needed aspects. The construction of a model of the entire concrete woven system (which might be considerably larger than either of those used in the modular verification) and its direct verification do not have to be carried out at all.

So far, when aspects are treated separately from a specific weaving, it has been necessary to add a restriction: that the aspect returns control to the base system in a state that already existed for some computation of the base system without the aspect woven into it. Such aspects are called *weakly invasive* in [7], where the other categories of aspects mentioned in this paper are also defined. The reasoning behind the restriction is easy to understand: the aspect's assumption about the base system only relates to those computation sequences and states (known as *reachable* states) that can occur for some fair execution of the base system without the aspect. When an aspect returns control to the base system code, but in a state of the base variables that does not occur for any computation of the base system that begins from a "normal" initial state, there is no restriction on the behavior of the continuation. Instructions from the base code are executed, but with values that were never expected or tested, and with no restriction on the outcome. Thus the overall behavior of such a system is hard to analyze in a modular manner, separating the reasoning about the base from the reasoning about the aspects to be woven. In such cases, modular reasoning was thought unfeasible.

On the one hand, this restriction still allowed treating most aspects. Several kinds of aspects, including *spectative* ones that merely gather information, and *regulative* ones that merely restrict possible steps, are weakly invasive. Moreover, often the category of such aspects can be identified using dataflow techniques, as described in [7, 11, 13], and many commonly used aspect examples are weakly invasive. Nevertheless, there are other aspects that definitely are strongly invasive, and that occur in real applications, so that a more complete approach is desirable.

In this paper we show that such a restriction is unnecessary, and that a modular approach can be realized even for so-called *strongly invasive* aspects that do return control to the base system in new states that were unreachable in the base system executing alone. To do this, we take advantage of the usual organization of model checkers for linear

time systems, and of the facilities they commonly provide. An extension of the MAVEN aspect verification system is presented, that can treat strongly invasive aspects, and an example of a bonus aspect for student grades is described.

The basic idea of the new approach is to add to the specification an assumption about the base system that restricts the computation segments that may become reachable after a strongly invasive aspect is woven. We then show once-and-for-all that when the aspect is woven into any base system with a reachable part that satisfies the previous type of assumption and an unreachable part that satisfies the added one, the result of the weaving will satisfy the guarantee. For a particular base system, we then have to show that the assumptions are true for both the reachable and unreachable parts (or at least the unreachable part that may become reachable after weaving). These tasks are made feasible due to the fact that many model checkers actually generate a state transition system that includes the unreachable parts of the computation, as a side-effect of the construction, and that marking the reachable states is a built-in operation.

The original MAVEN system [4], over NuSMV [1], builds a single model that can be checked to establish the correctness of a weakly-invasive aspect relative to its assume-guarantee specification, given in Linear Temporal Logic (LTL). (In the examples in this paper we use only the LTL modalities $\mathsf{G}\,p$ - for "from now on, p", $\mathsf{F}\,p$ - for "eventually, p", and $p\ \mathsf{U}\ q$ - for "p is true until q becomes true"). The tableau state machine of the assumption is built using a module of NuSMV, and then the transition system of the aspect advice is woven into it, with pointcuts defining transitions to the beginning of advice state machine fragments, and with transitions back to the states of the base system that match the end states of the advice segments. It is then proven that whenever this particular model satisfies the guarantee assertion, then a woven system with any base satisfying the assumption, and the model corresponding to the aspect woven into it, satisfies the guarantee.

In the following section, precise definitions of the terms involved are presented, the theory behind the verification algorithm is described, and a proof of soundness is given, that extends the one given for the simpler MAVEN system. In Section 3 algorithms are given for computing the last states of the aspect, for determining the category of the aspect, for verifying the aspect, and for checking the base system for the needed assumptions. In Section 4 the specification and verification of an aspect for adding bonus points for student exercises and exams is described, and some concluding remarks are in Section 5.

2. VERIFICATION THEORY FOR STRONGLY INVASIVE ASPECTS

Definition 1. *An aspect A is* strongly invasive *relative to a model M if a state of M that was unreachable in M becomes reachable in the woven system M+A and transitions of M are applied to it.*

The last part of the definition is needed to ensure that the aspect advice (sometimes) finishes in a state of M that was previously unreachable, and then the code of M is applied to the new state.

2.1 Refined Aspect Specification

The assumption of a strongly invasive aspect has to contain more information than the assumption of a weakly invasive one: it sometimes needs to define restrictions on the behavior of the unreachable part of the base system into which the aspect can be woven, in order to ensure an appropriate behavior of the woven system from the states that are made reachable by the strongly invasive aspect.

The specification of aspect A is now a triple: (P_A, U_A, R_A), where, as before (in [4]), P_A is the assumption about the reachable part of the base system and R_A is the result assertion guaranteed to hold in the woven base with the aspect. The new U_A statement is an LTL formula defining the restrictions on the unreachable part of the base system which is made reachable by completing an aspect advice fragment. The restriction is posed on computations of the base system that start in the states that might be reached by completing the aspect advice, which were previously unreachable. We now may define the correctness of an aspect relative to such a specification, relating to a base system $S = S_{reach} \cup S_{unreach}$ where S_{reach} represents the reachable part, and $S_{unreach}$ the unreachable part.

Definition 2. *An aspect A is* correct *with respect to its refined assume-guarantee specification (P_A, U_A, R_A) if, whenever it is woven (by itself) into a system $S = S_{reach} \cup S_{unreach}$, where S_{reach} satisfies P_A and the part of $S_{unreach}$ that might become reachable after weaving satisfies U_A, the result will satisfy the guarantee, R_A.*

The property of the unreachable part of the system is relevant only for computation segments starting from a state that can be the last state of an advice execution. The reason is that only by an advice execution can a computation of the woven system pass from a state that was reachable in the base system to a state which was unreachable in the base system. Thus in order to check that the unreachable part of the base system satisfies the requirements of the aspect, it is enough to verify a formula of the form $L_A \rightarrow U_A$ on it, where L_A is a state formula describing the set of all the possible last states of the advice state machine, projected on the base system variables.

With some abuse of notation, we denote by L_A the set of possible last states of aspect A (identifying the unary predicate with the set it describes). Note that this set consists exactly of all the states in the base system into which a computation can arrive after finishing advice execution.

2.2 Refined Tableau Construction

Given an aspect A and its refined specification, (P_A, U_A, R_A), we need to construct a refined tableau to serve as a representation of all the base systems into which our aspect will possibly be woven. But now in order to build the tableau of the assumption of the aspect, it is not enough to build the tableau of P_A: we need to restrict the unreachable part of the tableau. The tableau needs to represent the systems, the reachable part of which satisfies P_A, and the unreachable part of which satisfies $L_A \rightarrow U_A$, where L_A is the predicate defining the set of all the possible return states of the advice. The refined tableau, T, is constructed in three steps:

Step 1: Automatically construct the predicate L_A. The construction is shown in Section 3.1.

Step 2: Use the `ltl2smv` module of the NuSMV model checker to build the tableau T_1 of the LTL formula $(P_A \vee (L_A \wedge U_A))$.

Step 3: Take the tableau T to be the same as T_1 except for the initial states definition. To obtain the $INIT$ predicate of T, restrict the $INIT$ predicate of T_1 to include only states that should be reachable in the base system: $INIT \wedge P_A$.

Note that T_1 is the tableau of $(P_A \vee (L_A \wedge U_A))$ and not of $(P_A \vee (L_A \rightarrow U_A))$, because the only way to reach the part of the base system that does not satisfy P_A is by application of an aspect advice, and this will bring the computation to a state in which L_A must hold. This intuition will be justified during the proof of Theorem 1.

Let us denote the refined tableau constructed as above by $T_{(P_A,(U_A,L_A))}$.

THEOREM 1. *Let A be an aspect with the refined assume-guarantee specification (P_A, U_A, R_A), and let L_A be a formula describing the set of all the possible last states of A. Then A is correct with respect to (P_A, U_A, R_A) if the result of weaving A into $T_{(P_A,(U_A,L_A))}$ satisfies R_A.*

We delay the proof of the theorem until after bringing some helpful definitions and lemmas needed for the proof. They appear below, together with the intuition for the proof.

In order to prove the theorem we need to show that if the result of weaving A into $T_{(P_A,(U_A,L_A))}$ satisfies R_A, then for every base system M such that its reachable part satisfies P_A and the unreachable part satisfies $L_A \rightarrow U_A$, the result of weaving A into M satisfies R_A. For this purpose it is enough to show that for every infinite fair path σ in the woven system $M + A$ there exists a corresponding infinite fair path π in the woven tableau, $T_{(P_A,(U_A,L_A))} + A$, such that $label(\sigma) \mid_{AP} = label(\pi) \mid_{AP}$. (Where AP is the set of all the atomic propositions appearing in the specification of A, a label of a state s, $label(s)$, is the set of all the atomic predicates that hold at the state s, and a label of a path τ, $label(\tau)$, is defined to be the sequence of the labels of the states of τ, so that if $\tau = s_0, s_1, s_2, \ldots$, $label(\tau) = (label(s_0), label(s_1), label(s_2), \ldots)$). In that case indeed in order to prove that every path in the woven system satisfies R_A, it is enough to show that every path in the woven tableau satisfies this property.

To simplify the notation, let us denote $T_{(P_A,(U_A,L_A))}$ by T. The task of finding a fair path in $T + A$ that corresponds to the given fair path of $M + A$ will be divided into steps according to prefixes of σ, and at each step a longer prefix will be treated. The following lemma will help to extend the treated prefixes:

LEMMA 1. *Let S be a system, and let $s_0, \ldots, s_k$ be states in S such that s_0 and s_k are reachable by a fair path from some initial state of S (the paths and the initial states for s_0 and s_k might be different), and for each $0 \leq j < k$, the transition (s_j, s_{j+1}) exists in S. Then there exists a fair computation in S which contains the sequence of states $s_0, \ldots, s_k$.*

Proof.
A computation is *fair* if it visits states from the *Fairness set* of the system model infinitely often. Let π_0 and π_k be fair computations in S in which s_0 and s_k occur, respectively. Then $\pi_0 = \sigma_0 \cdot s_0 \cdot \ldots$, and
$\pi_k = \ldots \cdot s_k \cdot \sigma_k$ for some σ_0 and σ_k. Let us take
$\pi = \sigma_0 \cdot s_0 \cdot s_1 \cdot \ldots \cdot s_k \cdot \sigma_k$. This is obviously a path in S, and it starts from an initial state, as did σ_0. Moreover, π is a fair computation, because it has the same infinite suffix, σ_k, as the fair computation π_k.
Q. E. D.(Lemma 1)

The following definition will be useful for identifying the "interesting" prefixes of the path σ:

DEFINITION 3. *Any infinite path π in a transition system can be represented as a sequence of* path segments - *$\pi = \pi^0 \cdot \pi^1 \cdot \ldots$, where each path segment π^i is a sequence of states such that:*

- *If $i = 0$, the first state of π^i is the initial state of π*
- *If $i > 0$, the first state of π^i is either an initial state of an advice or a resumption state of the base system (i.e., a state in the base system into which the computation arrives after an advice execution is finished)*
- *The last state of π^i is either a pointcut state or a last state of an advice (after which the computation returns to the base system), or the last state of the path, if π is finite*
- *There are no pointcut states and no last states of advice inside π^i (i.e., in the states of π that are not the first or the last state)*
- *π is the concatenation of the path segments of π in the order of their indices*

Note that the decomposition of a path to path segments is unique, and that, because of loops, there can be resumption states within a segment. Note also that we could have an infinite (last) segment - in the reachable part of the base, or in the unreachable part, or even in the aspect. In our case all the paths in question are infinite, so the last state of each finite path segment will be either a pointcut or a last state of an advice. A resumption state might be unreachable in the system before weaving - in case of a strongly invasive aspect.

Now if we are given a path of M+A, $\sigma = \sigma^0 \cdot \sigma^1 \cdot \ldots$ where σ^i-s are the path segments of σ, for each finite prefix of σ consisting of a number of path segments we define the set of *corresponding path-segment prefixes* of fair paths in T+A:

$$\Pi_i = \{\pi^0 \cdot \pi^1 \cdot \ldots \cdot \pi^i \mid label(\pi^0 \cdot \ldots \cdot \pi^i) \mid_{AP} = label(\sigma^0 \cdot \ldots \cdot \sigma^i) \mid_{AP}, \exists \pi \text{ fair path in } T+A \text{ such that } \pi = \pi^0 \cdot \ldots \cdot \pi^i, \ldots\}$$

Each element in Π_i is a prefix of an infinite fair computation of $T + A$ corresponding to the i-th prefix of σ, thus the following lemma will show that for every finite prefix of σ there exists a corresponding prefix of a fair computation in $T + A$:

LEMMA 2. *Given a fair computation σ of M+A, and sets of prefixes Π_i-s as defined above, $\forall i \geq 0. \Pi_i \neq \emptyset$.*

Proof.
The proof is by induction on i.

Base: $i = 0$.

To show that Π_0 is not empty we need to show the existence of π^0 such that $label(\pi^0) \mid_{AP} = label(\sigma^0) \mid_{AP}$ and π^0

is a prefix of some fair path π in T+A. σ^0 is the first path-segment of a fair path in $M+A$, thus there is no advice application before σ^0 or inside it. So σ^0 is also the first path segment of a fair computation in M. According to the assumption on M, $M \models P_A$, thus for every fair path starting from an initial state of M there exists a corresponding fair path in T. In particular, there exists a fair path $\pi = t_0, \ldots, t_k, \ldots$ in T such that $label(\sigma^0) \mid_{AP} = label(t_0, \ldots, t_k) \mid_{AP}$. Then again, as $t_0, \ldots, t_k$ is a beginning of a fair path in T, and there are no pointcuts in it, except maybe for the last state, it is also a beginning of a fair computation in $T + A$. So let us take $\pi^0 = s_0, \ldots, s_k$. We are left to show that π^0 is indeed a path-segment, and then it will follow that $\pi^0 \in \Pi_0$, meaning that Π_0 is not empty.

$label(t_0) = label((\sigma^0)_0)$, thus t_0 is an initial state of $T+A$. There is no pointcut inside σ^0, because it is a path-segment, so the last state of σ^0 cannot be a return state of advice application, which means that it has to be a pointcut state. Due to the agreement on labels, the last state of π^0 will also be marked as a pointcut state. For the same reason, there are no pointcut states among $t_0, \ldots, t_{k-1}$, which, in the same way as for σ^0, implies that there are no advice return states also. Thus both ends of π^0 are legal ends of a path-segment, and there are no pointcut states and no advice return states inside π^0, which makes it, indeed, a legal path-segment.

Induction step.

Let us assume that for every $0 \leq i < k$, $\Pi_i \neq \emptyset$. We need to prove that $\Pi_k \neq \emptyset$.

The induction hypothesis holds, in particular, for $i = k-1$, thus there exists some prefix $\pi^0 \cdot \pi^1 \cdot \ldots \cdot \pi^{k-1}$ of a fair computation of $T + A$, corresponding to the prefix $\sigma^0 \cdot \sigma^1 \cdot \ldots \cdot \sigma^{k-1}$ of $M + A$'s computation, σ. Let us denote by *s_first(i)* the first, and by *s_last(i)* the last state of $i - th$ path-segment of σ (σ^i), and symmetrically for the states of path segments of $T + A$ - by *t_first(i)* the first, and by *t_last(i)* the last state of $i - th$ path-segment. There are two possibilities for *s_last(k-1)*:

1. *s_last(k-1)* is a pointcut. Then *t_last(k-1)* is also a pointcut, because due to the induction hypothesis $label(s\_last(k\text{-}1)) \mid_{AP} = label(t\_last(k\text{-}1)) \mid_{AP}$. Then in every continuation of the computation both in $M + A$ and in $T + A$ the advice of the aspect will be performed, thus the k-th path-segment will in both cases be the application of the same advice from the same state, and the agreement on the labels of the k-th path-segments will be trivially achieved. Moreover, for the same reason the existence of an infinite fair path with the prefix $\pi^0 \cdot \pi^1 \cdot \ldots \cdot \pi^{k-1}$ implies the existence of an infinite fair path with the prefix $\pi^0 \cdot \pi^1 \cdot \ldots \cdot \pi^k$, because every continuation of the first prefix had to be an advice application. From the above it follows that in this case $\Pi_k \neq \emptyset$.

2. *s_last(k-1)* is a last state of the advice. This, in particular, implies that *s_last(k)* is a pointcut state, and no advice has been applied between *s_last(k-1)* and *s_last(k)*. Here are again two possibilities:

   - *s_last(k-1)* is a reachable state in M (more precisely, the state reachable in M is the projection of *s_last(k-1)* on AP). As no advice is applied between *s_last(k-1)* and *s_last(k)*, we have that the whole path-segment σ^k is in the reachable part of M. Moreover, due to Lemma 1, as both *s_last(k-1)* and *s_last(k)* are reachable by some fair paths from some initial states of M, we also have that there exists a fair computation of M containing the sequence *s_last(k-1)*, *s_first(k)*, ..., *s_last(k)*. All the fair computations of the reachable part of M are represented in the tableau of P_A, which is exactly the reachable part of T. Thus, in particular, the above fair path has a corresponding path in T, and, as there was no pointcut or advice application inside the sequence *s_last(k-1)*, *s_first(k)*, ..., *s_last(k)*, there are also no pointcuts and advice applications in the corresponding sequence in the computation of T, and thus there exists a corresponding sequence of states in $T+A$, π^k. The first state of π^k, *t_last(k-1)*, is reachable from the initial state of $T + A$ by some fair path, as Π_{k-1} is not empty. Moreover, all the prefixes of such fair pathes appear in Π_{k-1}, thus at least one of them continues to the sequence π^k. So indeed we obtain that there exists a sequence of states π^k corresponding to σ^k in the woven tableau, for which a fair continuation exists. We are left to see that the sequence of states, π^k, is indeed a path segment in the woven tableau computation. But this is true due to the agreement on labels of the states, $label(\pi^k) \mid_{AP} = label(\sigma^k) \mid_{AP}$: the path segment σ^k started from a return state of an advice, ended by a pointcut, and had no advice applications in the internal states, so the same is true for π^k and thus π^k is a path segment.

   - The last case left is that *s_last(k-1)* is unreachable in M. Additionally, *s_last(k-1)* is the last state of the advice, thus *s_first(k)* is the return state of the advice, and also is unreachable in M, because according to the weaving algorithm $label(s\_last(k\text{-}1)) \mid_{AP} = label(s\_first(k)) \mid_{AP}$. From the fact that *s_first(k)* is unreachable in M, together with the assumption on the unreachable part of M, we have that $L_A \rightarrow U_A$ holds in the suffix of any path starting from *s_first(k)*. But from the agreement on labels with *s_last(k-1)* we also have that *s_first(k)* $\models L_A$. Together we obtain that U_A holds in the suffix of any computation in M starting from *s_first(k)*, and, in particular, for the computation $\sigma\prime$ containing the next path segment of σ, σ^k (because there is no advice application inside σ^k, all its states are states of the original system, M - either in the reachable or the unreachable part). Now let us examine the states of the woven tableau. The tableau of $L_A \wedge U_A$ is included in the refined tableau T, thus every computation satisfying U_A that starts from a state satisfying L_A is represented in T (though its initial state might be unreachable before the aspect is woven into T). Let $\pi\prime$ be a computation that corresponds to the suffix of $\sigma\prime$ that starts from *s_first(k)*. The first state of $\pi\prime$ agrees on its label with *s_first(k)*, and thus with *s_last(k-1)*, which, according to the induction hypothesis, implies agreement on labels with *t_last(k-1)*. According to the weaving algorithm, the last state

of the advice is connected to all the states in the underlying system with which it agrees on labels. Thus, in particular, *t_last(k-1)* (which is the last state of the advice, in the same way as *s_last(k-1)*), is connected to the first state of $\pi\prime$. So we can take the first state of $\pi\prime$ to be the first state of π^k. Let us then take π^k to be the first path-segment of $\pi\prime$. It is indeed a path segment of a fair computation (due to Lemma 1), it is connected to π^{k-1} and agrees on labels with σ^k, so we found what we needed.

Thus, indeed, the set of possible continuations, Π_i, is never empty.
Q. E. D.(Lemma 2)

Theorem 1 proof:.

Now let us return to the proof of Theorem 1. Let us be given an infinite fair path σ in the woven system $M + A$. From Lemma 2 it follows that there exists an infinite path π in the woven tableau corresponding to the given path σ - all the prefixes of π appear in the Π_i-s above, and due to the lemma, the Π_i-s are all non-empty. So in order to complete the proof of the theorem we need only to notice that every path constructed from the prefixes in Π_i-s above is fair, for the following reason: There are two possibilities for the infinite suffix of π. It either has infinitely many advice applications, or there exists some infinite suffix in which no aspect state is visited. If there are infinitely many advice applications, some state of the advice must be visited infinitely often, and all the states of the advice are defined as fair. If there is no advice application after some state, then there are only a finite number of path segments of π, and the last path segment is infinite. But, as we know, this path segment belongs to some fair path in $T + A$, so this must be a fair suffix, and so the computation π is indeed fair. This completes the proof of Theorem 1
Q. E. D.

3. ALGORITHMS

3.1 Computing L_A Automatically

Given a model of the aspect, A, in MAVEN format, we would like to automatically compute the state formula defining the set of all the possible last states of A's advice. The algorithm we propose consists of four steps:

Step 1: Construct a formula φ defining the pointcut of the aspect: take φ to be the disjunction of all the POINTCUT expressions in A.

Step 2: Run MAVEN on a model $A\prime$ which is the same as A except for a change in the specification. The assumption of the aspect is replaced by φ, and the guarantee of the aspect is replaced by *true*. The purpose of this operation is to obtain a system in which all the possible computations of the aspect are represented, and this goal is achieved in the following way:

- At the first step of its work, MAVEN will automatically construct the tableau of the new assumption of the aspect, φ, using the `ltl2smv` module of NuSMV. Note that in this tableau, T_φ, only the initial states are restricted, and the initial states are exactly all the possible join-points of the aspect.
- At the second step, MAVEN will perform the weaving of the aspect into the constructed tableau. The obtained woven system, $T_\varphi + A$, will contain all the possible computations of the aspect, because the initial states of the tableau are all the possible pointcut states that can occur in either reachable or unreachable parts of the base systems into which A will be woven (as the ranges of all the base variables as defined in the aspect model definition are the maximal possible, and the combinations of variables values are restricted only by the formula φ).

Note that if we added other restrictions on the computations of the tableau T_φ, we may not be able to guarantee that all the possible runs of the advice of A will appear in the woven tableau. For example, if we demand that the computations of the tableau should satisfy P_A, then after the weaving we would not obtain the runs of the aspect from the states that were unreachable in the base system. Since in the unreachable part of the base system which becomes reachable after the weaving there might be join-points of A, we have to model the computations of the advice starting from these states. However, there are cases when additional restrictions might be posed on the computations of the tableau built. For example, there might be some invariant that holds both in the reachable and the unreachable parts of the base system, and then it could be added to φ. Additionally, there might exist an assertion that holds for all the pointcut states, but is not explicitly written as part of the pointcut. Then it would be possible to restrict the initial states of the constructed tableau by this assertion.

Step 3: Take the woven system obtained in Step 2, $T_\varphi + A$, and use the built in functionality of NuSMV to compute the set of all the reachable states of this model, $(T_\varphi + A)_{reachable}$. For each of the states in $(T_\varphi + A)_{reachable}$, check whether it satisfies any of the RETURN conditions of the aspect. If it does, add it to the set L_A.

Step 4: Now L_A is the set of all the possible last states of A. What is left is only to construct the predicate describing this set. This is done by taking the disjunction of all the predicates describing the states in L_A.

Sometimes it might be easy to see a compact description of the possible last states of the aspect. For this case we provide the user a possibility to supply a manually constructed predicate L. But such a predicate should be checked before use, because the intuition of the user might be wrong. Then we use the above algorithm to construct the full L_A predicate, and check that the supplied predicate L is implied by L_A. If indeed $L_A \rightarrow L$ holds, the verification using L will still be sound, because it just might check additional paths, but no relevant path will be left unverified.

3.2 Determining the Aspect Category

Before applying the full verification technique it is very desirable to determine the category of the aspect. If the aspect is of the weakly invasive category (or a simpler category included within the weakly invasive one), then the method described in [4] is applicable to it. Otherwise, the method described in Section 3.3 should be used.

Some ways of determining the category of the aspect using code analysis, dataflow techniques and semantic definitions are described in [7, 11, 13, 3]. If none of them gives a positive answer, the algorithm presented below can help to determine whether the aspect is uniformly strongly inva-

sive, i.e., is always strongly invasive for every possible base system to which it can be woven. But first some definitions and observations are needed:

REMARK 2. *From Definition 1 in Section 2 it immediately follows that for any system M in which all the states not reachable from the initial state by some fair path have been removed, if an aspect A is strongly invasive relative to M, there is a deadlock in the system $M + A$: Let s be a last state of advice execution such that there exists no reachable state $s\prime$ in M for which $label(s\prime) = label(s) \mid_{AP}$. Then this state is a deadlock state in the woven system.*

LEMMA 3. *Let aspect A have the specification (P_A, U_A, R_A), where AP is the set of all the atomic propositions appearing in the specification and T_P denotes the tableau of P_A. Aspect A is* strongly invasive *with respect to P_A if when A is woven into T_P, there exists a state s in $T_P + A$ such that:*

- *s is the last state of advice execution, and*
- *there exists no state $s\prime$ in T_P such that $s\prime$ is reachable by some computation of T_P and $label(s\prime) = label(s) \mid_{AP}$*

Proof.
Immediate from the above remark.

DEFINITION 4. *Given a tableau T of an LTL formula ϕ, the tableau TP obtained from T by removing all the states that are not reachable from the initial state of T by any fair path (and only them) is called the* pruned tableau *of ϕ.*

Note that the above defined pruned tableau is equivalent to a tableau obtained from T by removing all the states and transitions that only lead to deadlock states.

LEMMA 4. *Aspect A with the specification (P_A, R_A) is strongly invasive relative to P_A iff there exists a deadlock in the system $TP_A + A$, where TP_A is the pruned tableau of P_A.*

Proof.
The conditions of Remark 2 above hold, in particular, for $M = TP_A$, so there will be a deadlock state in $TP_A + A$.

On the other hand, if there exists a deadlock in the system $TP_A + A$, let s be the deadlock state. Let us denote by $s\prime$ the state of TP_A such that $label(s\prime) = label(s) \mid_{AP}$. There are two possibilities: If $s\prime$ is reachable in TP_A, then there exists some infinite computation $\pi = s\prime, s_2, \ldots$ from $s\prime$ in TP_A, because TP_A is a pruned tableau. In particular, there exists a state s_2 in TP_A (the second state of π) to which $s\prime$ is connected. However, in $TP_A + A$ the state s is no longer connected to s_2. According to the construction of $TP_A + A$, the only reason could be that an advice is applied at s. But if an advice was applied at s, s would not be a deadlock state. Thus when we assumed that the projection of s on AP is reachable in TP_A we obtained a contradiction. So we conclude that $s\prime$ is unreachable in TP_A.

But could $s\prime$ still be reachable in T_P? This can only be if $s\prime$ has been removed from T_P during the construction of the pruned tableau. This means that all the paths starting from $s\prime$ led to some deadlock states, and thus $s\prime$ couldn't be reached by any fair computation of T. But according to Lemma 3 this exactly means that the aspect A is strongly invasive relative to its assumption.
Q. E. D.

According to Lemma 4, the following algorithm verifies whether the given aspect is strongly invasive relative to its assumption:

1. Construct the pruned tableau TP_A from the tableau of the assumption of A. This is done automatically, by an iterative procedure that we have added to MAVEN. The procedure is as follows:
   - Run NuSMV to detect deadlock states in the tableau.
   - If a deadlock state is detected, construct a predicate describing this state, p
   - Rule out the deadlock state: Add the negation of p to the initial state definition, and to the predicate defining possible next states of the transitions.

   Repeat the procedure until there are no more deadlocks in the tableau.
2. Use MAVEN to weave the aspect into the above constructed tableau.
3. Run NuSMV to check whether there are deadlocks in the woven tableau. If a deadlock is detected, the aspect is strongly invasive relative to its assumption. Otherwise, the aspect A is weakly invasive relative to P_A.

Note that the algorithm presented here gives a positive answer only if the aspect is strongly invasive *relative to the tableau of its assumption*, but not relative to a concrete base system. Thus if the algorithm gives a positive answer, the aspect is strongly invasive relative to all the possible base systems into which it might be woven. But if the algorithm gives a negative answer, there might exist a base system satisfying the assumption of the aspect, with respect to which our aspect is still strongly invasive.

Given a base system S, there is one more way for us to check whether the given aspect, A, is strongly invasive relative to this system. Intuitively, what we would like to do is to look at all the unreachable states of the base system, and check whether there are last states of our aspect among these unreachable states. For that purpose we can check satisfiability of the following formula: $\varphi = S_U \wedge L_A$, where S_U is the formula defining the set of all the unreachable states of S, and L_A is the formula defining the set of all the possible last states of A. φ can be constructed automatically: the way to construct L_A automatically is shown in Section 3.1, and the way to construct S_U automatically is shown in Section 3.4.1. And then the satisfiability of φ can be automatically checked using a SAT solver (such as, for example, Chaff [10]). If φ is found unsatisfiable, it means that there are no last states of the aspect A in the unreachable part of S, so A has to be weakly invasive relative to S, and the simpler model check in [4] can be used. If φ is found satisfiable, it doesn't necessarily imply that A is strongly invasive relative to S, because the predicate L_A is an over-approximation: it contains all the possible last states of the aspect, but maybe some of them will never occur in the computations of the woven system $S+A$, and thus will not bring the computation to states that were unreachable in S. But this over-approximation is a safe one: if we declare some aspect as strongly invasive

when it is weakly invasive, we will just have to work harder to prove its correctness than we would if we knew its exact category, but the verification results will be sound.

3.3 Verifying the Aspect

Given an aspect A and its refined assume-guarantee specification, (P_A, U_A, R_A), the verification of correctness of A with respect to (P_A, U_A, R_A) is performed as follows:

1. Construct the refined assumption tableau for A as shown in Section 2.2 - the $T_{(P_A,(U_A,L_A))}$.
2. Use MAVEN to weave A into $T_{(P_A,(U_A,L_A))}$ and to run the NuSMV model checker on the resulting system and check the R_A property on it.

3.4 Base System Correctness Verification

3.4.1 Non-optimized solution

Given a base system S, we need to verify that it satisfies the refined assumption of our aspect, (P_A, U_A):

- Verify that the reachable part of S, S_{reach}, satisfies P_A
- Verify that all the computations starting from the unreachable part of S, $S_{unreach}$, satisfy $L_A \rightarrow U_A$.

The first verification task can be done by usual model-checking of S versus P_A. The meaning of the second task is as follows: we need to examine the model of $S_{unreach}$ and check all the fair computations that start from states satisfying L_A (note that a computation starting from a state in $S_{unreach}$ might return to the reachable part of S at some state). All these computations should satisfy U_A. The verification is performed in three steps:

1. Automatically compute the state formula S_U defining the set of all the unreachable states of S: S_U is the negation of the formula S_R defining all the reachable states of S, and in NuSMV there exists a possibility to compute S_R automatically for a given system S.
2. In the model of the base system, S, automatically replace the initial states definition by the formula $S_U \wedge L_A$
3. Run NuSMV on the obtained model and the formula U_A. If the verification succeeds, it means that the given base system satisfies the restriction on the unreachable part.

3.4.2 Optimization

In some cases, the requirement in the second part of the verification process can be relaxed due to the structure of U_A. For example, in case when U_A is some safety property, i.e., U_A has the form $\mathsf{G}\,\varphi$, we do not have to verify that φ holds all along the computations starting from resumption states in the unreachable part of the system. We need to check only the segments between a resumption state and the next join-point or reachable state. So if we denote by *ptc* the predicate defining the pointcut of the aspect, and by *reachable* - the predicate defining the reachable states of the base system, then it is enough to verify the following formula on the unreachable part of the system: $L_A \rightarrow (\varphi\ \mathsf{U}\ (\mathit{reachable} \vee (\mathit{pointcut} \wedge \varphi)))$. The reason is that when the computation reaches a join-point, in the woven system the advice will be executed at that point, so the information about the possible continuations of the computation in the base system from that point is useless. And if a computation leaves the unreachable part and arrives to some previously reachable state, its continuation will behave as specified by the assumption of the aspect about the reachable part of the base system, and all these continuations are already checked during the reachable part verification.

As an example of the situation described above, we can take a look at an aspect that is in charge of the scheduling policy of a semaphore-guarded resource. The purpose of the aspect is to implement a possibility of a waiting queue for the semaphore. As a result, the semaphore that could previously have only values 0 or 1 can now have negative values (according to the number of waiting processes). Thus the aspect is indeed strongly invasive. But there is a part of the system invariant that we need to extend to the unreachable part of the base system: regardless of the semaphore value and the concrete scheduling algorithm, we demand that no two processes hold the guarded resource at the same time. So if the formula ψ encodes the fact that two processes hold the resource at the same time, the assumption of the aspect about the unreachable part of the base system should be $U = \mathsf{G}\neg\psi$. But when verifying the computations starting in the unreachable part of the base system, it is enough to check that after each possible last state of the aspect the computation satisfies $\neg\psi$ until it arrives to a pointcut state or to a reachable state.

4. EXAMPLE

In this example we discuss an aspect that can be used in any grades-managing system. The aspect B provides a way of giving bonus points for assignments and/or exams (thus making it possible to have assignment/exam grades that are more than 100), but still keeping the final grade within the 0..100 range.

The aspect has two kinds of pointcuts, and two corresponding pieces of advice. The first pointcut of B is the moment when an assignment or exam grade is entered to the system. At this point the original system would accept only grades between 0 and 100, but the aspect offers a possibility of giving a bonus on the grade, and stores the new grade successfully even if it exceeds 100. The second pointcut of B is the moment when the final grade calculation of the base system is performed. Then if the calculation resulted in a grade that exceeds 100, the aspect replaces this grade by 100 (otherwise keeping the grade unchanged).

Aspect B is strongly invasive in the systems into which it can reasonably be woven, because its operation results in states in which some grades are more than 100, which is impossible in the base systems without bonus policies. And this example, though simple, is still of interest to us, because the aspect here exhibits a typical behavior we would like to treat: when it is woven into a system, the calculations there are performed partly in the aspect, and partly in the base system code, but using new inputs, that were impossible before the aspect was woven in.

The specification of B can be formalized as follows:

- The assumption on the reachable part of the base system is that all the grades appearing in the grading system - homework assignment grades (*hw_i*), exam grades (*exam_j*), final grade (*f*) - are between 0 and

100, and after the final grade is ready (*f_ready*) (i.e., all the assignments and exams that comprise the grade have been checked, and the final grade has been calculated from them according to the base system grading policy), the final grade is published (*f_published*). The result of the final grade calculation is represented by *calc*.

$$\begin{aligned} P_B = [\, & \mathsf{G}(f\_ready \rightarrow ((f = calc) \wedge \mathsf{F}\, f\_published)) \\ & \mathsf{G}(f\_published \rightarrow f = calc) \wedge \\ & \mathsf{G}(0 \leq f \leq 100) \wedge \\ & \mathsf{G}(\forall 1 \leq\ i \leq 10(0 \leq hw\_i \leq 100)) \wedge \\ & \mathsf{G}(\forall 1 \leq\ j \leq 2(0 \leq exam\_j \leq 100))] \end{aligned}$$

Here, for modeling purposes, we have to provide some bounds on the number of assignments and exams, so we assume that there are no more than 10 home assignments and no more than 2 exams in each course. We also show the specification for the grades of a single student (because the grades of different students are independent, and calculations involving them can be viewed as orthogonal). When the model of the aspect is built, the ranges of all the variables - both the aspect variables and the relevant base system ones - are defined. Let us assume, for example, that our aspect gives bonuses in range of 0..20 points, then all the grade variables defined in the model of B are in the range 0..120.

- The assumption on the unreachable part of the base system is in our case a weakening of P_B. We still want the final grades to be published after they are ready, but now the final and the intermediate grades do not have to be bound by 100, but by 120. So we are left with the following property:

$$\begin{aligned} U_B = [\, & \mathsf{G}(f\_ready \rightarrow ((f = calc) \wedge \mathsf{F}\, f\_published)) \wedge \\ & \mathsf{G}(f\_published \rightarrow f = calc) \wedge \\ & \mathsf{G}(0 \leq f \leq 120) \wedge \\ & \mathsf{G}(\forall 1 \leq\ i \leq 10(0 \leq hw\_i \leq 120) \wedge \\ & \mathsf{G}(\forall 1 \leq\ j \leq 2(0 \leq exam\_j \leq 120))] \end{aligned}$$

- The guarantee of the aspect now is that regardless of the existence of bonuses on the components of the final grade, the final grade will be the one calculated by the base system function, but rounded down to 100 if needed:

$$R_B = [\, \mathsf{G}(f\_published \rightarrow f = min(calc, 100))]$$

The guarantee of the aspect might also include a statement about the bonus policy it enforces, saying that the aspect calculates the bonuses as desired. But to simplify the discussion, we omit it here.

- The pointcut of the aspect can be formalized using the following predicates, which define the moments when the grades are entered into the system: *enter_hw_i* for homework grades, and *enter_exam_j* for exam grades.

$$\begin{aligned} Pointcut_B = [(\bigvee_{i=1}^{10} & (enter\_hw\_i)) \vee \\ & (enter\_exam\_1) \vee (enter\_exam\_2) \vee \\ & (f\_ready \wedge (f > 100))] \end{aligned}$$

Let us follow the verification algorithm, applying it to aspect B. The first step is the refined tableau construction. It begins with calculating the predicate L_B, defining all the possible last states of B. In our example, we get

$$\begin{aligned} L_B = [& (f\_ready \rightarrow ((f = 100) \wedge (calc > 100))) \wedge \\ & (\neg f\_published) \wedge \\ & \forall 1 \leq\ i \leq 10(\neg enter\_hw\_i) \wedge \\ & \forall 1 \leq\ j \leq 2(\neg enter\_exam\_j) \wedge \\ & (0 \leq f \leq 120) \wedge (0 \leq calc \leq 120) \wedge \\ & \forall 1 \leq\ i \leq 10(0 \leq hw\_i \leq 120) \wedge \\ & \forall 1 \leq\ j \leq 2(0 \leq exam\_j \leq 120)] \end{aligned}$$

And here is the explanation: All the combinations of exams and assignments grades values in range 0..120 are possible at the last state of the aspect, because all the grades of assignments and exams are independent. There is a connection between the final grade and the other grades, but only when the final grade is declared to be ready and still is not published. Then the final grade is equal to the minimum between the calculated value (*calc*) and 100. However, as we do not want to restrict the calculation function of the base system, we cannot establish this connection, and at the other states of the computation the value of the final grade is not restricted (except by its range), so effectively we have to enable any combination of the final grade value and the other grades. The values of the other system variables are restricted as follows: The variables *enter_hw_i* and *enter_exam_j* for all i-s and j-s are $false$, because no grade is entered by the user at the last state of the advice. The variable *f_published* is also $false$, because the aspect does not publish the grades - even if it was called at the moment when the final grade was calculated, it just modifies the calculated grade, but does not publish it. Publishing the grades is done by the base system. The next variable to discuss is *f_ready*. If the aspect was called at the moment of grades entering, the variable *f_ready* is $false$ at the join-point. The final grade is not calculated by the aspect in this case, so the variable remains $false$ at the last state of the advice. However, if the aspect was called at a join-point when the final grade is calculated, the variable *f_ready* is true there and remains true after the advice finishes its execution. In this case, as we said earlier, we will also have $f = 100$ and $calc > 100$.

Now after the predicate L_B is constructed, the tableau of the $(P_B \vee (L_B \wedge U_B))$ formula is created, its initial states are restricted to those satisfying P_B (that is, the refined tableau $T_{(P_B,(U_B,L_B))}$ is built), and then B is woven into the result. The last part of the verification process is running NuSMV on the woven tableau in order to check the R_B property on it. And for the above described aspect, with the specification given, the verification succeeds, so our algorithm shows that indeed it is correct with respect to its refined assume-guarantee specification. Intuitively, the reason for the success of the verification is that the base system performs only some arithmetic operations on the grades the aspect modifies, and thus we can expect that the result of performing old operations on the new arguments will be as anticipated, if only there is no overflow or type declaration problem. (By a type declaration problem we mean, for example, the case when the type of the grades variables is defined in the base code by some *typedef* to be 0..100, so that larger values

cause a fatal type error.) But the assertion U_B ensures that this will not happen, because U_B will not hold for the base system in case such problems arise.

Note that the aspect does not restrict the grade calculation process of the base system, so this aspect is highly reusable, as long as the calculation can handle values greater than 100 (as seen in U_B). Moreover, this aspect can appear in a library of aspects providing different grading policies: different types of bonuses for homework assignments, or factors on the exam grades. All these aspects will have the same requirements from the base system as B does, so when some grading system is checked for applicability of one of the aspects from this library, it is automatically inferred that all the other aspects from the library are also applicable to this base system. Thus the grading policy can be changed as needed at any time, by replacing the applied aspect, without any further checks on the base system.

5. CONCLUSIONS

We have shown that strongly invasive aspects can be specified and shown correct relative to their specification, independently of a particular base system. Moreover, it is reasonable to check the properties needed from the unreachable part of the base system because the possible transitions of the base are considered bottom up, independently of the initial states, thus generating the unreachable part of the base system as a byproduct of model checking. Strongly invasive aspects typically extend the functionality of the base system to situations not originally covered. The examples seen in the paper, of a semaphore with negative values, and of aspects to give bonus points beyond the normal range, are typical. Often some invariants true in the base system alone will no longer hold after weaving such aspects, but other invariants will continue to hold, and are essential to the correctness of the woven system.

The verification method presented here is modular, and thus has an advantage over a straightforward non-modular verification of a woven system: the possibility of reuse without proof. There are two types of such reuse we see, both of which are demonstrated by the aspect described in Section 4. One case is when one and the same aspect is applicable to different base systems. Then the verification of the advice versus the assume-guarantee specification is performed only once, and in order to be able to apply the aspect to a given base system we need only to perform the base system verification described in Section 3.4. Another case is when a library of aspects is given, where all the aspects are built for the same purpose (like defining some action policy) and have a common assumption (P, U) about the base system. Then if we have a base system that satisfies the above assumptions, we can change the policy defined in this system at any time, by applying different aspects from the library - one at a time, of course - without any further checks.

When model-checking is used, the size of the verified system and of the specification is very important, as it strongly affects the verification time, and sometimes, if the model verified is too large, the model-checker can even fail to provide any answer. For the complexity analysis purpose, we denote by m the size of the base system model, by a - the size of the aspect model ($|A|$), by r - the size of all the formulas in A's specification (assuming, without loss of generality, that all the formulas used in verification - P_A, U_A, R_A - are approximately of the same size). When a formula of size k is verified on a model of size m, the space complexity of the model checking is $O(m \cdot 2^k)$ ([2]). Thus the complexity of a straightforward verification of the woven system is $O(2^r \cdot (m \cdot a))$, because a system of size m is verified against a formula of size r (the guarantee of A, in this case). Let us find the complexity of the modular verification method. It is the sum of the following components:

- The verification of the base system. It is of $O(2^r \cdot m)$ for the reachable part, and the same for the unreachable part, so together we obtain $2 \cdot O(2^r \cdot m) = O(2^r \cdot m)$
- Verification of the aspect. Here, first the refined assumption tableau is constructed, $T_{(P_A,(U_A,L_A))}$. The complexity of this step is $O(2^{2r})$ (Note that L_A is always a state formula, and thus does not increase the complexity.) Then the woven tableau is built, and we obtain a system of size $O(2^{2r} \cdot a)$. At the last step, the woven tableau is verified against the guarantee of A, R_A, and this requires complexity of $O(2^r \cdot (2^{2r} \cdot a))$

The total complexity thus is $O(2^r \cdot (2^{2r} \cdot a)) + O(2^r \cdot m)$. But the size of the base system model is usually very large, so $m \geq 2^{2r}$, and thus the complexity of our verification is usually not worse than that of the straightforward woven system check. Even when this is not the case, the possibilities for reuse make the modular approach preferable.

6. REFERENCES

[1] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proc. Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in LNCS, pages 495–499. Springer, July 1999. NuSMV home page: `http://nusmv.itc.it`.

[2] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[3] S. Djoko Djoko, R. Douence, and P. Fradet. Aspects preserving properties. In *Proc. of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM'08)*, pages 135–145. ACM, 2008.

[4] M. Goldman and S. Katz. MAVEN: Modular aspect verification. In *Proc. of TACAS 2007*, volume 4424 of *LNCS*, pages 308–322, 2007.

[5] E. Katz and S. Katz. Verifying scenario-based aspect specifications. In *Proc. Formal Methods: International Symposium of Formal Methods Europe (FM'05)*, volume 3582 of *LNCS*, pages 432–447. Springer, 2005.

[6] E. Katz and S. Katz. Incremental analysis of interference among aspects. In *Proc. of the 7th workshop on Foundations of aspect-oriented languages FOAL '08*, pages 29–38. ACM, 2008.

[7] S. Katz. Aspect categories and classes of temporal properties. *Transactions on Aspect Oriented Software Development (TAOSD)*, 1:106–134, 2006. LNCS 3880.

[8] S. Katz and M. Sihman. Aspect validation using model checking. In *Proc. of International Symposium on Verification*, LNCS 2772, pages 389–411, 2003.

[9] S. Krishnamurthi and K. Fisler. Foundations of incremental aspect model-checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2), 2007.

[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. of the 38th Design Automation Conference, DAC'01*, pages 530–535, 2001.

[11] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proc. of International Conference on Foundations of Software Engineering (FSE04)*, 2004.

[12] H.B. Sipma. A formal model for cross-cutting modular transition systems. In *Proc. of Foundations of Aspect Languages Workshop (FOAL03)*, 2003.

[13] N. Weston, F. Taiani, and A. Rashid. Interaction analysis for fault-tolerance in aspect-oriented programming. In *Proc. Workshop on Methods, Models, and Tools for Fault Tolerance, MeMoT'07*, pages 95–102, 2007.

Unweaving the Impact of Aspect Changes in AspectJ

Luca Cavallaro
Politecnico di Milano
Piazza L. da Vinci, 32 – 20133 Milano, Italy
cavallaro@elet.polimi.it

Mattia Monga
Università degli Studi di Milano
Via Comelico 39 – 20135 Milano, Italy
mattia.monga@unimi.it

ABSTRACT

Aspect-oriented programming (AOP) fosters the coding of tangled concerns in separated units that are then woven together in the executable system. Unfortunately, the oblivious nature of the weaving process makes difficult to figure out the augmented system behavior. It is difficult, for example, to understand the effect of a change just by reading the source code. In this paper, we focus on detecting the run time impact of the editing actions on a given set of test cases. Our approach considers two versions of an AspectJ program and a test case. Our tool, implemented on top of the *abc* weaver and the AJANA framework is able to map semantics changes to the atomic editing changes in the source code.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Language, Verification

Keywords

AspectJ, Software Maintenance, Change Impact Analysis

1. INTRODUCTION

Software maintenance of an evolving code base is a complex problem. A major source of complexity is understanding the effect of source code changes on the behavior of the program. Even small changes can have non-local effects that can make difficult to grasp the impact of the global properties of the system. In object-oriented programs polymorphism and dynamic binding may affect the behavior of virtual method calls that are not lexically near the allocation site. The problem is even critical in aspect-oriented software, due to the intrinsic obliviousness [4] of join points. In fact, the woven aspect may change control and data dependencies of the base code, and a change in the aspect code can significantly affect the semantics of the whole system. This is actually the very motivation of introducing an aspect: in practice, however, especially when both aspects and classes evolved separately, it is very easy to get unexpected results. We aim at supporting the programmer in tracking down the causal chain from his/her changes to the surprising effect, in order to ease the conception of a solution. An important contribution to this problem is offered by *change impact analysis* [2], a collection of techniques for determining the effects of a source code editing action on the behavior of a set of test cases for a program. Recently, many techniques have been proposed to support change impact analysis of object-oriented software [9, 11] but very little effort has been done to apply this technique to Aspect Oriented Software

In this paper we present an application of change impact analysis to AspectJ programs. Our approach considers two versions of an AspectJ [6] program and captures their syntactical differences, breaking them in atomic changes. Then we observe the behavior of the program under a test case that gives unexpected results. We build a representation of the executed parts of the two versions and we compare them, in order to understand how the semantics has changed and how this maps to the atomic changes introduced. We implemented our approach on the top of *abc* [1], an extensible aspect compiler, and AJANA [16], a framework for AspectJ analysis.

2. MOTIVATING EXAMPLE

We illustrate the need for change impact analysis with an example for which we report the source code in Listings 1 2, and 3: it implements a simple system composed of two Java classes and one AspectJ aspect. The class `Point` (see Listing 1) has two fields, and the proper getters and setters for those fields; moreover it exposes a `setRectangular` method, which sets both fields. The class `PointExt` (see Listing 2) should be considered an evolution of `Point`, in fact it is a subclass of it that overrides two methods.

Listing 3 shows an aspect that is expected to be woven to the above base system. This aspect declares some "introductions" to the base system, two pointcuts and six pieces of advice. The evolution of the aspect consists in adding a new field and modifying the advice marked as `before2`.

A test case for the system is listed in Listing‘4.

The code reported in the example implements an observer pattern by using the aspect `BoundPoint`. The latter defines

FOAL'09, March 2, 2009, Charlottesville, Virginia, USA.

Listing 1: Source code for Point class

```
public class Point {
    // Y part omitted to save space
    int x = 0;
    public int getX() { return x; }
    public void setRectangular(int newX, int newY)
        throws Exception {
        setX(newX);
        setY(newY);
    }
    public void setX(int newX)
        throws Exception {
        if (newX < 0) throw new Exception();
        x = newX;
    }
}
```

Listing 2: Source code for PointExt class

```
public class PointExt extends Point {
    //Overridden in modified version
    public void setRectangular(int newX, int newY)
        throws Exception {
         setX(newX + 1);
         setY(newY + 1);
    }
     //Overridden in modified version
    public void setX(int newX) throws Exception {
        if (newX < 0) throw new Exception();
        x = (int)newX /2;
    }
}
```

some pieces of advice woven into join points identified by the calls to the method setX. These pieces of advice check the parameter passed to the called method and keep a history of the older values of the x field. We imagine to slightly modify the initial version of the program by overriding two methods in the class *PointExt*, which is a subclass of *Point*; moreover, we add a field in the aspect BoundPointand a line in the before2 advice.

From the viewpoint of editing changes, the overriding of the methods in *PointExt* can be decomposed in two actions: a first step that brings a new method in the subclass, then a second step that changes the method body of the added method. Finally we also consider that the overriding brings a change in the lookup table of the program: before the change when a method *setX* was invoked on an object with dynamic type *PointExt* the superclass method was invoked. After the overriding the method that is actually invoked in this case is the overridden one.

Finally in listings 4 we show a test case for the example program. Of course the outcome of the test for the first and for the modified version of the program will be different, but not all the introduced changes could be responsible for the result change.

The main purpose of our analysis is to help a developer understand which code edits originated the change and which actions he should take to bring the program back into a state in which it gave the previous output.

3. CHANGE IMPACT ANALYSIS FOR AOP

Our approach considers two versions v_0 and v_1 of the same program and a set of test cases that should apply on both. We aim at mapping source code changes to the semantic differences induced by a test case t. In order to achieve this goal, we proceed as follows:

Listing 3: Source code for BoundPoint aspect

```
public aspect BoundPoint {
    //Added in modified version
    private int previousValue;

    // a reference to a Point object
    PropertyChangeSupport support =
        new PropertyChangeSupport(this);
    public void addPropertyChangeListener
        (PropertyChangeListener l){
        support.addPropertyChangeListener(l);
    }
    void firePropertyChange(Point p,
                            String property,
                            double oldval,
                            double newval) {
        p.support.firePropertyChange(property,
            new Double(oldval),new Double(newval));
    }

    // ====== pointcuts =======
    pointcut setterX(Point p):
    call(public void Point+.setX(*)) && target(p);

    pointcut setterXonly(Point p):
    setterX(p) &&
    !cflow(
     execution(void Point+.setRectangular(int,int)));
    // ====== advices ======
    before(Point p, int x) throws InvalidException:
     setterX(p) && args(x) { // before1
        if (x < 0) throw new InvalidException("bad");
    }
    void around(Point p): setterX(p) { // around1
      int oldX = p.getX(); proceed(p);
      firePropertyChange(p,"setX",oldX,p.getX());
    }
    void around(Point p): setterXonly(p) { // around2
      int oldX = p.getX(); proceed(p);
      firePropertyChange(p,"onlysetX",oldX,p.getX());
    }
    before (Point p): setterX(p){ // before2
      //added in modified version
      this.previousValue = p.getX();
      System.out.println("start setting p.x");
    }
    after(Point p) throwing (Exception ex):
     setterX(p) { // afterThrowing1
      System.out.println(ex);
    }
    after(Point p): setterX(p){ // after1
      System.out.println("done setting p.x");
    }

}
```

1. we compare the source code of v_0 and v_1 and find the textual changes between the two. The difference is decomposed into *atomic changes* detailed in Section 3.1;
2. we build a control flow representation ($AJIG$) for each version of the program and we mark the differences we found between $AJIG_0$ and $AJIG_1$ as *dangerous edges* (Section 3.2);
3. the two programs are instrumented and run whit the test set, in order to collect dynamic pieces of information that we use to decorate $AJIG_0$ and $AJIG_1$ by marking the executed paths (Section 3.3);
4. for each test case t that gives different results when applied to v_0 and v_1 we consider the decorated graphs $AJIG_0$ and $AJIG_1$ and we finally map dangerous edges to source code changes.

The idea is that only the statements on a dangerous path solicited by the test case t can be responsible for the change

Listing 4: Source code for a test case for program in listings 1 2 3

```
public class Demo implements
                    PropertyChangeListener {
    public void propertyChange
        (PropertyChangeEvent arg0) { /* ... */ }
    public static void main(String[] a)
        throws Exception {
        Point p1 = new Point();
        p1.addPropertyChangeListener(new Demo());
        p1.setRectangular(5,2);
        if(p1.x > 5) { p1.setX(6);}
        Point p2 = new PointExt();
        p2.addPropertyChangeListener(new Demo());
        p2.setRectangular(5,2);
        p2.setX(5);
    }
}
```

of results of t: we are thus interested in them when we are trying to understand the impact of our source code editing.

3.1 Atomic changes

The first step in change impact analysis is to decompose the difference between v_0 and v_1 into a set of *atomic changes.* We consider the atomic changes as proposed by [10, 8] for the base system, and, by [18] for the aspect part. A set of atomic changes that is able to reproduce the difference between v_0 and v_1 is computed by comparing the abstract syntax trees of the two versions of the program and by finding their differences. Since an AspectJ program is composed by a plain Java code base and some aspect-oriented code units, one can apply the different catalogs of atomic changes separately.

In general, a given change may depend on some previous one. Intuitively, an atomic change A_1 is dependent on another atomic change A_2 if applying A_1 to the original version of the program without also applying A_2 results in a syntactically invalid program: in order to be possible to change a method (CM), that method has to exist: if it was added by the new version (AM), the atomic change CM depends on the specific AM.

Three types of dependence can be considered: *structural,* which captures the necessary sequences that occur when new elements are added or deleted in a program; *declarative,* which captures all the necessary element declarations that are required to create a valid intermediate version; *mapping,* which captures implicit dependencies introduced by changing the class hierarchy or overriding methods. An example of dependencies between atomic changes is shown in Figure 1. Here we report the dependencies computed for the example of Section 2. In the picture the arrows denotes the interdependencies between changes. The overriding of the methods in the class PointExt is broken into four atomic changes: the addition of an empty method (AM) and the corresponding modification of the method body (CM). The latter is structurally dependent on the former. Moreover we also have two changes for the virtual methods lookup: let's consider for example the overriding of the method setX. Each time someone is going to call the method setX on a reference statically typed as Point, if the runtime object will be of type PointExt the call will reach the method defined in the subclass. The same will happen if the reference has as static type PointExt. These changes are represented as lookup changes (LC) and the addition of the method setX in PointExt has a mapping dependency on them. The changes in the aspect BoundPoint are decomposed in an added field (AF) and a changed advice body (CAB) changes. The latter is declaration dependent on the former, because it uses the field definition added by the AF change.

3.2 Control flow representation of the program

To capture the differences in behavior we need to build a representation of the program. To accurately model AspectJ semantics, we use a control-flow representation, the AspectJ Inter-module Graph (AJIG), presented in [15]. This representation is an extension to aspect-oriented encapsulation units of the Java Interclass Graph [5]. The main goal of this graph is to make explicit the interactions between the base system and the aspect part of an AspectJ program with particular regard to the weaving of multiple advice at the same join point.
The AJIG is designed to represent precisely all interactions involving the pieces of advice that are anonymous pieces of code analogous to object-oriented methods that are executed when a specific dynamic join point occurs: such interactions are at therefore the core of aspect-oriented programming. Unlike explicit method calls, an advice is invoked implicitly at the shadow of a certain join point. The execution of the advised Java code is completely replaced by the combination of pieces of advice and the join point shadow that matches it. Before- and after- advice can be considered as special cases of around-advices with an implicit **proceed** statement. Thus, for each advised piece of Java code its control-flow subgraph is replaced with the representation of woven pieces of advice called Interaction Graph (IG).

To create an interaction graph we need to compute the precedence the woven pieces of advice are executed with, and to do this the AJIG uses an *advice nesting tree*, which represents the run-time advice nesting relationships. Each tree level contains at most one around-advice, which is the root of all pieces of advice in the lower levels of the tree. With each around-advice A the algorithm associates (1) a possibly-empty set of before-pieces of advice and after-pieces of advice, (2) zero or one around-pieces of advice, and possibly (3) the actual call site that could be invoked by the call to **proceed** in A. These pieces of advice and the call site appear as if they were nested within A.

For example, a call to Point.setX gives the advice nesting tree shown in Figure 2. The information captured by the advice nesting tree is used to weave advice bodies at a shadow and can be exploited to build the AspectJ Interaction Graph. A condensed picture of the AJIG for Point.setX is shown in Figure 3.

The AJIGs built for the two versions of the program are then compared to find *dangerous edges,* i.e., edges that are different in the two versions. The algorithm for comparing AJIGs is a depth first traversal of the graphs that aims at finding the differences between paths in the graphs and it was proposed and described in detail in [15]. Basically, if an edge is present in the first graph and is missing in the second or if it was added in the second or if the edges of the path changed their label, then they are marked as "dangerous". An example of a dangerous path is shown in Figure 4. The example refers to the AJIG of Figure 3 and it corresponds to the impact of adding a field and modifying the body for advice before2 in BoundPoint aspect and overriding the method setX in the class PointExt. There are two dangerous edges (marked in red) in the figure. The first one is the one

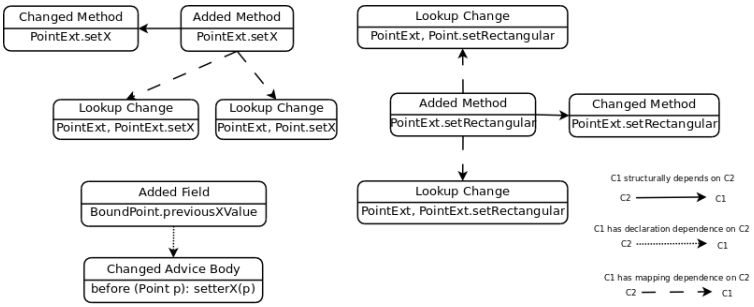


Figure 1: Representation of atomic changes dependencies for the example

Figure 2: Advice nesting tree for BoundPoint at shadow this.setX() in the example presented in section 3.1

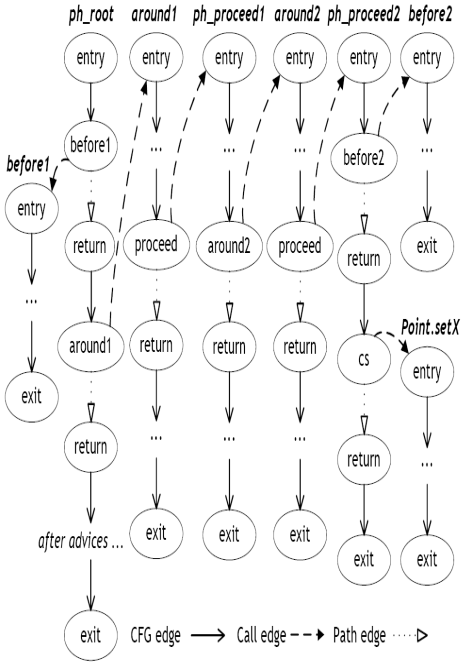


Figure 3: AspectJ Interaction Graph for the example presented in section 3.1

starting from the before2 entry point and determined by the addition of the statement **this**.previousValue = p.getX() in Listing 3. The second one is determined by the addition of the path going through the method PointExt.setX, overridden in class PointExt.

3.3 Combining dynamic information

Dangerous edges are only a potential cause of an unwanted effect. However, the execution of a test case that gives unexpected results can be *explained* in terms of dangerous path traversals. In other words, we use the information gathered from the instrumentation of the program to consider only dangerous paths executed by a test case and we map these paths on the atomic changes computed earlier. The mapping takes place by considering that nodes in the paths interested by dangerous edges represent instructions in the program, so we perform the mapping considering their line numbers in the compilation units they come from. At this point we have a set of atomic changes, representing syntactical changes in the source code of the program, mapped on a set of dangerous edges in the program control flow representation, which represents changes of behavior of a test case. This gives the causal chain between the editing and the unexpected result: the set of changes mapped onto the traversed dangerous edges are responsible for the observed behavior. In fact, undoing the atomic changes set will produce a version of the program that will not show anymore the altered test result. Moreover, by removing the set of changes, we would have a syntactically correct intermediate program version, since atomic changes interdependencies consider syntactical dependencies between changes. Atomic changes not mapped on dangerous edges for a given test are not responsible for the change of that test result and can be left in the modified version of the program. Finally we can consider that changes not mapped on any test case dangerous edges are not stressed at all by the test suite.

4. IMPLEMENTATION

For the implementation of our solution we relied on the *abc* compiler [1] and on AJANA framework [16].
The *abc* compiler is an extensible compiler for AspectJ based on Soot [12]. *abc* gave us the possibility to manipulate the abstract syntax trees of the programs we analyzed and to find atomic changes. Moreover it provided us with an easy to analyze intermediate representation of the program, Jimple, which we used to build our AJIG. Finally *abc* performs a two phases weaving process. In the first phase it computes the pointcuts shadows in the system code but keeps the weaving information separated from the system bytecode, then it performs some optimizations and finally produces code to insert pieces of advice at the proper shadow. The possibility to keep weaving information separated from the code gave us way to build our AJIG witout considering the extra code that AspectJ compiler generates to weave an advice at a shadow. In this way we could produce a more accurate representation of the program control flow.
AJANA is a general framework for ApectJ analysis. It provided us the AJIG representation. We had to slightly modify this representation to implement our change impact analysis, in order to keep track of some pieces of information originally not needed for AJIG construction.

5. RELATED WORK

Change impact analysis for object-oriented programs was introduced by Ryder et al. in [10]. In their paper they proposed a technique for Java software and in [9, 8] they

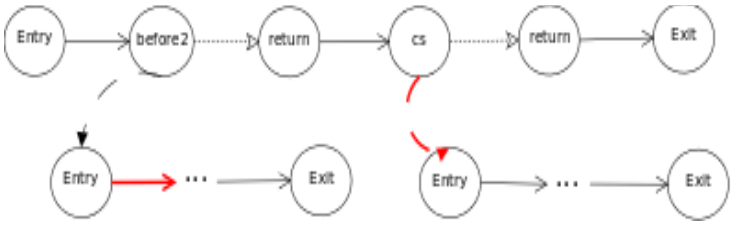


Figure 4: Dangerous edges determined by modification of before2 advice body in class BoundPoint and overriding of method setX in PointExt class

extended the applicability of the technique and provided tool support.

The problem of change impact analysis for AspectJ software was previously treated in [18]. This paper presents a lightweight approach to change impact analysis based on a static program representation. We based our approach on a dynamic representation of the program which captures the actually executed paths in the control flow. Since from our previous work [3] we noticed that for AspectJ static representations introduces a considerable quantity of dependencies that are not to be considered at run time we think that our approach could lead to an increased precision in the analysis.

Change impact analysis is related with regression testing and debugging. We based our work on a framework developed for a regression test selection technique [15]. Many other techniques were proposed for aspect oriented programs regression testing. In [13] an approach based on a wrapper class synthesis technique and a framework for generating test inputs for AspectJ programs are presented; in [14] the authors propose an approach to generate regression test cases starting from a specification of the program. All these approaches can be considered as complementary to ours, since they focus on discovering if a failure was introduced by changes while we want to unwind the relation between source code changes and failures found in the program.

Delta debugging was originally introduced by Zeller in [17]. This technique allows to create intermediate versions of the program by adding or removing a set of atomic changes from the program source. Change impact analysis for Java programs was successfully integrated with delta debugging in [9], we plan to do the same in our future work.

6. CONCLUSION AND FUTURE WORK

In this work we developed a prototype to perform change impact analysis on AspectJ programs. Our prototype can relate changes in the source code of an AspectJ program to changes in its behavior. The tool is based on a modified version of the *abc* aspect weaver and the AJANA framework. We tested our prototype on simple examples of evolving AspectJ programs that advise call and execution join points. Since our first results are promising, we plan to extend our experimentation to real world programs. Moreover, we think that the level of abstraction of atomic changes is too low in most cases, even if it worked well in our toy examples. In fact, Ryder et al. report in [11] that Java change impact analysis based only on a syntactical classification produces changes sets that are difficult to be interpreted by humans. Thus, we plan to study some higher level view to give to the programmer a more effective means of understanding his or her changes.

7. REFERENCES

[1] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *Proc. of AOSD'05*.

[2] S. A. Bohner. Software change impact analysis. *Wiley-IEEE Computer Society Pr*, 1996.

[3] A. C. D'Ursi, L. Cavallaro, and M. Monga. On bytecode slicing and AspectJ interferences. In *Proc. of FOAL'07*.

[4] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, RIACS, 2000.

[5] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proc. of OOPSLA'01*, 2001.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. of ECOOP'01*, 2001.

[7] M. Marin, L. Moonen, and A. van Deursen. An integrated crosscutting concern migration strategy and its application to JHotDraw. In *Proc. of SCAM'07*, 2007.

[8] X. Ren, O. Chesley, and B. G. Ryder. Identifying failure causes in Java programs: An application of change impact analysis. *IEEE TSE.*, 32(9), 2006.

[9] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of Java programs. In *Proc. of OOPSLA'04*, 2004.

[10] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proc. of PASTE'01*, 2001.

[11] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in Java programs using change classification. In *Proc. of FSE-14*. ACM, 2006.

[12] R. Vallée-Rai, L. Hendrena, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot – a Java optimization framework. In *Proc. of CASCON'99*, 1999.

[13] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of aspectj programs. In *Proc. of AOSD'06*. ACM, 2006.

[14] D. Xu and W. Xu. State-based incremental testing of aspect-oriented programs. In *Proc. of AOSD'06*, 2006.

[15] G. Xu and A. Rountev. Regression test selection for AspectJ software. In *Proc. of ICSE'07*, 2007.

[16] G. Xu and A. Rountev. AJANA: a general framework for source-code-level interprocedural dataflow analysis of AspectJ software. In *Proc. of AOSD'08*, 2008.

[17] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Proc. of ESEC'99*, 1999.

[18] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Change impact analysis for AspectJ programs. In *Proc. of ICSM 2008*.

Enhancing Base-code Protection in Aspect-Oriented Programs

Mohamed ElBendary
University of Wisconsin-Milwaukee
Milwaukee, WI 53211
mbendary@cs.uwm.edu

John Boyland
University of Wisconsin-Milwaukee
Milwaukee, WI 53211
boyland@cs.uwm.edu

ABSTRACT

Aspect-oriented programming (AOP) promises to localize concerns that inherently crosscut the primary structural decomposition of a software system. Localization of concerns is critical to parallel development, maintainability, modular reasoning, and program understanding. However, AOP as it stands today causes problems in exactly these areas, defeating its purpose and impeding its adoption. First, the need to open up systems' modules for aspects' interaction competes with the need to protect those modules against possible fault injection by aspects. Second, since aspects are written in terms of base code interfaces, base system components must be stable before aspect components can be developed. This dependency hinders parallel development. This work proposes a language-based solution that allows base code classes to regulate aspect invasiveness, and provides loose coupling of aspects and base code.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Aspect-Oriented Programming; D.2 [**Software Engineering**]: Compilers

General Terms

Algorithms, Design

1. INTRODUCTION

Aspect-oriented software development (AOSD) is supposed to apply over a system's entire lifetime, positively impacting software measures such as cost, quality, and time-to-market [8]. AOP promises to localize cross-cutting concerns by providing language-based mechanisms for explicitly representing their structure and/or behavior. AOP does provide a cleaner separation of concerns. However, AOP negatively impacts modularity by crossing module boundaries in a completely unregulated fashion [10].

This work is an attempt to resolve two points of contention that are impeding the adoption of AOP. The first point is the competition between the need to open up systems' modules for AOP and the need to protect those modules against possible fault injection by AOP. The second point is the need to have base system components stabilized before aspect components can be developed, which reduces opportunities for parallel development.

We believe that pure obliviousness (currently, the dominant approach to AOP, as in AspectJ) is problematic for the following reasons. First, while clients (including aspects) may be insensitive to changes in implementation details of the components they use, they are tightly coupled to their interfaces. For example, an aspect that references method `m` in class `C` breaks if method `m` is now called `n`, this problem is referred to as the *fragile pointcut problem*. Second, pure obliviousness offers no information on the base side regarding what elements of an interface are being advised or which aspects are involved. Aspects can infuse the component's internals through introductions and advice mechanisms making it impossible to reason about a base component by examining it in isolation. Third, pure obliviousness renders the base code component entirely helpless in the face of *harmful aspects*. A harmful aspect is an aspect that violates a base code policy as it extends (advises) base components, for example, by replacing the body to be executed at a joinpoint with something entirely different.

Our philosophy is that since the base code and the aspect code participate in making up a module's interface, they should explicitly cooperate to preserve the module's boundary. We see a module boundary extending beyond traditional class or aspect module boundaries with base code classes being responsible for establishing their module boundaries within the system, using advising constraints to limit aspects' invasiveness.

We believe that pure obliviousness can be sacrificed to maintain ease of reasoning, ease of maintenance, separation of concerns, and code locality. This work focuses on separation of concerns, code quality, and ease of maintenance as primary "concerns".

2. INTERFACE IMAGE (I2) APPROACH

An Interface Image (I2) is a level of indirection through which all advising requests are carried out. It provides a mechanism by which a class exposes a set of joinpoints through aliasing base code interface elements. The image incorporates advising constraints per exposed joinpoint. Aspects are developed against the aliases defined in the interface images of base code classes. Aspects are not allowed to advise classes directly. This indirection limits the scope of

FOAL'09, March 2, 2009, Charlottesville, Virginia, USA.

image_declaration ::= `image {`
`[opento: {` *TypeAccess** `};]`
`[`*alias definition**`]`
`}`
alias_definition ::=
method-header `=` *method-header* `{` *constraints* `}`
| `* = * {` *constraints* `}`

method-header ::=
`[`*modifiers*`]` *RT method-name*`(`*P*`)` *throws_list*

modifiers: *Java-style member method modifiers*
RT ::= *TypeAccess*
method-name: *Java-style method identifier*
P: *Java-style method parameter list*
throws_list ::= *TypeAccess**
TypeAccess ::=
Java-style type access (qualified and simple type names)

constraints ::= `kind: { `*Advice_Kind** `};`
| `(origin=`*Origin*`, boundary=`*Boundary*`);`
| `exceptions: { `*Exception_Type** `};`

Advice_Kind ::= `before` | `after` | `after_returning`
| `after_throwing` | `around`
Origin ::= `internal` | `external`
Boundary ::= `method` | `class` | `package`
Exception_Type ::= *TypeAccess*

Figure 1: Interface image syntax.

dependency of aspects on base code to that of images only. I2 lends itself to a feature-obliviousness design, as the next section will show. Our design requires cooperation from the base code developers so it is not language-level oblivious. I2 is not designer-oblivious either since it assumes designers are aware of aspects realizing functionality.

In this design, an I2 provides the following benefits:

1. The base code is now an active participant in the advising process since it is up to each class to expose joinpoints on which it permits advice. For each exposed joinpoint, advising constraints can be attached to disallow unwanted aspect advising.

2. Response to changes in the interface of a class is limited to updates in the class' interface image. Aspects are not involved. Parallel development can benefit from this loose coupling.

3. The I2 serves as a specification of advisable interface elements for base code and aspect developers alike.

This work studies the interface image approach in the context of classes only. We leave augmenting interfaces and aspects with interface images for future work.

Interface images are defined using the `image` construct. An `image` can only appear within the scope of a class definition. Figure 1 shows the syntax and Fig. 2 shows an example instance. An empty image `image{}` exposes the class to unrestricted (AspectJ-style) advising.

```
class Point extends Shape {
  protected int x, y;
  public void moveby(int dx, int dy){
        x += dx; y += dy;
  }

  image {
    opento: {CheckScene};
    public void moveby(int dx, int dy) =
    public void translate(int dx, int dy){
     kind: {after};
     (origin=external, boundary=class);
     exceptions: {SceneInvariantViolation};
    }
  }
}
```

Figure 2: Example `image` for a class `Point`

2.1 The `opento` clause

The `opento` clause allows a class to provide a list of aspects allowed to perform introductions on it. If an image does not declare an `opento` list, then the enclosing class will accept introductions from any aspect. An empty `opento` list prohibits any aspects from performing introductions on the declaring class.

2.2 Aliases

An alias definition has a signature on the left-hand side of an equal sign followed by an alias signature and an attached scope for declaring advising constraints. The aliases are used to name aspect joinpoints—only aliased methods can be advised.

A class can only alias methods that it explicitly declares. Both instance and static methods are aliasable. The wildcard form `* = *` permits all declared methods in the class to be advised under a single set of advising constraints.

An image can also declare multiple aliases for the same joinpoint to further allow constraint refinement per joinpoint. If two aspects implementing two different concerns each with a different set of, possibly conflicting, advising constraints at the same joinpoint, accommodating both is easily done by defining different aliases on the same joinpoint and having each aspect use a different alias definition. Providing different hooks (aliases) with different advising constraints essentially allows joinpoints to "fan-out" different channels for aspects to communicate with the base code.

2.3 Advising Constraints

The `kind` clause lists the advice kinds allowed at this joinpoint. For example (Fig. 2), the clause `kind: {after}`, would only allow `after` advice at this joinpoint for this alias. If an aspect declares a pointcut that matches this alias, and declares an advice of a kind other than this kind (e.g. `around`), this advice application will be disabled. This is useful for enforcing the design intent that the translation cannot be skipped.

An empty `kind` clause turns off *any* advising on this joinpoint through this alias. If a `kind` clause is omitted, all advice kinds are allowed.

The `(origin, boundary)` pair, if it exists, specifies whether advising is permitted for calls originating inside (`internal`) or outside (`external`) module boundary or both. A module boundary is either `method`, `class`, or `package`. If the `(origin, boundary)` pair is omitted, all calls may be advised.

The `exceptions` clause, if it exists, lists all exception types that if thrown by this joinpoint cannot be "softened" by an aspect with a matching pointcut of a `declare soft()` statement. Omitting the `exceptions` clause allows all exceptions to be softened. An empty exceptions list prevents softening of any exception through this alias.

2.4 Summary

The interface image technique has the following benefits to the base code designer: The base code can limit advice to join points that are semantically relevant to the outside, and can limit which aspects are permitted to perform introductions. The base code can determine which exceptions can be safely softened. No extra aspect is required to check advising constraints.

On the other hand, the aspect designer can still make use of the full power of aspect orientation and is now only dependent on the alias names, not the actual method names. This avoids the overloading of method signatures with two different meanings, from the client's perspective they are service access points while from the aspect's perspective they are joinpoints.

3. EXAMPLE

This banking authorization example is adapted from Laddad [9]. Ladded developed it to showcase modularity of an AspectJ solution over a conventional Java solution. The example is an authorization service in a banking system. The base code (not shown) for this example consists of methods for performing simple operations on bank accounts: debit, credit and transfer.

Laddad used an abstract aspect, shown in Fig. 3 to implement an authorization protocol. The abstract pointcut `authOperations()` acts as a hook for concrete derived aspects to quantify which operations in the system they want to apply the authorization protocol to. A derived concrete aspect, `BankingAuthAspect`, that fully implements the authorization concern is shown in Fig. 3.

The first `before()` advice in Fig. 3 performs authentication if the subject accessing the system has not been authenticated yet. The `around()` advice wraps authorization around the banking operations' calls. The example uses banking methods' names as the permission names. We rely on the `JoinPoint.StaticPart` parameter to access method names at the joinpoints in the body of `getPermission()`.

The implementation given uses strictly two kinds of advice, `before` and `around`. So it is safer to allow exactly those kinds of advice and explicitly disable all others, possibly allowing more as new concerns are added. It is important to note that allowing `around` does not automatically include `before` and `after` kinds, even though their effects may be possible.

The solution provided by Laddad [9] does not soften the checked exception, `InsufficientBalanceException`, used by the application to prevent invalid withdrawals and/or transfers. The solution provides an aspect implemented specifically for preserving this exception. We believe that,

```
public abstract aspect AbstractAuthAspect{
  public abstract pointcut authOperations();

  before() : authOperations() {
   // authentication logic
  }

  public abstract Permission getPermission(
   JoinPoint.StaticPart joinPointStaticPart);

  Object around()
    : authOperations() &&
      !cflowbelow(authOperations()) {
    // Perform authorized operation
  }

  before() : authOperations(){
    // Authorization logic
  }

}  // Abstract aspect ends here
```

Figure 3: AbstractBankingAuthAspect [Laddad].

```
public aspect BankingAuthAspect
        extends AbstractAuthAspect{
  public pointcut authOperations()
    : execution(public * banking.Account.*(..))
   || execution(
        public *
    banking.InterAccountTransferSystem.*(..));

  public Permission getPermission(
    joinPoint.StaticPart joinPointStaticPart){
      return new BankingPermission(
        joinPointStaticPart.getSignature().
                getName());
    }
}
```

Figure 4: Concrete aspect BankingAuthAspect [Laddad].

```
  image {
    opento: {};
    * = * {
      kind: { before, around };
      exceptions: { InsufficientBalanceException };
      (origin=internal, boundary=package);
    }
  }
```

Figure 5: Banking example interface image declaration.

for an exception that is part of the contract of a core class method, the decision of whether it is softened or not, is for the core class to make. Clients on the base code may want to "know" about this situation and handle it in their own specific ways. The `exceptions` clause also saves substantial coding, since it replaces an entire aspect that had to be developed in the classical AspectJ solution.

Given the nature of this application and its operations, calls to `credit` and `debit` originating outside `banking` should be considered dubious. Currently, AspectJ guards against this using pointcut matching. However, a single misplaced wild card, could jeopardize the integrity of the application and the base code is simply helpless. Instead, base code classes could easily add the `(origin, boundary)` pair in Fig. 5, improving the robustness of advising and preventing the potentially erroneous behavior of *unintended* wild-cards.

The image in Fig. 5 is the finished product, this is all the code we need in order to alleviate the potential problems outlined above. In this example, the aspect side does not perform introductions of any sort. However, it does not make sense to keep the base classes open for intertype declarations, even though no errors will be caused in this case. This is because it is safer to progressively open classes to specific aspects as needed than leaving them open for all aspects and deal with possible maintenance problems later.

4. IMPLEMENTATION

I2 is implemented as an extension to AspectJ within the *aspectbench compiler* (abc) [2]. Our implementation uses the `JastAdd` [4] front-end of abc.

4.1 Image Semantic Checking

An image has to pass an error checking phase before it can be translated, including that the enclosing class does not have multiple image declarations, and that aliases are for methods declared in the enclosing class. We also check for duplicate constraint declarations.

If an aspect wishes to make introductions, the effected classes must have a `image` declaration and permit the introduction (see `opento`). Then for each joinpoint that the advice potentially applies to, we check the advice for violations of `kind` and `exceptions` clauses.

4.2 Image Translation

The image construct is translated internally into a privileged static nested aspect that performs method introductions into the class declaring the image. Being privileged allows the translated aspect to refer to private members of enclosing classes, which may be aliased.

One introduced wrapper method that called the original method is generated for each alias definition. An `around` advice is used to intercept calls to the original method and direct them to the wrapper. This advice in controlled by the `(origin, boundary)` pair constraint for the alias. Table 1 shows the translation to AspectJ conditions.

Additionally, we add `&& !within(imageAspect)` to each generated pointcut to prevent internal aspects from advising themselves, where "`imageAspect`" is some identifier used only internally that identifies the generated aspect.

We use AspectJ's precedence declaration to ensure that the generated aspects are applied before the concern-specific aspects:

Table 1: Translation of *(origin, boundary)* pairs to pointcuts.

(origin,boundary)	Pointcut Translation
(internal, method)	within(*signature(m)*)
(internal, class)	within(*enclosing type*)
(internal, package)	within(*enclosing package*)
(external, method)	!within(*signature(m)*)
(external, class)	!within(*enclosing type*)
(external, package)	!within(*enclosing package*)

```
class Point extends Shape {
  protected int x, y;
  public void moveby(int dx, int dy){
        x += dx; y += dy;
  }

  privileged static imageAspect {
   public void Point.translate(int dx, int dy){
     moveby(dx, dy);
   }
   void around(Point p):
    target(p) && !within(imageAspect) &&
    !within(Point) &&
    call(
      public void Point.moveby(int dx,int dy)){
        p.translate(dx, dy);
   }
  }
}
```

Figure 6: Class Point after translation

```
public aspect _internalOrderingAspect {
  declare precedence: *..*imageAspect*, *;
}
```

Any concern-specific precedence declarations are rewritten to add the pattern `*..*imageAspect*` to the front of the precedence list of each one of them.

Class `Point` shown in Fig. 2 translates internally to the one shown in Fig. 6. It shows the call to the aliased method `moveby()` wrapped inside introduced method `translate()`. It also shows the `around` advice and the pointcuts generated from the `(origin, boundary)` constraint. The `target(p)` pointcut exposes `p` for use in the advice body.

5. EVALUATION

This section presents evaluation of the I2 approach. This quantitative evaluation uses the AspectJ Development Tools' (AJDT) cross-cutting map generator to measure coupling. In the context of this study, coupling means the existence of a "cross-cutting relationship" between two components. A cross-cutting relationship in the source code arises from intertype declarations and advice declarations.

This study uses two examples from the AspectJ programming guide published by the Eclipse foundation [6]. These are, the `Observer` and the `Telecom Simulation`. The third is the `ants simulation` program used to demonstrate Open

Table 2: Coupling data reported by AJDT.

Example	AspectJ	AJI2	Coupling Change(%)
Observer	38	48	+26.3%
Telecom	30	24	-20.0%
Ants	670	626	-6.6%

Modules [1]. For each example, two implementations are analyzed, a classical AspectJ implementation and an AspectJ with I2-style implementation.

In order to allow AJDT to process I2 sources, we simulate the effects of I2 syntax by modifying the implementations of the examples as follows. We use a field introduction of an aspect instance into the class that uses `opento`, to account for classes referring back to aspects using the `opento` clause. We use one introduction declaration within each aspect referenced in the `opento` list. We also change method names in pointcuts to use alias names instead of the original names of their signatures in classes. For every alias that we use, we write a method in base code classes selected by the pointcut using the alias. This additional method bears the alias signature and wraps the call to the original aliased method. This arrangement simulates limiting the scope of aspects' references to images as opposed to the entire program as in AspectJ. The table above summarizes the "cross-cutting relationships" data reported by AJDT for each example.

In weighing these percentages a few details need to be taken into consideration. In `ants simulation`, the 6.6% corresponds to 44 less cross-cutting relationships in the program, which in turn corresponds to 27 spots in the source code where independent evolution is possible. We can only expect the number of spots to grow for larger programs with a similar feature mix. The 27 spots represent a 9% reduction in coupled spots in the code, a substantial percentage in a large program.

6. RELATED WORK

In Open Modules (OM) [1], a module exposes pointcuts that can be advised by external aspects as part of the module's interface. Thus only "external origin" calls are advisable. Also unlike I2, Open Modules do not allow advice to crosscut modules unless the modules are related through inheritance.

Cross-cutting Pointcut Interfaces (XPI) [5] separate a traditional aspect into three aspects: one to specify pointcuts and advising constraints; another aspect with advice implementations; and a third aspect to check advising constraints. Roughly, I2 can be seen as a way to automatically create and check the advising constraints of XPI.

Using Explicit Join Points (EJPs [7]), the base code adds syntactic hooks that look like static method calls at points where advising is required. This is more powerful than AspectJ or I2 because these hooks can occur in arbitrary blocks of code, at the cost of requiring the base code designer to put forward more effort.

Modular Aspects with Ownership (MAO [3]) enables modular reasoning using ownership to constrain heap effects. Advice can be declared as having no or only limited control effects, or no or limited data (heap) effects. Unrestricted aspects must be "accepted" by the base code to which they apply, in a similar mechanism to I2's "opento" declaration, which only applies to aspects with introductions. I2 preserves more feature obliviousness while MAO enables stronger modular reasoning.

Ptolemy [11] solves the fragile pointcut problem by introducing named event (joinpoint in AspectJ terminology) types that are declared independently from the modules (aspects) that announce/handle them. This is arguably superior to the pattern-matching used by AspectJ (and I2) to identify joinpoints. Ptolemy does not appear to support advising constraints.

7. CONCLUSION

This work addresses two modularity problems in AOSD: (1) aspect brittleness and sensitivity to base code changes; (2) the inability for base code to control advising. We provide a language-level solution to both problems in the form of a new construct added to classes that exports a view of the advisable class interface for aspects. Advising constraints can be attached to joinpoints that restrict what advice may apply. A prototype implementation as an AspectJ extension along with evaluation studies show it is possible to realize a design that loosely couples the evolution of the base code interfaces from the aspect-oriented code advising base components. We hope that these ideas can encourage greater adoption of AOP by the software engineering community.

8. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP '05*, pages 144–168, 2005.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *AOSD 2005*, pages 87–98, 2005.

[3] C. Clifton, G. T. Leavens, and J. Noble. Mao: Ownership and effects for more effective reasoning about aspects. In *ECOOP '07*, pages 451–475, 2007.

[4] T. Ekman and G. Hedin. The jastadd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.

[5] W. G. Griswold, K. Sullivan, Y. Song, Y. Cai, M. Shonle, N. Tewari, and R. Hridesh. Modular software design with crosscutting interfaces. *IEEE Softw.*, pages 51–60, 2006.

[6] A. P. Guide. The AspectJ programming guide. http://www.eclipse.org/aspectj/doc/released/progguide/.

[7] K. Hoffman and P. Eugster. Bridging java and AspectJ through explicit join points. Technical Report ejp-200705-1, Purdue University, 2007.

[8] I. Jacobson. A case for aspects. *Software development Magazine*, October 2003.

[9] R. Laddad. *AspectJ IN ACTION, Practical Aspect-Oriented Programming.* Manning Publications Co., 2003. ISBN 1-930110-93-6.

[10] G. T. Leavens and C. Clifton. Multiple concerns in aspect-oriented language design: A language engineering approach to balancing benefits, with examples. Technical Report TR 07-01a, Iowa State University, 2007.

[11] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP '08*, pages 155–179, 2008.

A Machine-Checked Model of Safe Composition*

Benjamin Delaware, William Cook, Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712 U.S.A.
{bendy,wcook,batory}@cs.utexas.edu

ABSTRACT

Programs of a software product line can be synthesized by composing *features* which implement some unit of program functionality. In most product lines, only some combination of features are meaningful; *feature models* express the high-level domain constraints that govern feature compatibility. Product line developers also face the problem of *safe composition* — whether every product allowed by a feature model is type-safe when compiled and run. To study the problem of safe composition, we present Lightweight Feature Java (LFJ), an extension of Lightweight Java with support for features. We define a constraint-based type system for LFJ and prove its soundness using a full formalization of LFJ in Coq. In LFJ, soundness means that any composition of features that satisfies the typing constraints will generate a well-formed LJ program. If the constraints of a feature model imply these typing constraints then all programs allowed by the feature model are type-safe.

Categories and Subject Descriptors

F.3.3 [**Studies of Program Constructs**]: Type structure

General Terms

Design, Languages

Keywords

Product lines, Type safety, Feature model

1. INTRODUCTION

Programs are typically developed over time by the accumulation of new features. However, many programs break away from this linear view of software development: removing a feature from a program when it is no longer useful, for example. It is also common to create and maintain multiple versions of a product with different sets of features. The result is a *product line*, a family of related products.

The inclusion, exclusion, and composition of features in a product line is easier if each feature is defined as a modular unit. A given feature may involve configuration settings, user interface changes, and control logic. As such, features typically cut across the normal class boundaries of programs. Modularizing a program into features, or *feature modularity*, is quite difficult as a result.

There are many systems for feature modularity based on Java, such as the AHEAD tool suite [4]. In these systems, a feature is a collection of Java class definitions and *refinements*. A class refinement is a modification to an existing class, adding new fields, new methods, and wrapping existing methods. When a feature is applied to a program, it introduces new classes to the program and its refinements are applied to the existing classes.

Figure 1 is a simple example of a product line containing two features, Bank and Sync. The Bank feature in Figure 1a implements an elementary Account class with setBalance and update methods. Feature Sync in Figure 1b implements a synchronization feature so that accounts can be used in a multi-threaded environment. Sync has a refinement of class Account that modifies update to use a lock, which is intro-

```
feature Bank {
  class Account
      extends Object{
    int balance = 0;
    void update(int x) {
      int newBal =
          balance + x;
      balance = newBal;
  }}}
```

(a) Bank Feature

```
feature Sync {
  refines class Account
      extends Object{
    static Lock lock
        = new Lock();
    refines void update(int x) {
      lock.lock();
      Super.update(x);
      lock.unlock();
  }}}
```

(b) Synchronized Feature

```
class Account extends Object {
  int balance = 0;
  static Lock lock = new Lock();
  void update(int x) {
    lock.lock();
    int newBal = balance + x;
    balance = newBal;
    lock.unlock(); }}}
```

(c) A composed program: Sync•Bank

1: Account with synchronization feature

*This material is based upon work supported by the National Science Foundation under Grant CCF-0724979.

duced as a static variable. Method refinement is accomplished by inheritance; Super.update(x) indicates a substitution of the prior definition of method update(x). Composing the refinement of Figure 1b with the class of Figure 1a produces a class that is equivalent to that in Figure 1c. The Bank feature can also be used on its own. While this example is simple, it exemplifies a feature-oriented approach to program synthesis: adding a feature means adding new members to existing classes and modifying existing methods. The following section presents a more complex example and more details on feature composition.

Not all features are compatible, and there may be complex dependencies among features. A *feature model* defines the legal combinations of features in a product line. A feature model can also represent user-level domain constraints that define which combinations of features are useful.

In addition to domain constraints, there are low-level implementation constraints that must also be satisfied. For example, a feature can reference a class, variable, or method that is defined in another feature. *Safe composition* guarantees that a program synthesized from a composition of features is type-safe. While it is possible to check individual programs by building them and then compiling them, this is impractical. In a product line, there can be thousands of programs; it is more desirable to ensure that all legal programs are type-safe without synthesizing the entire product line. This requires a novel approach to type checking.

We formalize feature-based product lines using an object-oriented kernel language extended with features, called *Lightweight Feature Java* (LFJ). LFJ is based on Lightweight Java [11], a subset of Java that includes a formalization in the Coq proof assistant [6], using the Ott tool [10]. A program in LFJ is a set of features containing classes and class refinements. Multiple products can be constructed by selecting and composing appropriate features according to a *product specification* - a composition of features.

Features modules are separated by implicit interfaces that govern their composition. One solution to type checking these modules is to require explicit feature interfaces. We instead infer the necessary feature interfaces from the constraints generated by a constraint-based type system for LFJ. The type system and its safety are formalized in Coq. We then show how to relate the constraints produced by the type system to the constraints imposed by a feature model, using a reduction to propositional logic. This reduction allows us to statically verify that a feature model will only allow type-safe programs without having to generate and checking each product individually.

2. SAFE COMPOSITION

Feature refinements can make significant changes to classes. Features can introduce new methods and fields to a class and alter the class hierarchy by changing the declared parent of a class. They can also refine existing methods by adding new statements before and after a method's body or by overwriting it altogether.

The features in Figure 2 illustrate how these modifications affect the `Account` class in the feature Bank. The RetirementAccount feature refines the `Account` class by updating its parent to `Lehman`, introducing a new field for a 401k account balance with an initial balance of 10000, and rewrites the definition for the update method to add x to the 401k balance. InvestmentAccount also refines `Account`, updating its

```
feature InvestmentAccount {
  refines class Account extends WaMu {
    int 401kbalance = 0;
    refines void update (int x) {
      x = x/2; Super(); 401kbalance += x;
  }}}

feature RetirementAccount {
  refines class Account extends Lehman {
    int 401kbalance = 10000;
    int update (int x) {
      401kbalance += x;
  }}}

feature Investor {
  class AccountHolder extends Object {
    Account a = new Account();
    void payday (int x; int bonus) {
      a.401kbalance += bonus;
      return a.update(x);
  }}}
```

2: Definitions of InvestmentAccount, Investor, and RetirementAccount features.

```
class Account extends Lehman{
  int balance = 0;
  int 401kbalance = 10000;
  void update(int x) {
    401kbalance += x;
  }}
```

3: RetirementAccount•Bank

parent to `WaMu` and introducing a 401k field, but it refines the update method to put half of x into a 401k before adding the rest to the original account balance.

A software product line can be modelled as an algebra that consists of a set of operations, where each operation implements a feature. We write $M = \{$ Bank, RetirementAccount, InvestmentAccount, Investor$\}$ to mean model M has the features (operations) Bank, RetirementAccount, InvestmentAccount, Investor declared above. One or more features of a model are constants that build base programs through a set of class introductions:

Bank	a program with only the generic `Account` class
Investor	a program with only the `AccountHolder` class

The remaining operations are unary functions on programs, and are program refinements or extensions:

InvestmentAccount•Bank	builds an investment account
RetirementAccount•Bank	builds a retirement account

where • denotes function application and $B \bullet A$ is read as "feature B refines program A" or equivalently "feature B is added to program A". A refinement can extend the program with new definitions or modify existing definitions. The design of a program is a composition of features: a *product specification*.

P_1 = RetirementAccount•Bank	Fig. 3
P_2 = InvestmentAccount•Bank	Fig. 4
P_3 = RetirementAccount•Investor•Bank	Fig. 5

This model of software product lines is based on stepwise development: one begins with a simple program (e.g., constant feature Bank) and builds more complex programs by progressively adding features (e.g., adding feature InvestmentAccount to Bank).

A set of n features can be composed in an exponential

```
class Account extends WaMu{
  int balance = 0;
  int 401kbalance = 0;
  void update(int x) {
    x = x/2;
    int newBal = balance + x;
    balance = newBal;
    401kbalance += x;
  }}
```

4: InvestmentAccount•Bank

```
class Account extends Lehman{
  int balance = 0;
  int 401kbalance = 10000;
  void update(int x) {
    401kbalance += x;
  }}

class AccountHolder extends Object {
  Account a = new Account();
  void payday (int x; int bonus) {
    a.401kbalance += bonus;
    return a.update(x);
  }}
```

5: RetirementAccount•Investor•Bank

number of ways to build a set of order $n!$ programs. A product line is a subset of these programs described by a feature model which constrains the ways in which features can be composed. A composition of features might fail to meet the dependencies of its constituent features, resulting in a program that fails to type check. Only a subset of the programs built from a set of features is well-typed. The goal of safe composition is to ensure that the product line described by a feature model is contained in this set, i.e. that all the programs in the product line are well-typed.

The combinatorial nature of product lines presents a number of problems to statically determining safe composition. The members and methods of a class referenced in a feature might be introduced in several different features. Consider the `AccountHolder` class introduced in the Investor feature: this account holder is the employee of a company which gives a small bonus with each paycheck which the employee adds directly into the 401k balance in his account. In order for a composition including the Investor feature to build a well-typed Java program, it must be composed with a feature that introduces this field to the `Account` class, in this case either InvestmentAccount or RetirementAccount. This requirement could also be met by a feature which sets the parent of `Account` to a different class from which it inherits the `401kbalance` field. Since a parent of a class can change through refinement, the inherited fields and methods of the classes in a feature are dependent on a specific product specification. Each feature has a set of type-safety constraints which can be met by the combination of a number of different features, each with their own set of constraints. To study the interaction of feature composition and type safety, we first develop a model of Java with features.

3. LIGHTWEIGHT FEATURE JAVA

Lightweight Feature Java (LFJ) is a kernel language that captures the key concepts of feature-based product lines of Java programs. LFJ is based on Lightweight Java (LJ), a minimal imperative subset of Java [11]. LJ supports classes, mutable fields, constructors, single inheritance, methods and dynamic method dispatch. LJ does not include local variables, field hiding, interfaces, inner classes, or generics. This imperative kernel provides a minimal foundation for studying a type system for feature-oriented programming. LJ is more appropriate for this work than Featherweight Java [8] because of its treatment of constructors. When composing features, it is important to be able to add new member variables to a class during refinement. Featherweight Java requires all member variables to be initialized in a single constructor call. As a result, adding a new member variable causes all previous constructor calls to be invalid. Lightweight Java allows such refinements through its support of more flexible initialization of member variables. In addition, Lightweight Java has a full formalization in Coq, which we extended to prove the soundness of LFJ mechanically. The proof scripts for the system are available at `http://www.cs.utexas.edu/~bendy/featurejava.php`.

Product specification
`PS` ::= $\overline{\texttt{FD}}$
Feature declarations
`FD` ::= **feature** `F` {$\overline{\texttt{cld}}$; $\overline{\texttt{rcld}}$}
Class refinement
`rcld` ::= **refines class** `dcl` **extending** `cl`{$\overline{\texttt{fd}}$; $\overline{\texttt{md}}$; $\overline{\texttt{rmd}}$}
Method refinement
`rmd` ::= **refines method** `ms` {`rmb`}
Method refinement
`rmb` ::= $\overline{\texttt{s}}$; **Super()**; $\overline{\texttt{s}}$; **return** `y`

6: Modified Syntax of Lightweight Feature Java.

The syntax of LFJ extends LJ to support feature-oriented programming is given in Figure 6. A feature definition *FD* maps a feature name F to a list of class declarations $\overline{cld}$ and a list of class refinements $\overline{rcld}$. A class refinement *rcld* includes a class name *dcl*, a set of LJ field and method introductions, $\overline{fd}$ and $\overline{md}$, a set of method refinements $\overline{rmd}$, and the name of the updated parent class *cl*. A method refinement advises a method with signature *ms* with two lists of LJ statements $\overline{s}$ and an updated return value y. When applied to an existing method, a method refinement wraps the existing method body with the advice. The parameters of the original method are passed implicitly because the refinement has the same signature as the method it refines. The set of features from which a product line can be built is called the *feature table*. A product specification *PS* is a sequence of distinct feature names.

3.1 Feature Composition

A LJ program can be modelled as a partial function from class names to their definitions: $CT : dcl \rightarrow cld$. In the operational semantics of LJ, this function is concretely realized as the function $\mathbf{path} : P \rightarrow dcl \rightarrow cld$ which looks up a class definition in a given program. In this context, CT is simply the **path** specialized on P: $CT = \mathbf{path}_P$. Features are themselves functions from LJ programs to LJ programs. Composition of a feature **feature** `FD` {$\overline{\texttt{cld}}$; $\overline{\texttt{rcld}}$} with an LJ program P produces a new mapping, CT':

$$CT'(dcl) = \begin{cases} \mathbf{path}_{\overline{cld}}(dcl) & dcl \in \overline{cld} \\ \overline{rcld} \bullet CT(dcl) & dcl \notin \overline{cld} \end{cases} \quad (1)$$

In the case that `FD` introduces a class named dcl, CT' returns this class, ignoring any previous declarations and refinements. Otherwise, CT' finds the definition of dcl in the previous program using the original CT function and returns the resulting class definition, cld, refined by $\overline{rcld}$. If a class refinement $rcld$ in $\overline{rcld}$ is named dcl, the $\bullet$ operator builds a refined class by first advising the methods of cld with the method refinements in $rcld$. The fields and methods introduced by $rcld$ are then added to this class and its parent is set to the superclass named in $rcld$. CT' is undefined if cld lacks a method refined by $rcld$.

A product specification builds a LJ program by recursively composing its features in this manner, starting with the empty LJ program. Each LFJ feature table can construct a family of programs through composition. The set of class definitions in a program is determined by the sequence of features which produced it. The class hierarchy is also potentially different in each product: refinements can alter the parent of a class, and two mutually exclusive features can define the same class with a different parent.

3.2 Typechecking Feature Models

A feature model is safe if it only allows the creation of well-formed LJ programs. For any particular product specification, this can be checked by composing its features and then checking the type safety of the resulting program using the standard LJ type system. A naive approach to checking the safety of a feature model is simply to iterate over all the programs it describes, type checking each individually. This approach constructs a potentially exponential number of programs, making it computationally expensive. Instead, we propose a type system which allows us to statically verify that all programs described by a feature model are type-safe without having to synthesize the entire family of programs.

The key difficulty with this approach is that features are typically program fragments which make use of class definitions made in other features; these external dependencies can only be resolved during composition with other features. Every LJ construct has two categories of requirements which must be met in order for it to be well-formed in the LJ type system. The first category consists of premises which only depend on the structure of the construct, e.g. the requirement that the parameters of a well-formed method be distinct. The remaining premises access information from the surrounding program through the CT function. For example, CT is used to determine that the type of a variable y is a subtype of the type of variable x when assigning y to x in a method body. Intuitively, these premises define the structure of the programs in which LJ constructs are well-formed. In the standard LJ type system, the structure of the surrounding program is known. In a software product line, however, each feature can be included in a number of programs, and the final makeup of the surrounding program depends on the other features in a product specification. Converting these kinds of premises into constraints provides an explicit interface for an LJ construct with any surrounding program. A feature's interface determines which features must be included in a product specification in order for its constructs to be well-formed in the final LJ program.

4. LFJ TYPE SYSTEM

In this section, we present a constraint-based type system for LFJ based on a constraint-based type system we have developed for LJ. The constraint-based systems retain the premises that depend on the structure of the construct being typed and convert those that rely on external information into constraints. By using constraints, the external typing requirements for each feature are made explicit, separating derivation of these requirements from their satisfaction. Generating a set of constraints for a feature is separated from consideration of which product specifications have a combination of features satisfying these constraints.

Composition Constraints
- dcl **introduces** ms **before** F
- dcl **introduced before** F

Uniqueness Constraints
- $cl\ f$ **unique in** dcl
- $cl\ m\ (\overline{vd_k}^k)$ **unique in** dcl

Structural Constraints
- $cl_1 \prec cl_2$
- $cl_2 \prec \mathbf{ftype}(cl_1, f)$
- $\mathbf{ftype}(cl_1, f) \prec cl_2$
- $\mathbf{mtype}(cl, m) \prec \overline{cl_k}^k \rightarrow cl$
- $\mathbf{defined}(cl)$
- $f \notin \mathbf{fields}(\mathbf{parent}(dcl))$
- $\mathbf{pmtype}(dcl, m) = \tau$

7: Syntax of Lightweight Feature Java typing constraints.

The constraints used by our type system are given in Figure 7 and are divided into three categories. The two composition constraints guarantee successful composition of a feature F by requiring that refined classes and methods be introduced by a feature in a product line before F. The two uniqueness constraints ensure that member names are not overloaded within the same class, a restriction in the LJ formalization. The structural constraints come from the standard LJ type system and determine the members of a class and its inheritance hierarchy in the final program. The subtype constraint is particularly important because the class hierarchy is malleable until composition; if it were static, constraints that depend on subtyping could be reduced to other constraints or eliminated entirely.

The typing rules for LFJ are found in Figure 8-10 and rely on judgements of the form $\vdash J \mid \xi$, where J is a typing judgement from LFJ and ξ is a set of constraints, called a *signature*. The signature ξ provides an explicit interface which guarantees that J holds in any product specification that satisfies ξ. Typing rules for statements, methods, and classes are those from LJ augmented with signatures. Typing rules for class and method refinements in a feature F are similar to those for the objects they refine, but require that the refined class or method be introduced in a feature that comes before the F in a product specification. Method refinements do not have to check that the names of their parameters are distinct and that their parameter types and return type are well-formed: a method introduction with these checks must precede the refinement in order for it to be well-formed. The signature of a product specification PS

$\boxed{\vdash_{\tau,\mathtt{F}} md \mid \mathcal{C}}$ Method well-formed in class with signature

$$\frac{\begin{array}{c}\mathbf{distinct}(\overline{var_k}^{\,k}) \qquad \overline{\mathbf{type}(cl_k) = \tau_k}^{\,k} \qquad \mathbf{type}(cl) = \tau' \\ \Gamma = [\overline{var_k \mapsto \tau_k}^{\,k}][\mathbf{this} \mapsto \tau] \qquad \overline{\Gamma \vdash s_\ell \mid \mathcal{C}_\ell}^{\,\ell} \qquad \Gamma(y) = \tau''\end{array}}{\vdash_\tau cl\ meth\ (\overline{cl_k\ var_k}^{\,k})\ \{\overline{s_\ell}^{\,\ell}\ \mathbf{return}\ y;\} \mid \{\tau'' \prec \tau', \overline{\mathbf{defined}\ cl_k}^{\,k}\} \cup \bigcup_\ell \mathcal{C}_\ell} \quad \text{(WF-Method)}$$

$\boxed{\vdash cld \mid \mathcal{C}}$ Class well-formed with signature

$$\frac{\begin{array}{c}\mathbf{distinct}(\overline{f_j}) \qquad \mathbf{distinct}(\overline{m_k}) \qquad dcl \neq cl \qquad \mathbf{type}(dcl) = \tau \qquad \overline{\vdash_\tau cl_k\ meth_k\ (\overline{cl_{\ell,k}\ var_{\ell,k}}^{\,\ell})\ mb_k \mid \mathcal{C}_k}^{\,k} \\ \xi = \bigcup_j \{f_j \notin \mathbf{fields}(\mathbf{parent}(dcl))\} \qquad \upsilon = \bigcup_j \{cl_j\ f_j\ \mathbf{unique\ in}\ dcl\} \qquad \upsilon' = \bigcup_k \{cl_k\ meth_k\ (\overline{cl_{\ell,k}\ var_{\ell,k}}^{\,\ell})\ \mathbf{unique\ in}\ dcl\} \\ \xi' = \bigcup_k \{\mathbf{pmtype}(dcl, meth_k) = \overline{cl_{\ell,k}}^{\,\ell} \to cl_k\}\end{array}}{\vdash \mathbf{class}\ dcl\ \mathbf{extends}\ cl\ \{\overline{cl_j\ f_j}^{\,j};\ \overline{cl_k\ meth_k\ (\overline{cl_{\ell,k}\ var_{\ell,k}}^{\,\ell,k})\ mb_k}^{\,k}\ \} \mid \bigcup_k \mathcal{C}_k \cup \{\mathbf{defined}\ cl, \overline{\mathbf{defined}\ cl_j}^{\,j}\} \cup \xi \cup \xi' \cup \upsilon \cup \upsilon'} \quad \text{(WF-Class)}$$

$\boxed{\vdash_{\tau,\mathtt{F}} rmd \mid \mathcal{C}}$ Refined method well-formed in class of feature with signature

$$\frac{\begin{array}{c}\mathbf{type}(cl) = \tau' \qquad \Gamma = [\overline{var_k \mapsto \tau_k}^{\,k}][\mathbf{this} \mapsto \tau] \\ \Gamma(y) = \tau'' \qquad \overline{\Gamma \vdash s_j \mid \mathcal{C}_j}^{\,j} \qquad \overline{\Gamma \vdash s_\ell \mid \mathcal{C}_\ell}^{\,\ell} \\ \mathcal{C} = \{\tau'' \prec \tau', \tau\ \mathbf{introduces}\ cl\ meth\ (\overline{cl_k\ var_k}^{\,k})\ \mathbf{before}\ \mathtt{F}\} \cup \bigcup_j \mathcal{C}_j \cup \bigcup_\ell \mathcal{C}_\ell\end{array}}{\vdash_{\tau,\mathtt{F}} \mathbf{refines\ method}\ cl\ meth\ (\overline{cl_k\ var_k}^{\,k})\ \{\overline{s_j}^{\,j};\ \mathbf{Super}();\ \overline{s_\ell}^{\,\ell};\ \mathbf{return}\ y;\} \mid \mathcal{C}} \quad \text{(WF-Refines-Method)}$$

$\boxed{\vdash_{\mathtt{F}} rcld \mid \mathcal{C}}$ Class refinement well-formed in feature with signature

$$\frac{\begin{array}{c}dcl \neq cl \qquad \mathbf{type}(dcl) = \tau \qquad \overline{\vdash_\tau cl_k\ meth_k\ (\overline{cl_{\ell,k}\ var_{\ell,k}}^{\,\ell})\ mb_k \mid \mathcal{C}_k}^{\,k} \qquad \overline{\vdash_{\tau,\mathtt{F}} rmd_m \mid \mathcal{C}'_m}^{\,m} \\ \xi = \bigcup_j \{f_j \notin \mathbf{fields}(\mathbf{parent}(dcl))\} \qquad \upsilon = \bigcup_j \{cl_j\ f_j\ \mathbf{unique\ in}\ dcl\} \qquad \upsilon' = \bigcup_k \{cl_k\ meth_k\ (\overline{cl_{\ell,k}\ var_{\ell,k}}^{\,\ell})\ \mathbf{unique\ in}\ dcl\} \\ \xi' = \bigcup_k \{\mathbf{pmtype}(dcl, meth_k) = \overline{cl_{\ell,k}}^{\,\ell} \to cl_k\}\end{array}}{\vdash_{\mathtt{F}} \mathbf{refines\ class}\ dcl\ \mathbf{extending}\ \mathtt{cl}\ \{\overline{cl_j\ f_j}^{\,j}; \overline{\vdash_\tau cl_k\ meth_k\ (\overline{cl_{\ell,k}\ var_{\ell,k}}^{\,\ell,k})\ mb_k}^{\,k};\ \overline{rmd_{\ell,k}}^{\,\ell,k}\} \mid \bigcup_k \mathcal{C}_k \cup \bigcup_m \mathcal{C'}_m \cup \{\mathbf{defined}\ cl, \overline{\mathbf{defined}\ cl_j}^{\,j}, dcl\ \mathbf{introduced\ before}\ \mathtt{F},\} \cup \xi \cup \xi' \cup \upsilon \cup \upsilon'} \quad \text{(WF-Refines-Class)}$$

9: Typing Rules for LFJ method and class refinements.

is the union of the constraints on each of the features in PS.

Once the signature of a product specification PS is generated according to the rules in Figure 10, we evaluate whether it is satisfied by PS using the rules in Figure 11. Compositional constraints on a feature F are satisfied when a feature with the appropriate introductions precedes F in PS. Uniqueness constraints are satisfied when no two features in PS introduce a member with the same name but different signatures to a class dcl. In LFJ, satisfaction of structural constraints is evaluated as in LJ, replacing uses of **path** with the CT function built by composition of the features in PS.

The compositional and uniqueness constraints guarantee that each step during the composition of a product specification builds an intermediate program. These programs need not be well-formed: they could rely on definitions which are introduced in a later feature or have classes used to satisfy typing constraints which could also be overwritten by a subsequent feature. For this reason, our typing rules only consider the final product specification, making no guarantees about the behavior of intermediate programs.

4.1 Soundness of the LFJ Type System

The soundness proof is based on successive refinements of the type systems of LJ and LFJ, reducing them to the proofs of progress and preservation of the original LJ type system given in [11]. We first use our constraint-based type system for LJ, utilizing the structural constraints listed in Figure 7 and the corresponding judgements in Figure 11 to check constraint satisfaction. This type system is shown to be equivalent to the original LJ type system, in that a program with unique class names and an acyclic class hierarchy satisfies its signature if and only if it is well-formed according to the original typing rules. We then show that if a single LFJ product specification is well-formed according to the constraint-based LFJ type system, it produces a LJ program that is also well-formed. We have formalized in the Coq proof assistant the syntax and semantics of LJ and LFJ presented in the previous section, as well as all of the

$\boxed{\Gamma \vdash s \mid \mathcal{C}}$ Statement well-formed in context with signature

$$\frac{\overline{\Gamma \vdash s_k \mid \mathcal{C}_k}^{k}}{\Gamma \vdash \{s_k\} \mid \bigcup_k \mathcal{C}_k} \quad \text{(WF-Block)}$$

$$\frac{\Gamma(x) = \tau_1 \qquad \Gamma(var) = \tau_2}{\Gamma \vdash var = x; \mid \{\tau_1 \prec \tau_2\}} \quad \text{(WF-Var-Assign)}$$

$$\frac{\Gamma(x) = \tau_1 \qquad \Gamma(var) = \tau_2}{\Gamma \vdash var = x.f; \mid \{\mathbf{ftype}(\tau_1, f) \prec \tau_2\}} \quad \text{(WF-Field-Read)}$$

$$\frac{\Gamma(x) = \tau_1 \qquad \Gamma(y) = \tau_2}{\Gamma \vdash x.f = y; \mid \{\tau_2 \prec \mathbf{ftype}(\tau_1, f)\}} \quad \text{(WF-Field-Write)}$$

$$\frac{\begin{array}{c}\Gamma(x) = \tau_1 \qquad \Gamma(y) = \tau_2 \\ \Gamma \vdash s_1 \mid \mathcal{C}_1 \qquad \Gamma \vdash s_2 \mid \mathcal{C}_2 \\ \mathcal{C}_3 = \{\tau_2 \prec \tau_1 \vee \tau_1 \prec \tau_2\}\end{array}}{\Gamma \vdash \mathbf{if}\ x == y\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3} \quad \text{(WF-If)}$$

$$\frac{\Gamma(var) = \tau_1 \qquad \mathbf{type}\ (cl) = \tau_2}{\Gamma \vdash var = \mathbf{new}\ cl() \mid \{\tau_2 \prec \tau_1\}} \quad \text{(WF-New)}$$

$$\frac{\begin{array}{c}\Gamma(x) = \tau \qquad \Gamma(var) = \pi \qquad \overline{\Gamma(y_k) = \pi_k}^{k} \\ \mathcal{C} = \{\mathbf{mtype}(\tau, meth) \prec \overline{\pi_k}^{k} \to \pi\}\end{array}}{\Gamma \vdash var = x.meth(\overline{y_k}^{k}) \mid \mathcal{C}} \quad \text{(WF-MCall)}$$

8: Typing Rules for LJ and LFJ statements.

soundness proofs that follow. For this reason, the following sections elide many of the bookkeeping details, instead presenting sketches of the major pieces of the soundness proofs.

Theorem 4.1 (Soundness of constraint-based LJ Type System)**.** *Let P be a LJ program with distinct class names and an acyclic, well-founded class hierarchy. Let $\mathcal{C}$ be the set of constraints generated by a class cld in P. cld is well-formed if and only if P satisfies $\mathcal{C}$: $P \vdash cld \leftrightarrow P \models \mathcal{C}$ where $\vdash cld \mid \mathcal{C}$.*

Proof. The two key pieces of this proof are: showing that satisfaction of each of the constraints guarantees that the corresponding judgement holds, and that there is a one-to-one correspondence between the constraints generated by the typing rules in Figure 9 and the premises used in the declarative LJ type system. The former is straightforward except for the subtyping constraint, which relies on the **path** function to check for satisfaction. We can prove their equivalence by induction on the derivation of the subtyping judgement in one direction and induction on the length of the path in the other. We can then show that the two type systems are equivalent by examination of the structure of P. At each level of the typing rules, the structural premises are identical and each of the external premises of the rules appears as a constraint in the signature. As a result of the previous argument, satisfaction of the signature guarantees that premises of the typing rules hold for each structure in P. Having shown the two type systems are equivalent, the

$\boxed{\vdash P \mid \mathcal{C}}$ Program well-formed with signature

$$\frac{\begin{array}{c}\overline{\vdash cld_k \mid \mathcal{C}_k}^{k} \qquad P = \overline{cld_k}^{k} \\ \mathbf{distinct\ names}\ (P)\end{array}}{\vdash P \mid \bigcup_k \mathcal{C}_k} \quad \text{(WF-Program)}$$

$\boxed{\vdash \mathtt{F} \mid \mathcal{C}}$ Feature well-formed with signature

$$\frac{\overline{\vdash cld_k \mid \mathcal{C}_k}^{k} \qquad \overline{\vdash_F rcld_\ell \mid \mathcal{C}_\ell}^{\ell}}{\vdash \mathbf{feature}\ \mathtt{F}\ \{\overline{cld_k}^{k}\ \overline{rcld_\ell}^{\ell}\} \mid \bigcup_k \mathcal{C}_k \cup \bigcup_\ell \mathcal{C}_\ell} \quad \text{(WF-Feature)}$$

$\boxed{\vdash \mathtt{PS} \mid \mathcal{C}}$ Product specification well-formed with signature

$$\vdash \emptyset \mid \emptyset \quad \text{(WF-Specification-Nil)}$$

$$\frac{\vdash F \mid \mathcal{C} \qquad \vdash \overline{F_k}^{k} \mid \mathcal{C}'}{\vdash F, \overline{F_k}^{k} \mid \mathcal{C} \cup \mathcal{C}'} \quad \text{(WF-Specification)}$$

10: Typing Rules for LFJ Programs and Features.

proofs of progress and preservation for the constraint-based type system follow immediately. □

Theorem 4.2 (Soundness of LFJ Type System)**.** *Let PS be a LFJ product specification and $\mathcal{C}$ be a set of constraints such that $\vdash PS \mid \mathcal{C}$. If $PS \models \mathcal{C}$ and **Object** is in the path of every class introduced by a feature in PS, then the composition of the features in PS produces a valid, well-formed LJ program.*

Proof. This proof is decomposed into three key lemmas, corresponding to the three kinds of typing constraints:

(i) Composition of the features in PS produces a valid LJ program, P.

For each class or method refinement of a feature F in PS, a composition constraint is generated by the LFJ typing rules. Each of these are satisfied according to the definition in Figure 11, allowing us to conclude that a feature with appropriate declarations appears before F in PS. Each of these declarations will appear in the program generated by the features preceding F, allowing us to conclude that the composition succeeds for each feature in PS.

(ii) P is typeable in the constraint-based LJ type system with constraints $\mathcal{C}'$.

In essence, we must show that the premises of the constraint-based LJ typing judgements hold. Our assumption that each class in PS is a descendant of **Object** ensures that P has an acyclic, well-founded class hierarchy. The premises for the LJ methods and statements are identical, leaving class typing rules for us to consider. The LJ typing rules require that the method and field names for a class be distinct, but these premises are removed by the LFJ typing rules, as the members of a class are not finalized until after composition. This requirement is instead enforced by the uniqueness constraints in Figure 11, which are satisfied only when

$$\frac{\mathbf{ftype}(P,\tau_1,f)=\tau_3 \qquad \tau_2 \in \mathbf{path}(P,\tau_3)}{P \models \tau_2 \prec \mathbf{ftype}(\tau_1,f)}$$

$$\frac{\mathbf{ftype}(P,\tau_1,f)=\tau_3 \qquad \tau_3 \in \mathbf{path}(P,\tau_2)}{P \models \mathbf{ftype}(\tau_1,f) \prec \tau_2}$$

$$\frac{\mathbf{mtype}(P,\tau,m)=\overline{\pi'_k}^{\,k} \to \pi' \qquad \pi' \in \mathbf{path}(P,\pi) \qquad \overline{\pi_k \in \mathbf{path}(P,\pi'_k)}^{\,k}}{P \models \mathbf{mtype}(\tau,m) \prec \overline{\pi_k}^{\,k} \to \pi}$$

$$\frac{\mathbf{type}(cl) \in \mathbf{path}(P,\mathbf{type}(cl))}{P \models \mathbf{defined}(cl)}$$

$$\frac{\tau_2 \in \mathbf{path}(P,\tau_1)}{P \models \tau_1 \prec \tau_2}$$

$$\frac{\mathbf{ftype}(P,\mathbf{parent}(dcl),f)=\bot}{P \models f \notin \mathbf{fields}(\mathbf{parent}(dcl))}$$

$$\frac{\mathbf{mtype}(P,\mathbf{parent}(dcl),m)=\bot \ \vee \ \mathbf{mtype}(P,\mathbf{parent}(dcl),m)=\tau}{P \models \mathbf{pmtype}(dcl,m)=\tau}$$

$$\frac{FP=\overline{A_k}^{\,k} F \overline{B_\ell}^{\,\ell} H \overline{C_j}^{\,j} \qquad \tau.ms \in H \qquad \tau \notin \mathbf{introductions}(\overline{B_\ell}^{\,\ell})}{FP \models \tau \ \mathbf{introduces}\ ms \ \mathbf{before}\ F}$$

$$\frac{FP=\overline{A_k}^{\,k} F \overline{B_\ell}^{\,\ell} H \overline{C_j}^{\,j} \qquad \mathbf{dcl} \in H}{FP \models dcl \ \mathbf{introduced\ before}\ F}$$

$$\frac{\mathbf{type}(dcl)=\tau \qquad \forall A,B \in FP, \tau.cl_1\ f \in A \wedge \tau.cl_2\ f \in B \to cl_1=cl_2}{FP \models cl\ f\ \mathbf{unique\ in}\ dcl}$$

$$\frac{\mathbf{type}(dcl)=\tau \qquad ms_1=cl\ m\ (\overline{vd_k}^{\,k}) \qquad ms_2=cl'\ m\ (\overline{vd'_k}^{\,k}) \qquad \forall A,B \in FP, \tau.ms_1 \in A \tau.ms_2 \in B \to ms_1=ms_2}{FP \models cl\ m\ (\overline{vd_k}^{\,k})\ \mathbf{unique\ in}\ dcl}$$

11: Satisfaction of typing constraints.

a method or field name is introduced by a single feature. Since $PS \models \mathcal{C}$, it follows that the premises of the LJ typing rules hold for P and that there exists some set of constraints $\mathcal{C}'$ such that $\vdash P \mid \mathcal{C}'$.

(iii) P satisfies the constraints in $\mathcal{C}'$ and is thus a well-formed LJ program.

We break this proof into two sublemmas:

(a) $\mathcal{C}' \subseteq \mathcal{C}$.

The key observation for this proof is that every class, method, and statement in P originated from some feature in PS. The most interesting case is for the constraints generated by method bodies: a statement contained in a method body can come from either the initial introduction of that method or advice added by a method refinement. In either case, the statement was included in some feature in PS and thus generated some set of constraints in $\mathcal{C}$. Because method signatures are fixed across refinement, the context used in typing both kinds of statements is the same as that used for the method in the final composition. This does not entail that $\mathcal{C} = \mathcal{C}'$, however, as there could be some construct introduced in PS that is overwritten by an introduction in a subsequent feature.

(b) For any structural constraint $\mathcal{K}$, if $PS \models \mathcal{K}$, then $P \models \mathcal{K}$.

This reduces to showing that class declaration returned by $CT(dcl)$ is the same as the class with that identifier in P. This follows from tracing the definition of the CT function down to the final introduction of dcl in the product line. From here, we know that this class appears in the program synthesized from the product specification starting with this feature. Further refinements of this class are reflected in the $\bullet$ operator used recursively to build $CT(dcl)$; each refinement succeeds by (i) above. Since the two functions are the same, the helper functions which call **path** in P (i.e. **ftype**, **mtype**) and those that use CT in PS return the same values. We can thus conclude that the satisfaction judgements for PS and P are equivalent.

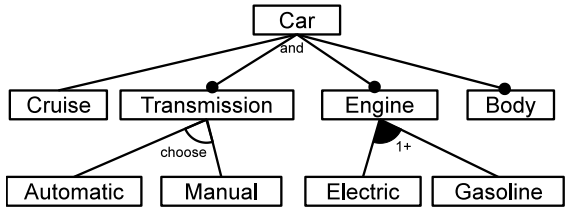
All constraints in $\mathcal{C}'$ appear in $\mathcal{C}$, so $PS \models \mathcal{C}'$. By (b) above, it follows that $P \models \mathcal{C}'$. P must therefore be a well-formed LJ program by Theorem 4.1. □

5. FEATURE MODELS

A *feature model* represents the dependencies and constraints between features that make up a product line. One common representation for feature models is a *feature diagram*. A feature diagram is a hierarchy of features where each node in the tree corresponds to a feature. Annotations on the tree represent constraints. Features required by a parent are marked with a dot.

5.1 Feature Diagrams

Consider an elementary automotive product line that differentiates cars by transmission type (automatic or manual), engine type (electric or gasoline), and the option of cruise control. Figure 12 shows the feature diagram of this product line. A car has a body, engine, transmission, and optionally a cruise control. A transmission is either automatic or manual (choose one), and an engine is electric-powered, gasoline-powered, or both.



12: Feature diagram

Besides hierarchical relationships, feature models also allow cross-tree constraints, although these are more difficult to represent in a feature diagram. Such constraints are often inclusion or exclusion statements of the form: if feature F is included in a product, then features A and B must also be included (or excluded). A cross-tree constraint is that cruise control requires an automatic transmission.

Feature models are compact representations of propositional formulas [5]. We exploit this representation in relating

$$
\begin{array}{ll}
\tau_1 \prec \tau_2 & \Rightarrow \mathbf{Sty}_{\tau_1,\tau_2} \\
\tau_2 \prec \mathbf{ftype}(\tau_1, f) & \Rightarrow \bigvee\{\mathbf{Sty}_{\tau_2,cl} \wedge \mathbf{Sty}_{\tau_1,\mathbf{type}(cld)} \wedge \mathbf{FinalIn}_{\mathbf{name}(cld),F} \mid \exists cld \in \mathbf{clds}(F), \exists cl, cl\ f \in \mathbf{fds}(cld)\} \vee \\
 & \quad \bigvee\{\mathbf{Sty}_{\tau_2,cl} \wedge \mathbf{Sty}_{\tau_1,\mathbf{type}(rcld)} \wedge \mathbf{Final}_{\mathbf{name}(rcld),F} \mid \exists rcld \in \mathbf{rclds}(F), \exists cl, cl\ f \in \mathbf{fds}(rcld)\} \\
\mathbf{ftype}(\tau_1, f) \prec \tau_2 & \Rightarrow \bigvee\{\mathbf{Sty}_{cl,\tau_2} \wedge \mathbf{Sty}_{\tau_1,\mathbf{type}(cld)} \wedge \mathbf{FinalIn}_{\mathbf{name}(cld),F} \mid \exists cld \in \mathbf{clds}(F) \exists cl, cl\ f \in \mathbf{fds}(cld)\} \vee \\
 & \quad \bigvee\{\mathbf{Sty}_{cl,\tau_2} \wedge \mathbf{Sty}_{\tau_1,\mathbf{type}(rcld)} \wedge \mathbf{Final}_{\mathbf{name}(rcld),F} \mid \exists rcld \in \mathbf{rclds}(F), \exists cl, cl\ f \in \mathbf{fds}(rcld)\} \\
\mathbf{mtype}(\tau, m) \prec \overline{\pi_k}^k \to \pi & \Rightarrow \bigvee\{\mathbf{Sty}_{cl,\pi} \wedge \bigwedge_k \mathbf{Sty}_{\pi_k,cl_k} \wedge \mathbf{FinalIn}_{\mathbf{name}(cld),F} \mid \exists cld \in \mathbf{clds}(F), \\
 & \qquad \exists cl, \overline{cl_k}^k, \overline{v_k}^k cl\ m(\overline{cl_k v_k}^k) \in \mathbf{mds}(cld)\} \vee \\
 & \quad \bigvee\{\mathbf{Sty}_{cl,\pi} \wedge \bigwedge_k \mathbf{Sty}_{\pi_k,cl_k} \wedge \mathbf{Final}_{\mathbf{name}(rcld),F} \mid \exists rcld \in \mathbf{rclds}(F), \\
 & \qquad \exists cl, \overline{cl_k}^k, \overline{v_k}^k cl\ m(\overline{cl_k v_k}^k) \in \mathbf{mds}(rcld)\} \\
\mathbf{defined}(cl) & \Rightarrow \bigvee\{\mathbf{In}_F \mid \exists cld \in \mathbf{clds}(F), \mathbf{name}(cld) = cl\} \\
\tau \textbf{ introduces } ms \textbf{ before } F & \Rightarrow \bigvee\{\mathbf{In}_G \wedge \mathbf{Prec}_{G,F} \wedge \bigwedge\{\mathbf{In}_H \to \mathbf{Prec}_{F,H} \vee \mathbf{Prec}_{H,G} \mid \exists cld' \in \mathbf{clds}(H)), \mathbf{type}(\mathbf{name}(cld')) = \tau\} \\
 & \quad \mid \exists cld \in \mathbf{clds}(G), \mathbf{type}(\mathbf{name}(cld)) = \tau \wedge ms \in \mathbf{methods}(\mathbf{mds}(cld))\} \vee \\
 & \quad \bigvee\{\mathbf{In}_G \wedge \mathbf{Prec}_{G,F} \wedge \bigwedge\{\mathbf{In}_H \to \mathbf{Prec}_{F,H} \vee \mathbf{Prec}_{H,G} \mid \exists cld' \in \mathbf{clds}(H)), \mathbf{type}(\mathbf{name}(cld')) = \tau\} \\
 & \quad \mid \exists rcld \in \mathbf{rclds}(G), \mathbf{type}(\mathbf{name}(rcld)) = \tau \wedge ms \in \mathbf{methods}(\mathbf{mds}(rcld))\} \\
dcl \textbf{ introduced before } F & \Rightarrow \bigvee\{\mathbf{In}_G \wedge \mathbf{Prec}_{G,F} \mid \exists cld \in \mathbf{clds}(F), \mathbf{name}(cld) = dcl\} \\
cl\ f \textbf{ unique in } dcl & \Rightarrow \bigwedge\{\neg\mathbf{In}_F \mid \exists cld \in \mathbf{clds}(F), \mathbf{name}(cld) = dcl \wedge \exists cl', cl' f \in \mathbf{fds}(cld) \wedge cl \neq cl'\} \wedge \\
 & \quad \bigwedge\{\neg\mathbf{In}_F \mid \exists rcld \in \mathbf{rclds}(F), \mathbf{name}(rcld) = dcl \wedge \exists cl', cl' f \in \mathbf{fds}(rcld) \wedge cl \neq cl'\} \\
cl\ m\ (\overline{vd_k}^k) \textbf{ unique in } dcl & \Rightarrow \bigwedge\{\neg\mathbf{In}_F \mid \exists cld \in \mathbf{clds}(F), \mathbf{name}(cld) = dcl \wedge \exists cl', \overline{vd'_k}^k, cl' m\ (\overline{vd'_k}^k) \in \mathbf{mds}(cld) \wedge cl \neq cl' \vee \\
 & \qquad (\bigvee_k vd_k \neq vd'_k)\} \wedge \\
 & \quad \bigwedge\{\neg\mathbf{In}_F \mid \exists rcld \in \mathbf{rclds}(F), \mathbf{name}(rcld) = dcl \wedge \exists cl', \overline{vd'_k}^k, cl' m\ (\overline{vd'_k}^k) \in \mathbf{mds}(rcld) \wedge cl \neq cl' \vee \\
 & \qquad (\bigvee_k vd_k \neq vd'_k)\} \\
f \notin \mathbf{fields}(\mathbf{parent}(dcl)) & \Rightarrow \bigwedge\{\mathbf{In}_F \wedge \mathbf{FinalIn}_{\mathbf{name}(cld),F} \to \neg\mathbf{Sty}_{\mathbf{type}(dcl),cl} \mid \\
 & \quad \exists cld \in \mathbf{clds}(F), \mathbf{name}(cld) = cl \wedge dcl \neq cl \wedge \exists cl', cl' f \in \mathbf{fds}(cld)\} \wedge \\
 & \quad \bigwedge\{\mathbf{In}_F \wedge \mathbf{Final}_{\mathbf{name}(rcld),F} \to \neg\mathbf{Sty}_{\mathbf{type}(dcl),cl} \mid \\
 & \quad \exists rcld \in \mathbf{rclds}(F), \mathbf{name}(rcld) = cl \wedge dcl \neq cl \wedge \exists cl', cl' f \in \mathbf{fds}(rcld)\} \\
\mathbf{pmtype}(dcl, m) = \tau & \Rightarrow \bigwedge\{\mathbf{In}_F \wedge \mathbf{FinalIn}_{\mathbf{name}(cld),F} \to \neg\mathbf{Sty}_{\mathbf{type}(dcl),cl} \mid \exists cld \in \mathbf{clds}(F), \mathbf{name}(cld) = cl \\
 & \quad \wedge dcl \neq cl \wedge m \in \mathbf{methods}(cld) \wedge \mathbf{mtype}(cld, m) \neq \tau \\
 & \quad \bigwedge\{\mathbf{In}_F \wedge \mathbf{Final}_{\mathbf{name}(cld),F} \to \neg\mathbf{Sty}_{\mathbf{type}(dcl),cl} \mid \exists rcld \in \mathbf{rclds}(F), \mathbf{name}(rcld) = cl \\
 & \quad \wedge dcl \neq cl \wedge m \in \mathbf{methods}(rcld) \wedge \mathbf{mtype}(rcld, m) \neq \tau
\end{array}
$$

where

$$
\begin{array}{ll}
\mathbf{FinalIn}_{cl,F} & \leftrightarrow \mathbf{In}_F \wedge \bigwedge\{\mathbf{In}_G \to \mathbf{Prec}_{G,F} \mid cl \in \mathbf{names}(\mathbf{clds}(G)) \wedge G \neq F\} \\
\mathbf{Final}_{cl,F} & \leftrightarrow \mathbf{In}_F \wedge \bigwedge\{\mathbf{In}_G \to \mathbf{Prec}_{G,F} \mid cl \in \mathbf{names}(\mathbf{clds}(G))\}
\end{array}
$$

13: Translation of constraints to propositional formulas.

feature models to the constraint-based type system for LFJ. A given program specification can be tested for inclusion in a product line by checking if it satisfies the constraints expressed in a feature model. For example, the feature model *Auto* of the automotive product line is:

(Body ∧ (Automatic ∨ Manual) ∧
(Electric ∨ Gasoline) ∧ (Automatic ↔ ¬ Manual))

where Body is the lone constant.

6. SAFETY OF FEATURE MODELS

By the soundness of the LFJ type system, if a product specification satisfies the signature of every feature included in it, its composition is a well-formed LJ program. The signature of a feature F provides an interface with other feature modules. This interface can be translated into a propositional formula describing the minimal structural requirements that any product specification built from a feature table FT which includes F must satisfy in order for the constructs in F to be well-formed. The conjunction of these formulas builds a formula ϕ_{safe} which any product specification must satisfy in order to produce a well-formed program. The safety of a feature model can then be statically verified by using a SAT solver to check that its propositional representation implies this minimal formula.

The propositional variables of ϕ_{safe} have three basic forms,

$\mathbf{In}_A$: Feature A is included.
$\mathbf{Prec}_{A,B}$: Feature A precedes Feature B.
$\mathbf{Sty}_{\tau_1,\tau_2}$: τ_1 is a subtype of τ_2.

14: Description of propositional variables.

described in Figure 14. Note that a satisfying assignment to the **In** and **Prec** variables which obeys the properties of the precedence relation describes a unique product specification. The propositional constraints that impose these properties are given in Figure 15. The first three formulas enforce that a precedence relation is total on all features included in a product specification, that it is asymmetric, and that it is irreflexive. The next four ensure that each product specification dictates an assignment to the **Sty** variables corresponding to its class hierarchy. In effect, the STY_TOTAL rule builds the transitive closure of the subtyping relation, starting with the parent/child relationships established by the last definition of a class in a product specification. A satisfying assignment to WF_{Spec}, the conjunction of all these constraints, represents a unique product specification.

The makeup of the program built from a product specification depends upon the ordering of features and their introductions and refinements. The rules in Figure 13 gen-

erate a propositional formula for each kind of typing constraint. A satisfying assignment to a formula in Figure 13 which also satisfies WF_{Spec} represents a product specification which satisfies the associated constraint. The **Final** and **FinalIn** abbreviations ensure that introductions and refinements in features appearing before the feature with the final introduction are ignored. The composition and uniqueness constraints have straightforward propositional representations that govern the valid orderings and makeup of product lines. The translations of the structural constraints rely on the mutability of the class hierarchy: any class cl_1 that has a required field or method could ultimately satisfy a constraint on the members of another class, cl_2, if $cl_2 \prec cl_1$ in the final product specification.

Let ϕ_F be the conjunction of the formulas built from each constraint in the signature of a feature F according to the rules in Figure 13. ϕ_F describes the structure of all product specifications in which F is well-formed. ϕ_{safe} is constructed by first building a clause for each feature F stating its inclusion implies ϕ_F: $\mathbf{In}_F \rightarrow \phi_F$. The propositional constraints generated by STY_WF in Figure 15 are then added to this formula to ensure that the class hierarchy of a product specification is acyclic by requiring that each class included in a product specification be a subtype of `Object`.

The representation of a feature model in propositional logic, FM, describes the assignments that represent legitimate specifications of a product line, defining the family of programs it contains. It is possible to build FM using the variables in Figure 14. By construction, a satisfying assignment to ϕ_{safe} which sets $\mathbf{In}_F$ to true also satisfies ϕ_F. It follows that any satisfying assignment to $WF_{Spec} \rightarrow \phi_{safe}$ represents a product specification which satisfies the signatures of each of the features it includes. By Theorem 4.2, such a product specification produces a well-formed LJ program. Since FM and the minimal well-formedness formula share the same variables, a SAT solver can check whether $FM \wedge WF_{Spec} \rightarrow \phi_{safe}$ is valid. If so, the set of programs described by the feature model is a subset of those allowed by ϕ_{safe}. Thus, the composition of any product specification allowed by such a feature model is well-formed.

6.1 Feasibility of Our Approach

While checking the validity of $FM \wedge WF_{Spec} \rightarrow \phi_{safe}$ is co-NP-complete, the SAT instances generated by our approach are highly structured, making them amenable to fast analysis by modern SAT solvers. We have previously implemented a system based on this approach for checking safe composition of AHEAD software product lines [12]. The size statistics for the four product lines analyzed are presented in Table 1. The tools identified several errors in the existing feature models of these product lines. It took less than 30 seconds to analyze the code, generate the SAT formula, and run the SAT solver for JPL, the largest product line. This is less than the time it took to generate and compile a single program in the product line.

Product Line	# of Features	# of Prog.	Code Base Jak/Java LOC	Program Jak/Java LOC
PPL	7	20	2000/2000	1K/1K
BPL	17	8	12K/16K	8K/12K
GPL	18	80	1800/1800	700/700
JPL	70	56	34K/48K	22K/35K

1: Product Line Statistics from [12].

7. RELATED WORK

Our strategy of representing feature models as propositional formulas in order to verify their consistency was first proposed in [5]. The authors checked the feature models against a set of user-provided feature dependences of the form $F \rightarrow A \vee B$ for features F, A, and B. This approach was adopted by Czarnecki and Pietroszek [7] to verify software product lines modelled as feature-based model templates. The product line is represented as an UML specification whose elements are tagged with boolean expressions representing their presence in an instantiation. These boolean expressions correspond to the inclusion of a feature in a product specification. These templates typically have a set of well-formedness constraints which each instantiation should satisfy. In the spirit of [5], these constraints are converted to a propositional formula; feature models are then checked against this formula to make sure that they do not allow ill-formed template instantiations.

The previous two approaches relied on user-provided constraints when validating feature models. The genesis of our current approach was a system developed by Thaker et al. [12] which generated the implementation constraints of an AHEAD product line of Java programs by examining field, method, and class references in feature definitions. Analysis of existing product lines using this system detected previously unknown errors in their feature models. This system relied on a set of rules for generating these constraints with no formal proof showing they were necessary and sufficient for well-formedness, which we have addressed here.

If features are thought of as modules, the feature model used to describe a product line is a *module interconnection language* [9]. Normally, the typing requirements for a module would be explicitly listed by a "requires-and-provides interface" for each module. We instead infer a module's "requires" interface automatically by considering the minimum structural requirements imposed by the the type system. We verify that these interface constraints are satisfied by the implicit "provides" interface for each feature module in a product specification. If composition is a linking process, we are guaranteeing that there will be no linking errors. The difference with normal linking is that we check all combinations of linkings allowed by the feature model.

A similar type system was proposed by Anacona et al. to type check, compile, and link source code fragments [1]. Like features, the source code fragments they considered could reference external class definitions, requiring other fragments to be included in order to build a well-typed program. These code fragments were compiled into bytecode fragments augmented with typing constraints that ranged over type variables, similar to the constraints used in the LFJ typing rules. The two approaches use these constraints for different purposes, however. Anacona et al. solve these constraints during a linking phase which combines individually compiled bytecode fragments. If all the constraints are resolved during linking, the resulting code is the same as if all the pieces had been globally compiled. Our system uses these constraints to type check a family of programs which can be built from a known set of features.

The existing work on type checking feature-oriented lan-

$$
\begin{array}{rl}
\textsc{Prec\_Total}: & \forall A, B, A \neq B, \mathbf{In}_A \wedge \mathbf{In}_B \leftrightarrow (\mathbf{Prec}_{A,B} \vee \mathbf{Prec}_{B,A}) \\
\textsc{Prec\_ASym}: & \forall A, B, \mathbf{Prec}_{A,B} \rightarrow \neg \mathbf{Prec}_{B,A} \\
\textsc{Prec\_Irrefl}: & \forall A, \neg \mathbf{Prec}_{A,A} \\
\textsc{Sty\_Refl}: & \forall \tau, \mathbf{Sty}_{\tau,\tau} \leftrightarrow \bigvee \{\mathbf{In}_F \mid cld \in \mathbf{clds}(F) \wedge \mathbf{type}(\mathbf{name}(cld)) = \tau\} \\
\textsc{Sty\_Obj}: & \mathbf{Sty}_{\texttt{Object},\texttt{Object}} \\
\textsc{Sty\_ASym}: & \forall \tau_1, \tau_2, \mathbf{Sty}_{\tau_1,\tau_2} \rightarrow \neg \mathbf{Sty}_{\tau_2,\tau_1} \\
\textsc{Sty\_Total}: & \forall \tau_1, \tau_2, \tau_3, \mathbf{Sty}_{\tau_1,\tau_2} \leftrightarrow ((\mathbf{Sty}_{\tau_1,\tau_3} \wedge \mathbf{Sty}_{\tau_3,\tau_2}) \vee \\
& \bigvee \{\mathbf{In}_F \mid \exists cld \in \mathbf{clds}(F), \mathbf{type}(\mathbf{name}(cld)) = \tau_1 \wedge \mathbf{type}(\mathbf{parent}(cld)) = \tau_2\} \wedge \\
& \bigwedge \{\mathbf{In}_G \rightarrow \mathbf{Prec}_{G,F} \mid G \neq F \wedge \exists cld \in \mathbf{clds}(G), \mathbf{type}(\mathbf{name}(cld)) = \tau_1\} \wedge \\
& \bigwedge \{\mathbf{In}_G \rightarrow \mathbf{Prec}_{G,F} \mid G \neq F \wedge \exists rcld \in \mathbf{rclds}(G), \mathbf{type}(\mathbf{name}(rcld)) = \tau_1\} \vee \\
& \bigvee \{\mathbf{In}_F \mid \exists rcld \in \mathbf{rclds}(F), \mathbf{type}(\mathbf{name}(rcld)) = \tau_1 \wedge \mathbf{type}(\mathbf{parent}(cld)) = \tau_2 \wedge \\
& \mathbf{name}(rcld) \notin \mathbf{names}(\mathbf{clds}(F))\} \wedge \\
& \bigwedge \{\mathbf{In}_G \rightarrow \mathbf{Prec}_{G,F} \mid G \neq F \wedge \exists cld \in \mathbf{clds}(G), \mathbf{type}(\mathbf{name}(cld)) = \tau_1\} \wedge \\
& \bigwedge \{\mathbf{In}_G \rightarrow \mathbf{Prec}_{G,F} \mid G \neq F \wedge \exists rcld \in \mathbf{rclds}(G), \mathbf{type}(\mathbf{name}(rcld)) = \tau_1\}\ ) \\
\textsc{Sty\_WF}: & \forall A, \forall c \in \mathbf{clds}(A), \mathbf{In}_A \rightarrow \mathbf{Sty}_{\mathbf{ty}(\mathbf{name}(c)),\texttt{Object}}
\end{array}
$$

15: Constraints on the precedence and subtyping relations.

guages has focused on checking a single product specification, as opposed to checking an entire product line. Apel et al. [3] propose a type system for a model of feature-oriented programming based on Featherweight Java [8] and prove soundness for it and some further extensions of the model. *g*DEEP [2] is a language-independent calculus designed to capture the core ideas of feature refinement. The type system for *g*DEEP transfers information across feature boundaries and is combined with the type system for an underlying language to type feature compositions.

8. CONCLUSION

A feature model is a set of constraints describing how a set of features may be composed to build the family of programs in a product line. This feature model is safe if it only allows the construction of well-formed programs. Simply iterating all the programs described by the feature model is computationally expensive and impractical for large product lines. In order to statically verify that a product line is safe, we have developed a calculus for studying feature composition in Java and a constraint-based type system for this language. The constraints generated by the typing rules provide an interface for each feature. We have shown that the set of constraints generated by our type system is sound with respect to LJ's type system. We verify the type safety of a product line by constructing SAT-instances for the interfaces of each feature. The satisfaction of the formula built from these SAT-instances ensures the product specification corresponding to the satisfying assignment will generate a well-typed LJ program. Using the feature model to guide the SAT solver, we are able to type check all the members of a product line, guaranteeing safe composition for all programs described by that feature model.

9. REFERENCES

[1] D. Ancona and S. Drossopoulou. Polymorphic bytecode: Compositional compilation for java-like languages. In *In ACM Symp. on Principles of Programming Languages 2005*. ACM Press, 2005.

[2] S. Apel and D. Hutchins. An overview of the gDEEP calculus. Technical Report MIP-0712, Department of Informatics and Mathematics, University of Passau, November 2007.

[3] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In *GPCE '08: Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM Press, Oct. 2008.

[4] D. Batory. Feature-oriented programming and the AHEAD tool suite. *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 702–703, May 2004.

[5] D. Batory. Feature models, grammars, and propositional formulas. In *In Software Product Lines Conference, LNCS 3714*, pages 7–20. Springer, 2005.

[6] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, Berlin, 2004.

[7] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*. ACM Press, 2006.

[8] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[9] R. Prieto-Diaz and J. Neighbors. Module interconnection languages: A survey. Technical report, University of California at Irvine, August 1982. ICS Technical Report 189.

[10] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: effective tool support for the working semanticist. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 1–12, New York, NY, USA, 2007. ACM.

[11] R. Strnisa, P. Sewell, and M. J. Parkinson. The Java module system: core design and semantic definition. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA*, pages 499–514. ACM, 2007.

[12] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104, New York, NY, USA, 2007. ACM.

Graph-Based Specification and Simulation of Featherweight Java with Around Advice

Tom Staijen
Software Engineering Group
University of Twente
P.O. Box 217, 7500 AE
Enschede, The Netherlands
staijen@cs.utwente.nl

Arend Rensink
Formal Methods and Tools Group
University of Twente
P.O. Box 217, 7500 AE
Enschede, The Netherlands
rensink@cs.utwente.nl

ABSTRACT

In this paper we specify an operational run-time semantics of Assignment Featherweight Java — a minimal subset of Java with assignments — with around advice, using graph transformations. We introduce a notion of correctness of our specification with respect to an existing semantics and claim a number of advantages over traditional mathematical notations, that come forth from the executable nature of graph-transformation-based semantics.
Using test programs as graphs during specification of the semantics, simulation can help in verifying the correctness of the rules simply by testing, increasing the rigorousness of the specification process. Also, execution of the semantics results in a state space that can be used for analysis and verification, giving rise to an effective method for aspect program verification.
As a criterion for correctness, we use a structural operational semantics of this language from the so-called Common Aspect Semantics Base.

Categories and Subject Descriptors

F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—Operational semantics

General Terms

Languages, Verification

1. INTRODUCTION

Aspect-oriented programming (AOP) [6] is a popular paradigm that allows for the modular specification of cross-cutting concerns. However, aspect-oriented programs are not easy to get right and even harder to test or debug. For this reason, it is attractive to investigate formal verification for aspectual programs. For the purpose of formal verification of AOP languages and programs, it is essential to specify the semantics of such languages formally and unambiguously.

FOAL'09, March 2, 2009, Charlottesville, Virginia, USA.

In this paper we propose a formal specification approach for the run-time semantics of Assignment Featherweight Java (AFJ). We illustrate an instruction based representation of an AFJ program and define the semantics of a "machine" running these instructions. This language is an extension to Featherweight Java (FJ), a minimal subset of Java. This simple language - although not suitable for industrial implementations - is typically suitable for studying language extensions. We study an extension of this language with *around advice*, which can also be used to represent before and after advice.
The specification method proposed in this paper is graph transformations. Graph transformation is a formal specification technique that supports rule based specification as well as an intuitive visual representation of states and rules. As a matter of fact, we use Groove [8] for the definition of the rules and graphs. Using the GROOVE tool, the graph transformation-based operational semantics directly provides an executable model; given a start graph representing a program and the graph transformation based operational semantics of the language, this system can be simulated resulting in a state space, represented as a labelled transition system (LTS). We claim that graph transformation has the following main advantages over traditional, more mathematical notations of operational semantics.

- Specifying a semantics can be a complicated task; mistakes are easily made. The directly executable nature increases ease and confidence of specification of a semantics by giving the user a way to test the semantics without having to write a interpreter first, which may contain errors either copied from the semantics or made during implementation.
- By giving the semantics in this way, the road is opened towards applying existing verification methods, such as the work we have presented in [1]. Also, the LTS lends itself directly for model checking (see [5]).

In addition, we believe that the visual nature of the graph transformation rules will appeal to many readers that are not experts in mathematics.
To increase confidence in the correctness of our definitions, we show that they coincide with a formal specification of the AFJ language in a work called the Common Aspect Semantics Base (CASB) [2]. The CASB is presented as a reference model for the run-time semantics of aspect-oriented programming languages. It presents a structural operational

semantics (SOS) for the language at hand (AFJ with an aspectual extension).
In the next section we will explain details of Assignment Featherweight Java with around advice and give an impression of the used execution mechanism. Section 3 provides a background on graph transformations and the visual notation used in this paper, followed by brief intuition in Section 4 of the actual graphs and rules used in the graph transformation based semantics. In Section 5 we briefly discuss the reference semantics and formulate our notion of correctness. Finally, in Section 6 we briefly mention essential related work, followed by our conclusions in Section 7.

2. ASSIGNMENT FEATHERWEIGHT JAVA WITH AROUND ADVICE

In this paper, we specify an operational semantics of Assignment Featherweight Java extended with the possibility of declaring *around* advice. In this section, we describe the required background.

2.1 The Featherweight AspectJ Language

Featherweight Java (FJ) [3] is a subset of Java that contains only five forms of expression: object creation, method invocation, field access, casting, and variables. Assignment Featherweight Java (AFJ) [7] has extended this language with mutable field variables to bring it closer to the way Java programs are usually written. The minimal syntax and operational semantics make it a handy language for conceptual studies on the implications of language extensions. This makes the language useful for trying AOP language features. We actually study an extension of the AFJ language with around advice. For the duration of this paper, we will refer to the extended language as Featherweight AspectJ (FAJ). The grammar of FAJ is as folows:

$$
\begin{aligned}
Prog &::= \overline{L}; e; \overline{A} \\
L &::= \textbf{class}\, T\, \textbf{extends}\, T\, \{\overline{T\, f};\, \overline{M}\} \\
M &::= T\, m(\overline{T\, x})\{e;\} \\
e &::= x \mid e.f \mid e.m(\overline{e}) \mid \textbf{new}\, T(\overline{e}) \\
&\quad \mid e.f = e \mid (T)e \\
A &::= T\, \textbf{around}(\overline{T x})\ :\ P\, \{e'\} \\
P &::= \textbf{call}(T^*.m^*(\overline{T^*})) \\
e' &::= e \mid e.\textbf{proceed}(\overline{e}) \\
T^* &::= T \mid \textbf{*} \mid T+ \\
m^* &::= m \mid \textbf{*}
\end{aligned}
$$

Throughout the paper we use the overbar notation for lists. A program consists of a set of classes, a main expression, and a set of advice declarations. Classes contain a list of field names and types, and a list of methods. A method consists of a return type, an identifier, a list of arguments and a method body, which is an expression. Expressions can be (from left to right) variables (e.g. a method parameters), fields accesses, method invocations with a sequence of expressions as arguments, object creations with a sequence of expressions as parameter, castings, assignments, and proceed expressions (see below). Object creation is not handled by an explicit constructor. Instead, the ordered list of arguments is assigned to the ordered list of fields.
Aspects are represented as (global) declarations of an around advice and a point-cut. Advices, depicted in the grammar above by the letter A, are methods that can optionally contain a *proceed* expression. As usual, we can use this also to mimic *before* and *after* advice, by adding a `proceed` instruction after or before the instructions of the advice, respectively. An advice declaration is combined with a point-cut declaration (depicted by P in the grammar) that selects certain expressions. In this language we have limited the point-cut language to the selection of method calls. Such a point-cut is specified as a call to a certain receiver type T^*, which can be a concrete type, a wildcard "*" standing for an arbitrary type, or $T+$, selecting a type and all its subtypes. The same is used for the parameters of the call. The method identifier can be either a concrete method identifier, or a wildcard "*" selecting an arbitrary identifier.

$$
\begin{aligned}
mcode(id, r) &= S(mbody(id, r)); \textsc{return} \\
\\
S(x) &= \textsc{var}_x \\
S(e.f) &= S(e); \textsc{get}_f \\
S(e_0.f = e) &= S(e); S(e_0); \textsc{set}_f \\
S(\textsc{new}\, T(e_0, \ldots, e_n)) &= S(e_0); \ldots ; S(e_n); \textsc{new}_T \\
S(e.m(e_0, \ldots, e_n)) &= S(e_0); \ldots ; S(e_n); S(e); \textsc{call}_m \\
S(e.\textsc{proceed}(e_0, \ldots, e_n)) &= S(e_0); \ldots ; S(e_n); S(e); \textsc{proceed}
\end{aligned}
$$

Figure 1: Sequentialisation

2.2 Run-time Semantics

The run-time semantics of this language is specified in terms of sequences of instructions. That is, every expression in the grammar can be sequentialised into a sequence of stack-based instructions of the types: CALL, RETURN, NEW, VAR, GET, SET, and PROCEED. In this paper, we assume that expressions are pre-evaluated into such sequences; whenever we represent an FAJ program, method-bodies consist of a sequence of instructions instead of an expression. We define this sequentialisation as a function $mcode : Id \times T \rightarrow \overline{Instr}$ that returns a sequence of instructions given a method identifier. Given a function $mbody : Id \times T \rightarrow Expr$ that finds the body expression for a method identifier and a receiver type. This is defined as show in Figure 1.
Thus, the *mcode* function will use the given identifier and type to the *mbody* function, which looks up the method and returns the body expression. This expression is broken down into a sequence of instructions by function S. Finally, a RETURN instruction is added.
Run-time information is stored in both a heap and a number of global stacks:

- A so-called *continuation stack* contains the currently scheduled instructions, the top instruction being the first to be executed. Execution terminates when the continuation stack is empty.
- The results of evaluating an expression are placed on a so-called *value stack*.

For executing around advice, the following concepts are required:

- a *proceed stack* is used for postponing an action that triggers an advice; PROCEED instructions pop the top of the proceed stack onto the continuation stack.

Furthermore, a number of auxiliary instructions will be used:

- a DO instruction is used for invoking advices;
- a PUSHP instruction pushes the top of the continuation stack on top of the proceed stack;
- a POPP instruction pops the top of the proceed stack;

When a CALL instruction is matched by any aspects, these aspects are first scheduled (in a certain order) by placing a DO instruction on either the continuation stack (for the first advice), or the proceed stack (for any other advices). The PUSHP and POPP instructions are added to achieve a uniform handling of multiple around advice, all of which may contain a PROCEED instruction. To prevent instructions from being intercepted more then once, they are *tagged* the first time.

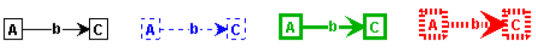
3. GRAPH TRANSFORMATIONS

Graph transformation is a systematic, rule-based transformation technique. It has a solid research foundation [9] and applications in many areas of computer science.
A graph is a type (N, E) where N is a set of nodes and $E \subseteq N \times Lab \times N$ a set of labelled edges. Nodes are graphically represented as black bordered boxes and edges as black arrows. A graph production system (GPS) is a set of *graph production rules*, each of which can transform a *source graph* into a new graph called the *target graph*. The rule specifies both the conditions under which it applies and the changes it makes to the source graph. Technically, a graph production rule consists of two partially overlapping graphs, a *left hand side* and a *right hand side*, and a set of *negative application conditions*, which are also (connected) graphs partially overlapping with the left hand side.
Graph transformations provide an attractive visual representation. In our visual representation of a rule used in this paper (which is taken from the GROOVE tool [8]) we combine all elements together in one graph, made up of four types of elements:

- *Readers*: elements that are used for matching; are depicted as with black borders and arrows (see Figure 2.a);
- *Erasers*: elements that will be erased during the transformation are depicted with thin dashed borders and arrows (see Figure 2.b);
- *Creators*: elements that will be created during transformation are depicted with thick light gray borders and arrows (see Figure 2.c);
- *Embargoes*: elements that are not allowed to be present in the graph when the rule is matched are depicted with dashed, dark gray edges (see Figure 2.d)

4. SEMANTICS

We specify the run-time semantics of FAJ using graph transformations. Due to the limited amount of space, we are merely able to give an intuition of the workings: the encoding of the graphs, and the actual rules that form the semantics.



(a) Reader (b) Eraser (c) Creator (d) Embargo

Figure 2: The graph production rule elements

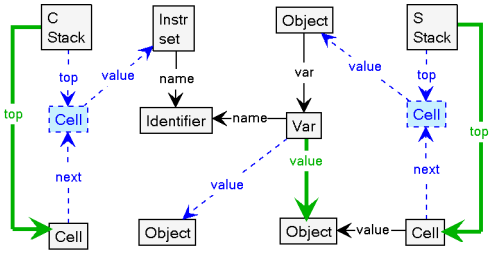


Figure 3: Rule for the set instruction

Graphs consists of a graph-based representation of an FAJ specification, and the run-time state. This run-time state consists of the stacks that we have introduced in Section 2, and nodes representing Objects. In fact, no distinction is made between memory locations and the actual instances. Expressions (method-bodies and the main expression) have already been process to sequences of instructions; the main expression is placed on the continuation stack.
There are rules for each of the instructions discussed in Section 2. To get a feeling of the proposed semantics, we describe the SET instruction. The complete set of rules can be found at *http://www.cs.utwente.nl/~staijen/faj/*.
Figure 3 shows the rule for the SET instruction, which originates from an expression of the kind $e.f = e_0$. The rule applies when a SET instruction is popped from the continuation stack, represented by a node labelled `Stack, C`. An outgoing edge of this stack points to the top `Cell` of the stack. A `next` edge points to the `Cell` underneath. The receiver object e and the new value e_0 are on the value stack, labelled with `Stack, S`. The variable to be updated is selected, that has the same name as the name argument of the SET instruction. The value of the variable is replaced, and the receiver is popped from the value stack. The new value remains on the value stack, since it is also result of $e.f = e_0$.

5. CORRECTNESS

We now give an intuition of how we can show that our operational semantics is *correct* in the sense that it corresponds to the prior semantics defined in SOS (Structural Operational Semantics) style in [2]. For this purpose, we define a mapping from the configurations in the SOS semantics to graphs, such that there is a one-to-one correspondence between SOS derivations and (sequences of) graph derivations. (It should be noted that our language is a slight adaptation of that in [2]; the most important difference is that we only allow call point-cuts, whereas they can define point-cuts for arbitrary instructions; on the other hand, we include parameters into the advice, which they do not. To establish correctness, we use an accordingly modified version of the SOS semantics.)

The static structure of a given FAJ program is captured by three partial functions:

- $FDecl : T \to (Ident \times T)^*$, yielding for each class type the sequence of field declarations (where *Ident* is the universe of identifiers);
- $MDecl : (T \times Ident) \rightharpoonup (Ident \times T)^* \times T \times Expr$, yielding for each class type the corresponding method declarations. A method declaration consists of a list of parameters (pairs of identifiers and types), a return type and a method body.
- $ADecl : (T \times Ident) \to \mathcal{P}((Ident \times T)^* \times Expr)$, yielding for each pair of class and method identifier the set of aspects that statically match calls of that method.

For every program, this triple of functions together with the initial expression plays exactly the same role as the initial graph. , except that there all expressions (i.e., the initial expression and the bodies of all methods and advices) have been sequentialised as discussed in Sect. 2.2. In fact, there is a straightforward translation from each valid combination $(FDecl, MDecl, ADecl, Expr)$ to an aspect program graph. For lack of space we omit the definition here, but below we use `Instr`, `Ident` and `Class` to denote the sets of nodes corresponding to *Instr*, *Ident* and T in the program.
The *dynamic* structure, i.e., the states of the program, are encoded in the SOS semantics as *configurations* (S, C, Σ, P) consisting of the same stacks and store as in our graph-based semantics:

- $C \in (Instr \times Bool)^*$ is the continuation stack, containing the instructions to be executed, combined with booleans indicating whether the instruction has already been adviced.;
- $S \in Object^*$ is the value stack, containing intermediate results; *Object* means location or heap address here;
- $\Sigma : Object \to T \times (Ident \rightharpoonup Object)$ is the heap, containing the run-time type and field values of all objects;
- $P \subseteq (Instr \times Bool)^*$ is the proceed stack, containing the (tagged or untagged) advices scheduled to be executed.

On the basis of these configurations, the SOS semantics consists of two types of rules: first, rules to sequentialise expressions to their corresponding instructions; and second, rules modelling the the execution of the instructions. In our semantics we have chosen to do sequentialisation as part of the pre-processing in Section 2.2; for the purpose of showing correctness in this section, we assume the same has happened on the SOS semantics side; that is, we assume that all expressions are already transformed into sequences of instructions.
The execution rule of instruction INSTR always has the form

$$\frac{\textit{side conditions}}{(\text{INSTR} : C, S, \Sigma, P) \to (C', S', \Sigma', P')}$$

meaning that, if the side conditions are fulfilled, a configuration in which the first instruction on the continuation stack is INSTR can perform a step, changing into the configuration on the right hand side. For instance, the rules for SET, CALL and RETURN are:

$$\text{SET}\quad \frac{\Sigma(v_0) = (T, F)}{(\text{SET}_f : C, v_0 : v : S, \Sigma, P) \to_b (C, v : S, \Sigma[v_0 \mapsto (T, F[f \mapsto v])], P)}$$

$$\text{CALL}\quad \frac{\Sigma(v_0) = (T, F) \quad MDecl(m, T) = ((x_1, \ldots, x_n), e)}{(\text{CALL}_m : C, v_0 : v_1 : \ldots : v_n : S, \Sigma, P) \to_b (e[x_1/v_1, \ldots, x_n/v_n], this/v_0 : C, S, \Sigma, P)}$$

$$\text{RETURN}\quad \frac{}{(\text{RETURN} : C, S, \Sigma, P) \to_b (C, S, \Sigma, P)}$$

Note that in these rules, the continuation stack elements are given as plain instructions rather than pairs of instructions and booleans; this is to indicate that we do not care about the instruction tags here.
The P-stack is only used for advice execution. Two example rules are given below: the AROUND-rule to schedule advice execution, and the rule for executing PROCEED. Notice that in these rules, we do care about tags.

$$\text{AROUND}\quad \frac{\Sigma(\mathbf{this}) = (T, Fd) \quad \Psi(T, m) = \{a_q \ldots a_n\}}{\begin{array}{l}((\text{CALL}_m, \mathbf{ff}) : C, S, \Sigma, P) \to \\ \quad (\text{DO}_{a_1} : \text{POPP} : C, S, \Sigma, \\ \quad\quad \text{DO}_{a_2} : \ldots : \text{DO}_{a_n} : (\text{CALL}_m, \mathbf{tt}) : P)\end{array}}$$

$$\text{PROCEED}\quad \frac{}{\begin{array}{l}(\text{PROCEED} : C, S, \Sigma, i : P) \\ \quad \to (i : \text{PUSHPP}_i : C, S, \Sigma, P)\end{array}}$$

5.1 From Configurations to Graphs

The translation of SOS configurations to graphs is defined by

$$Tra : (C, S, \Sigma, P) \mapsto [\![C]\!] \cup [\![S]\!] \cup [\![\Sigma]\!] \cup [\![P]\!]$$

where $[\![C]\!]$, $[\![P]\!]$ etc. are the graphs corresponding to the individual data structures; the combined graph is the union of these. The individual graphs in turn are defined as follows:

- For each of the stacks, we introduce a single special `Stack`-node that stands for the stack as a whole, and `Cell`-nodes that stand for the stack positions. As representatives we can use integer numbers:

  $\texttt{Stack} = \{-1\}$
  $\texttt{Cell} = \{0, \ldots, n\}$ where n is the stack size

  The nodes are linked with `top`-, `next`- and `value`-edges in accordance with the generated graphs. Using $|C|$ to denote the size of C and C^i to denote the value at position i (where the first position is numbered 0 and the last $|C| - 1$), the formal definition is:

  $$[\![C]\!] = (\texttt{Stack} \cup \texttt{Cell} \cup \texttt{Instr}, E_C) \quad \text{where}$$
  $$E_C = \{(-1, \texttt{top}, 0)\} \cup \{(i, \texttt{next}, i+1) \mid 0 \le i < |C|\} \cup \{(i, \texttt{value}, x) \mid 0 \le i < |C|, C^i = (x, b)\} \cup \{(i, \texttt{tag}, i) \mid 0 \le i < |C|, C^i = (x, \mathbf{tt})\}$$

  Note that stacks always contain a spurious `Cell`-node for the sake of uniformity, so that even the empty stack has a `top`-edge.

  The P-stack is encoded in the same way; so is the S-stack, except that the `value`-edges point to `Objects`, and no `tag`-edges are required.

- For the store, we assume a set of nodes `Object` corresponding to the objects in $dom(\Sigma)$, that is, those

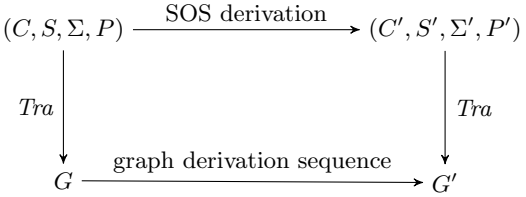
objects that are actually allocated on the heap. We also need auxiliary nodes to represent the object fields; these will be encoded as pairs (o, f) where $o \in \texttt{Object}$ and f is a field declared for o's type:

$$\texttt{Var} = \{(o, id) \mid \Sigma(o) = (t, Fd), id \in dom(Fd)\}$$

Using this set of nodes, the graph for Σ is defined by

$$\begin{aligned}[\![\Sigma]\!] = (\texttt{Class} \cup \texttt{Object} \cup \texttt{Ident} \cup \texttt{Var}, E_\Sigma) \quad \text{where}\\ E_\Sigma = \{(v, \texttt{name}, id) \mid v = (o, id)\} \cup\\ \{(o, \texttt{var}, v) \mid v = (o, id)\} \cup\\ \{(v, \texttt{value}, o) \mid v = (o', id), Fd(o') = o\} \cup\\ \{(o, \texttt{type}, t) \mid \Sigma(o) = (t, Fd)\}\end{aligned}$$

On the basis of these definitions, the correctness criterion is that the following correspondence must hold between the graph semantics and the SOS semantics:



This picture can be read top-down or bottom-up: for all single SOS derivations, there is a corresponding sequence of graph derivations, and for all sequences of graph derivations *between non-intermediate graphs* there is a corresponding SOS derivation — where a graph is intermediate if it is in between of a number of graph derivations that together correspond to the SOS derivation, i.e. when a SOS rule is specified using more then one graph transformation rule. The corresponding derivations are defined as a bisimulation. We are currently working on this final step.

6. RELATED WORK

The idea of the work reported here arose from [4], where a full graph transformation-based semantics is given for a custom defined object-oriented language. Also based on that work is the graph transformation-based semantics of a the Composition Filters language mentioned in [1], which however does not include a base language semantics and can therefore merely execute subsequent advices at a single join point. Both models use a different run-time state representation that is more suitable for object-oriented "machines". As mentioned before as our correctness criterion, Douence et al. [2] give an operational semantics of two base languages — a simple functional language and AFJ — and a large number of features of aspect-oriented language. The notation used is semi-formal yet mathematical and it does not provide means for execution.

7. CONCLUSION

In this paper we have propose a graph transformation-based semantics for a simple object-oriented language with around advice. The specified language, Assignment Featherweight Java, leans itself very well for studying language extensions. We have extended this language with around advice bound to point-cuts that select certain instructions.
We have illustrated that a graph transformation based operational semantics is a formal specification technique and can be complete with respect to a certain reference semantics. We have introduced a notion of correctness and illustrated how to prove this correctness. A graph tranformation based semantics is directly executable. This can help in finding bugs and testing the semantics. Due to its executable nature, the graph transformation-based specification has led to the discovery of oversights in the specification that is used as a correctness criterion; we see these errors as an unfortunate consequence of a purely formal notation.
The executable semantics allows simulation of program written in the specified language, if this program is represented as a graph as described in this paper. This gives a simple view on the execution of the program, and opens the road towards applying existing verification methods such as analysis based on model checking.

8. REFERENCES

[1] Mehmet Aksit, Arend Rensink, and Tom Staijen. A Graph-Transformation-Based Simulation Approach for Analysing Aspect Interference on Shared Join Points. In *AOSD '09: Proceedings of the 8th International Conference on Aspect-Oriented Software Development, Charlotteville, Virginia, USA*, Charlottesville, Virginia, USA, March 2009.

[2] Rémi Douence, Simplice Djoko Djoko, Pascal Fradet, and Didier Le Botlan. Towards a common aspect semantic base (casb). In *Deliverable 54, AOSD-Europe, EU Network of Excellence in AOSD*, August 2006.

[3] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[4] H. Kastenberg, A. G. Kleppe, and A. Rensink. Engineering object-oriented semantics using graph transformations. Technical Report TR-CTIT-06-12, University of Twente, Enschede, March 2006.

[5] H. Kastenberg and A. Rensink. Model checking dynamic states in groove. In A. Valmari, editor, *Model Checking Software (SPIN), Vienna, Austria*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305, Berlin, 2006. Springer-Verlag.

[6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[7] Thomas Mølhave and Lars H. Peterseny. Assignment featherweight java: Bringing mutable state to featherweight java. Master's thesis, University of Aarhus, 2005.

[8] Arend Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.

[9] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume I: Foundations. World Scientific, Singapore, 1997.