Modularizing Theorems for Software Product Lines: The Jbook Case Study

Don Batory
(University of Texas at Austin, USA
batory@cs.utexas.edu)

Egon Börger
(Università di Pisa, Italy
boerger@di.unipi.it)

Abstract: A goal of software product lines is the economical assembly of programs in a family of programs. In this paper, we explore how theorems about program properties may be integrated into feature-based development of software product lines. As a case study, we analyze an existing Java/JVM compilation correctness proof for defining, interpreting, compiling, and executing bytecode for the Java language. We show how features modularize program source, theorem statements and their proofs. By composing features, the source code, theorem statements and proofs for a program are assembled. The investigation in this paper reveals a striking similarity of the refinement concepts used in Abstract State Machines (ASM) based system development and Feature-Oriented Programming (FOP) of software product lines. We suggest to exploit this observation for a fruitful interaction of researchers in the two communities.

Keywords: ASM, features, composition, verification, AHEAD.

Categories: D.2.1, D.2.4, D.2.10, D.2.11.

1 Introduction

Product-lines are used in many industries to reduce product development costs, improve product quality, and increase product variability. The automotive, computer hardware, and software industries offer examples [BMW 2007][Dell 2007][Pohn 2005]. Sadly, what distinguishes software products is the absence of meaningful warranties [Java 2008]. While great strides have been made in verification over the last ten years, there are few results on verifying software product-lines (SPLs) [Blundell 2004][Krishnamurthi 2001][Krishnamurthi 2004][Thaker 2007].

Scaling verification to large programs is a long-standing problem. There is a growing community of researchers that believe verification must be intimately integrated with software design and modularity for scaling to occur; verification of programs should not be an after-thought [Hunt 2006][Xie 2003]. In this paper, we explore an approach that suggests how feature modularization may scale verification to product-lines of programs. We bring together results from previously unrelated communities: Abstract State Machines (ASM) based system development and Feature-Oriented Programming (FOP). The ASM method is a rigorous approach to step-wise program development and verification. FOP is a design methodology and compositional technology for customized program assembly. ASM and FOP both use step-wise refinement to construct programs and specifications. Although ASM and FOP were conceived independently (their roots trace back to the early 1990s), both have independently recog-
nized the value of features — increments in functionality — as a modularization centerpiece.

To explore the idea of assembling not only programs, but also theorems about program properties in a feature-oriented way, we use as a case study the 2001 Jbook [Stärk 2001] that among other results presented a Java/JVM compilation correctness proof for defining, interpreting, compiling, and executing bytecode for the Java 1.0 language. Among the pragmatic discoveries of Jbook were problems with bytecode verification, inconsistent treatment of recursive subroutines, method resolution and reachability definition, under-specification of static initializers (leading to portability problems of Java programs), concurrent initializations could deadlock, and existing Java compilers violated initialization semantics through standard optimization techniques [Börger 1999]. More recent work examined C# with similar results [Börger 2005][Fruja and Börger 2006][Fruja 2004][Fruja and Börger 2006].

Jbook and FOP both use features to modularize grammars and programs in an identical way. Using the Jbook case study we show how the modularization of Java programs can go in parallel with a feature-oriented description and verification of desired program properties. By composing features, complete grammars, programs, theorem statements and proofs can be assembled. To our knowledge, this is the first time that this has been shown. Thus, the first contribution of our paper is to document this claim for the Jbook case study. Our paper does not present a complete report on Jbook, which is presented elsewhere [Stärk 2001]. Rather, we explain that the Jbook illustrates all the characteristics of a verified SPL development using features. We can only illustrate each of these characteristics by giving concrete examples from the Jbook, and explaining what is needed for a fair understanding without assuming Jbook familiarity.

The second contribution of our paper is to make the two involved communities aware of the fact that the refinement concepts used in FOP of software product lines and ASM based system development are to a large extent the same. This leads us to discuss the generality of the proposed feature-based verification method, which combines ASMs and SPLs. Our approach is general: features provide a way to modularize all representations of programs, irrespective of the domain. We argue that our results are not limited to a verified language implementation and still less to the proven-correct compilation scheme of Java programs to JVM bytecode taken from the Jbook. What is important for a successful application of the method is to start from a precise definition of the application domain with well-understood features. We explain why we believe that our results are meaningful in the context of any SPL where each of its programs may have its own unique set of properties and requiring customized proofs. Instead of manually verifying individual programs, which is a laborious task, theorem statements and proofs may be assembled, like other program representations. Assembled proofs may then be certified manually or automatically using a proof checker.

Since we use FOP and ASMs, besides mentioning along the way what we need, we also give in Appendix I and Appendix II summaries of FOP and ASMs in an effort to make the paper self-contained for the reader who may not know both.
2 An Overview of the Jbook Product Line

Jbook [Stärk 2001] presents a structured way to incrementally develop the Java 1.0 grammar, its language interpreter, compiler, and bytecode (JVM) interpreter (including a bytecode verifier). The sublanguage of Java expressions is considered first, then it is progressively refined with the addition of Java statements, static class constructs, object constructs, and lastly support for exceptions. Each increment in functionality, here called a feature, builds upon previously defined functionalities by showing how the grammar, language interpreter, compiler, and bytecode interpreter are simultaneously and consistently refined. At the end of this horizontal refinement chain, a complete grammar, language interpreter, compiler, and bytecode interpreter for Java 1.0 are obtained.

Figure 1 shows the Jbook organization, what is called there the vertical refinement structure. Each oval represents a domain and each solid arrow denotes a tool that is a function that maps an object in its domain to an object in its codomain. The parser maps a Java program to an abstract syntax tree (AST). The interpreter maps an AST to an interpreter run or execution trace (InterpRun). The compiler maps an AST to bytecode. And the JVM interpreter, after having successfully run the bytecode verifier on the given bytecode, executes this bytecode to produce a JVM run.

At this point, various properties are considered, such as the correctness of the compiler. The dashed arrow in Figure 1 denotes the proof that interpreter runs are equivalent to JVM runs for the same Java program. Correctness is established by a mathematical proof of the equivalence of the interpreter execution of a Java 1.0 program and the JVM run (execution) of the compiled program.

Jbook was not developed with product lines in mind. It focused on the definition and verification of a single abstract interpreter and compiler scheme for the Java 1.0 language. To give Jbook an SPL architecture, we present a series of more elaborate FOP models that link the horizontal and the vertical refinement steps in a way that allows the theorems and proofs to be refined, in particular the compiler correctness theorem (i.e., its statement and proof).

We use for this purpose the GenVoca model of product-lines: base programs are values (0-ary functions) and features are unary functions that map programs to refined programs [Batory 1992]. A GenVoca model of Jbook is JB, where each element of JB is an

1. The set of instructions of the bytecode interpreter progressively grows with each additional feature. New instructions help execute a feature’s increment in functionality.
increment in Java language functionality, and different compositions of features yield
different variants of the Java language:

\[
\text{JB} = \{ \text{ExpI}, \quad \text{// imperative expressions} \\
\text{StmI}, \quad \text{// imperative statements} \\
\text{ExpC}, \quad \text{// static fields & expressions} \\
\text{StmC}, \quad \text{// method calls and returns} \\
\text{ExpO}, \quad \text{// object expressions} \\
\text{ExpE}, \quad \text{// expression exceptions} \\
\text{StmE}, \quad \text{// exception statements} \}
\]

\text{JB} has a single value \text{ExpI} which defines the Java sublanguage of imperative expressions. The remaining features are functions (refinements). \text{StmI} adds imperative statements; \text{ExpC} and \text{StmC} add static fields, static methods, and static initializers; \text{ExpO} adds object expressions; and \text{ExpE} and \text{StmE} add exceptions to expressions and exception handling statements. The version of Java that was verified is \text{Java1.0}, which composes all of these features:

\[
\text{Java1.0} = \text{StmE} \times \text{ExpE} \times \text{ExpO} \times \text{StmC} \times \text{ExpC} \times \text{StmI} \times \text{ExpI}
\]

where \(\times\) denotes function composition. That is, \text{Java1.0} was incrementally developed in the Jbook by starting from base \text{ExpI} (which defines the Java sublanguage of imperative expressions, an interpreter of this sublanguage, a compiler, etc.), then \text{StmI} refines it, then \text{ExpC} refines \text{StmI} \times \text{ExpI}, etc. In Jbook, only when the composition of \text{Java1.0} was complete was the correctness theorem (i.e. its statement and proof) developed. We show in the next section how theorem statements and correctness proofs can be assembled at each composition step.

Note: Figure 2 lists compositions of \text{JB} features that were given special names in the Jbook.

<table>
<thead>
<tr>
<th>Jbook Term</th>
<th>Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{JavaI}</td>
<td>\text{StmI} \times \text{ExpI}</td>
</tr>
<tr>
<td>\text{JavaC}</td>
<td>\text{StmC} \times \text{ExpC} \times \text{JavaI}</td>
</tr>
<tr>
<td>\text{JavaO}</td>
<td>\text{ExpO} \times \text{JavaC}</td>
</tr>
<tr>
<td>\text{JavaE}</td>
<td>\text{StmE} \times \text{ExpE} \times \text{JavaO}</td>
</tr>
<tr>
<td>\text{Java}</td>
<td>\text{Java1.0} [see (1)]</td>
</tr>
</tbody>
</table>

Figure 2: Jbook Compositions

Note Jbook treats Java expressions separately from statements. This separation allows one to use properties proved for expression evaluation as an inductive hypothesis when proving properties for statement execution.

To create a product-line, features can be omitted from \text{Java1.0} to produce sublanguages of Java. (A slightly different feature set than \text{JB} could be used to produce the Java Card language [Java 2008]. Another possibility is to add new language constructs: updating to Java 1.6, support for state machines [Batory 2004] and Lisp quote/unquote metaprogramming constructs [Taha 1997]. In either case, features can be mixed-andmatched, yielding a family or product-line of Java dialects and their tools (i.e., parser, interpreter, compiler). This is exactly how the language-extensible AHEAD tools were built [Batory 1998][Batory 2004].
3 AHEAD Representation of Jbook SPL

AHEAD is a generalization of GenVoca that exposes different representations of programs (source, grammars, documentation, makefiles, etc.) and reveals how features refine each of these representations by composition [Batory 2004]. We start with program representations of J extension features and consider theorems soon thereafter.

In this paper, we use shorter names for features and program representations than in the Jbook. Figure 3 lists correspondences of terms and their indices: term \( I \) with index \( \text{Exp}_I \) (i.e., \( \text{execJavaExp}_I \)).

<table>
<thead>
<tr>
<th>Our Term</th>
<th>Jbook Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G )</td>
<td>syntax</td>
<td>language grammar</td>
</tr>
<tr>
<td>( I )</td>
<td>execJava</td>
<td>language interpreter</td>
</tr>
<tr>
<td>( C )</td>
<td>compile</td>
<td>language compiler</td>
</tr>
<tr>
<td>( J )</td>
<td>trustfulVM</td>
<td>virtual machine</td>
</tr>
<tr>
<td>( T )</td>
<td>theorem</td>
<td>theorem of compiler correctness</td>
</tr>
</tbody>
</table>

Figure 3: Name Correspondences

3.1 Program Representations

Every program has multiple representations: source, documentation, bytecode, makefiles, etc. A GenVoca value is a tuple of representations for a base program, a notion we use now to represent the vertical refinement levels used in the Jbook. The representations of the J extension value \( \text{Exp}_I \) are: the grammar for Java imperative expressions \( G_{\text{Exp}_I} \), the ASM definition of the expression interpreter \( I_{\text{Exp}_I} \), the ASM definition of the expression compiler \( C_{\text{Exp}_I} \), the ASM definition of the bytecode (JVM) interpreter \( J_{\text{Exp}_I} \), and the verification (theorem) representation \( T_{\text{Exp}_I} \) which we will explain shortly. The tuple for program \( \text{Exp}_I \) is \([G_{\text{Exp}_I}, I_{\text{Exp}_I}, C_{\text{Exp}_I}, J_{\text{Exp}_I}, T_{\text{Exp}_I}]\).

A GenVoca function maps a tuple of program representations to a tuple of (in the Jbook called horizontally) refined representations. Feature \( \Delta \text{Stm}_I \) refines the base grammar by \( G_{\text{Stm}_I} \) (new rules for Java statements and tokens are added), the language interpreter by \( \Delta \text{Interp}_I \) (to implement the new statements), the compiler by \( \Delta \text{Comp}_I \) (to compile the new statements), etc. \( \text{Stm}_I \)'s tuple is \([\Delta \text{Stm}_I, \Delta \text{Interp}_I, \Delta \text{Comp}_I, \Delta \text{JVM}_I, \Delta \text{Th}_I]\).

The representations of a program are computed by tuple composition, where corresponding components are composed. The grammar, interpreter, compiler, etc. representations of the Java sublanguage \( \text{Java}_I \) that has imperative expressions and statements is:

\[
\text{Java}_I = \text{Stm}_I \times \text{Exp}_I \quad \text{// GenVoca expression}
= [\Delta \text{Stm}_I, \Delta \text{Interp}_I, \Delta \text{Comp}_I, \Delta \text{JVM}_I, \Delta \text{Th}_I] \times [G_{\text{Exp}_I}, I_{\text{Exp}_I}, C_{\text{Exp}_I}, J_{\text{Exp}_I}, T_{\text{Exp}_I}]
= [\Delta \text{Stm}_I \times G_{\text{Exp}_I}, \Delta \text{Stm}_I \times I_{\text{Exp}_I}, \Delta \text{Stm}_I \times C_{\text{Exp}_I}, \Delta \text{Stm}_I \times J_{\text{Exp}_I}, \Delta \text{Stm}_I \times T_{\text{Exp}_I}]
\]

That is, the grammar of the \( \text{Java}_I \) language is the base grammar composed with its refinement \( \Delta \text{Stm}_I \times G_{\text{Exp}_I} \), the ASM definition of the \( \text{Java}_I \) interpreter is the base defi-

1. In the Jbook also \( \text{compiler}_I \) has a modular structure, being composed out of a compiler \( E \) for expressions, \( S \) for statements, and \( B \) for flow-control expressions.
tion composed with its refinement ($\Delta_{\text{Stmt}} \cdot \Delta_{\text{Exp}}$), and so on. In general, the representations of a program are assembled by taking a GenVoca expression, replacing each term with its corresponding tuple, and composing tuples.

Note that features are not created in a haphazard way; they are carefully designed so that they (and their representations and refinements) are compatible, and their compositions yield the desired representations of the expected program. This design philosophy is present both in Jbook and AHEAD applications.

3.2 Theorems

Theorems proving program properties are another representation that is subject to refinement. The Jbook presents several theorems including the correctness of the Java compiler. We use this theorem, denoted by $T$, as a representative example.

$T_{\text{Exp}}$ denotes the theorem for the correctness of the $\text{Exp}$ compiler, i.e., the proof that interpreter runs of an $\text{Exp}$ program are equivalent to the JVM run of the compiled program. The refinement of this theorem by the $\text{Stmt}$ feature is denoted by $\Delta_{\text{Stmt}} \cdot T_{\text{Exp}}$ in tuple $\text{Stmt}$. The expression $\Delta_{\text{Stmt}} \cdot T_{\text{Exp}}$ assembles the correctness theorem for the Java language. Similarly for the languages $\text{JavaC}$, $\text{JavaO}$, $\text{JavaE}$, and $\text{Java}$. Before proceeding in section 5.3 with more details on theorem refinement we need to explain how AHEAD allows one to split program representations into subrepresentations and to define their refinements.

3.3 Nested Tuples

Program representations typically have subrepresentations, and recursively, subrepresentations may have subrepresentations. Hierarchical containment relationships are expressed by allowing each term of a tuple to be a tuple that can be refined. In general, the composition operator ($\cdot$) that we use recursively composes nested tuples. This is the essence of AHEAD [Batory 2004].

As an example, theorems have a tuple structure. Theorem $T$ has a statement $S$ and a proof $P$; $T$’s tuple is $[S, P]$. A theorem refinement $\Delta T$ may refine its statement ($\Delta S$) and/or its proof ($\Delta P$). A composite theorem is produced by composing its subrepresentations, i.e., $\Delta T \cdot T = [\Delta S \cdot S, \Delta P \cdot P]$. Examples of nested representations of code are given in [Batory 2004].

4 What is a Refinement?

A feature $F$ is a collection of transformations of the form $A \to A$ that maps an input artifact of a type $A$ to a modified (usually extended) artifact of the same type (e.g., source $\to$ source, grammars $\to$ grammars, etc.). These transformations are structure-preserving and monotonic in the following sense: new elements can be added to the input artifact and existing elements can be modified but not deleted.

Abstract state machines (ASMs) can be refined in several ways [Börger 2003] where AHEAD uses three. One is conservative extension: (a) define the condition for the new case, (b) define a new ASM to add the extra behavior, and (c) restrict the original machine by guarding it with the negation of the new case condition. Suppose the original machine is written in Java as method $m()$:
void m() {...}  // original machine

The structure of a method refinement in AHEAD that corresponds to a conservative
extension is:

```java
void m() {
    if (!condNew) SUPER.m(); // original actions
    else {...} // new actions
}
```

That is, if \( \text{condNew} \) (the condition of the new case) is not satisfied, invoke the
original method, which is denoted by \( \text{SUPER.m()} \). Otherwise execute the new actions. Ex-
ploting a technique that is well-known from logic one can prove a theorem for a con-
servative extension \( \Delta M \) of a machine \( M \) by first proving the theorem \( T = [S, P] \) for \( M \) and
then extending the statement \( S \) and proof \( P \) by what is needed to establish the theorem
\( \Delta T = [\Delta S, \Delta P] \). Typically this involves a case distin-
guion within an induction on runs of \( \Delta M \cdot M \), namely whether the considered step is an \( M \)-step, in which case the known
proof \( P \) for the statement \( S \) for \( M \) can be invoked, or a \( \Delta M \)-step, in which case the proof
extension \( \Delta P \) for the extended statement \( \Delta S \) is used. For a non-trivial example see the
extension of a trustful JVM interpreter by a bytecode verifier in Appendix V.

A second form of ASM refinement is parallel addition: (a) start with a given ASM,
typically guarded by some condition, (b) define a new ASM to add extra behavior, typ-
ically coming with the same guard of the original ASM. In effect, the example rule (b)
below is added “after” rule (a):

```java
if cond then update1  // ASM rule (a)
if cond then update2  // ASM rule (b)
```

By ASM semantics, both are executed simultaneously if \( \text{cond} \) is satisfied, effective-
ly extending the first rule to be:

```java
if cond then {update1; update2} // rules a + b
```

In AHEAD, the parallelism cannot be expressed directly. But one way to emulate it
is by the following refinement pattern:

```java
void m() { before; SUPER.m(); after; }
```

where either \( \text{before} \) or \( \text{after} \) could be null. Historically, a null \( \text{before} \) is called an af-
ter-method, a null \( \text{after} \) is a before-method, and non-null \( \text{before} \) and \( \text{after} \) actions are
an around-method [Kiczales 1991]. The naming relates to the fact that in AHEAD, the
semicolon expresses sequential execution. See Appendix III for one more case.

Sequential execution is one implementation of the parallel execution in ASMs; the
independence of the given and the new actions in ASMs is reflected in a semantically
correct way by a sequential execution only if different sequential orders produce seman-
tically equivalent executions.\(^1\)

The typical theorem refinement scheme for parallel addition is the conjunction,
where \( T = [S, P] \) represents the theorem for rule (a) and \( \Delta T = [\Delta S, \Delta P] = [Sb, Pb] \) the an-
glogue for rule (b).\(^2\)

---

1. The related issue of how to incorporate sequentiality into the parallelism of ASMs has been
addressed in [Börger 2000].
A third and by far most common form of refinement is adding new elements or equivalently, mapping a null artifact to a non-null artifact. Such refinements are called introductions. Adding new rules that apply to new states of execution is in the ASM framework a form of parallel addition and in fact is very common in Jbook. Introductions are also very common in AHEAD: a refinement of a class can add or introduce new members (fields, methods) and can modify existing methods (as indicated above). A refinement of a package can add or introduce new classes and refine existing classes. As we will see, the concepts of introduction and refinement apply to theorems as well.

5 Refinement of Artifacts

As we identify the feature-refinements of grammars, code, and theorems used in the Jbook, note the similarity of the underlying ASM design refinement to feature-based development. We will argue that one can incorporate into FOP the idea of feature-based refinement to verification as adopted in the Jbook, where it is used to a) rigorously define the complex program properties of interest there and b) to prove them.

5.1 Refining Grammars

The $G_{ExpI}$ grammar is shown below (in black font), where a Java imperative expression can be a literal, local variable, unary expression, binary expression, conditional expression, or expression assignment:

$$\text{Exp} ::= \text{Lit} \mid \text{Loc} \mid \text{Uop Exp} \mid \text{Exp Bop Exp}$$

$$\mid \text{Exp ? Exp} \mid \text{Asgn Field}$$

$$\mid \text{Class.Field Invk}$$

$$\text{Invk} ::= \text{Meth(Exps)} \mid \text{Class.Meth(Exps)}$$

$$\text{Exps} ::= (\text{Exp})^+$$

$$\text{Asgn} ::= \text{Loc = Exp Field = Exp}$$

$$\mid \text{Class.Field = Exp}$$

The $ExpC$ feature adds object fields and method calls by refining productions $Exp$ and $Asgn$ with additional right-hand sides, and introducing new productions $Invk$ and $Exps$ (indicated in italic font above). Similar extensions are used for $ExpC$, $ExpO$, $Stmt$, etc. features, covering the entire syntax of Java. Exactly the same technique was used in AHEAD to modularize\(^1\) and refine grammars [Batory 2004].

5.2 Refining Code

Although ASMs are rule-based, they can express object-oriented concepts of inheritance hierarchies and methods. Adding hierarchies and methods to programs is conceptually not very interesting, but refining them is. In the following, we present examples of Jbook refinements of both.

\[^2\] As we will see the composition operator ($\bullet$) for statements and proofs is logical conjunction:

$$\Delta \triangleright \Sigma = [Sb \land Sa, Pb \land Pa]$$

1. The grammar modules for various Java sublanguages can be viewed as partitioning of the entire grammar into independent and replaceable parts with precisely defined interfaces.
5.2.1 Refining Inheritance Hierarchies

Features progressively elaborate inheritance hierarchies by incrementally adding new subclasses. Figure 5 shows the kind of progressive elaboration used in the Jbook. The following explains the details.

Jbook calls expressions, statements and the result of executing an expression or statement a Phrase. ExpI defines a simple inheritance hierarchy rooted at Phrase (Figure 4). Val is a subclass of Phrase and it has many different subclasses (boolean, byte, short, etc.) which are depicted by a single class PrimValue.

Feature StmI adds imperative block statements subclasses to Phrase that represent the possible results of executing those imperative statements (e.g., Abrupton, Break, Continue, and Normal). Feature StmC adds to this the Return class; feature ExpO adds Reference and Null subclasses to Val, and feature ExpE adds Exception as a new subclass of Abrupton, where an abrupton is an interruption in flow control. Thus, the class hierarchy of Figure 4 is progressively revealed as features are composed. This is typical of FOP designs.

Figure 4: Refinement of Phrase Inheritance Hierarchy

5.2.2 Refining Methods

Besides adding new classes, features can refine existing classes. In particular, existing methods can be refined. The ASM concept of a “machine” or “submachine” closely resembles a Java method. For example feature StmC defines a submachine exitMethod for the actions (in black font) taken when a method is exited. It is refined by the special case of exiting a class initialization method (called clinit):

```java
exitMethod(result) = // ASM definition
    let (oldMeth, ...) = top(frames)
    ...
    if methNm(meth)="<clinit>" ^result="norm" then ...
    elseif methNm(meth)="<init>" ^ result="norm" then ...
    elseif ...
```

Feature ExpE adds constructor calls to the Java language. This requires exitMethod to be refined to handle the actions for returning from constructors. The refinement adds the definition in italic font above. This change can be easily expressed as a refinement of Java code.

1. ExpE adds exceptions to expressions. StmE adds throw-catch clauses to Java. Without the StmE feature, exceptions will be thrown by expressions and cannot be caught by a program.
5.3 Refining Theorems

Our example theorem $\tau$ defines the correctness of the Java compiler. The statement of $\tau$ for Java1.0 consists of thirteen invariants, nine are tersely described in Figure 5. A detailed knowledge of these invariants is not needed for this paper; it is sufficient to know that there are distinct named invariants. The next sections explain how the statement of $\tau$ (denoted by $\tau.S$) and its proof (denoted by $\tau.P$) are refined by features. Again, refinement means adding new elements (invariants, proof cases) and refining existing elements (invariants, proof cases). We show examples of each. Readers will note the similarity of theorem refinement with grammar and code refinement.

### 5.3.1 Adding New Invariants

$\tau.S$ is a conjunction of invariants, which is subject to incremental refinement by adding further conjuncts. The conjunction is represented here as a list. $\tau.S$ of the initial $\text{ExpI}$ sublanguage has the $(\text{reg})$, $(\text{begE})$ and $(\text{exp})$ invariants. The remaining invariants of Figure 5 are absent as they deal with abstractions (statements, abruptions, and class initializations) that cannot be defined using only $\text{ExpI}$ concepts.

The $\text{StmI}$ feature refines $\tau.S$ by conjunctively adding the invariants $(\text{begS})$, $(\text{stm})$ and $(\text{abr})$ which deal with the normal and abrupted termination of statement executions. The $(\text{stack})$, $(\text{clinit})$ and $(\text{exc})$ invariants are not included as they cannot be defined using $\text{ExpI}$ and $\text{StmI}$ concepts alone.

Similarly, the $\text{ExpC}$ feature adds as further conjuncts the $(\text{stack})$ and $(\text{clinit})$ invariants to $\tau.S$; the $\text{StmC}$ feature leaves $\tau.S$ unchanged.

Given these features, Figure 6 lists their compositions and the invariants of $\tau.S$ for each composition. The $\tau.S$ for composition $j_1$ has three invariants; the $\tau.S$ for $j_3$ and $j_4$ have eight. As Figure 6 shows, the set of invariants that define the statement of theorem $\tau$ in the JB product-line varies from program to program. The remaining JB features introduce the remaining invariants of $\tau$ for Java1.0.

<table>
<thead>
<tr>
<th>Invariant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\text{reg})$</td>
<td>the equivalence of local variables in the language interpreter and the associated registers in the JVM interpreter when both are in corresponding states</td>
</tr>
<tr>
<td>$(\text{begE})$</td>
<td>when the language interpreter begins to evaluate an expression, the JVM interpreter begins to execute the compiled code for that expression and the computed intermediate values are equivalent same as $(\text{begE})$ for a value returning termination of an expression evaluation</td>
</tr>
<tr>
<td>$(\text{exp})$</td>
<td>same as $(\text{begE})$ except it applies to statement execution</td>
</tr>
<tr>
<td>$(\text{begS})$</td>
<td>conditions for normal statement termination</td>
</tr>
<tr>
<td>$(\text{stm})$</td>
<td>conditions for abrupted statement execution</td>
</tr>
<tr>
<td>$(\text{abr})$</td>
<td>frame-stack equivalence condition</td>
</tr>
<tr>
<td>$(\text{clinit})$</td>
<td>class initialization status equivalence condition</td>
</tr>
<tr>
<td>$(\text{exc})$</td>
<td>conditions for exception statement execution</td>
</tr>
</tbody>
</table>

Figure 5: Invariants Used in Compiler Correctness Proofs
Note that part of the theorem statement refinement may also come through a grammar extension (if present). For example when new expressions are introduced by \( j_3 \), then the meaning of invariant \((\text{reg})\) ranges not only on the \( \text{ExpI} \) expressions, as in \( j_1 \), but also on the new expressions defined by the \( \text{ExpC} \) grammar. The same remark applies to the theorem statement refinement of \( \text{stm} \) properties.

<table>
<thead>
<tr>
<th>( j_1 )</th>
<th>( j_2 )</th>
<th>( j_3 )</th>
<th>( j_4 )</th>
<th>Java1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{reg} )</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>( \text{begE} )</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>( \text{exp} )</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>( \text{begS} )</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>( \text{stm} )</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>( \text{abr} )</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>( \text{stack} )</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>( \text{clinit} )</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>( \text{exc} )</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Figure 6: Statement of Correctness

5.3.2 Refining Existing Invariants

The program invariants themselves are also subject to refinement. As an example consider a sketch of the abruption \((\text{abr})\) invariant, which is as follows:

\[
\text{if restbody}_n/A=\text{abr} \text{ then } <\text{cond}_1> \quad (2)
\]

That is, \(<\text{cond}_1>\) must hold when an abruption occurs. In section 5.2.1 we saw that feature \( \text{ExpE} \) extends the definition of an abruption to include exceptions. The \(<\text{cond}_1>\) of (2) applies only to abruptions that are not exceptions.

\( \text{ExpE} \) uses a conservative extension to express this change. First, \( \text{ExpE} \) refines invariant (2) by adding the qualifying condition that the abruption is not an exception (below in italics). (2) becomes:

\[
\text{if restbody}_n/A=\text{abr} \text{ and abr is not an exception} \text{ then } <\text{cond}_1> \quad (3)
\]

Second, \( \text{ExpE} \) introduces two new invariants to \( \text{T.S} \) to cover the cases where an abruption is an exception \((\text{exc})\) and an exception is thrown during class initialization \((\text{exc-clinit})\). Both invariants have the following form:

\[
\text{if restbody}_n/A=\text{abr} \text{ and abr is an exception ... } \text{ then } <\text{cond}_2> \quad (4)
\]

In general, each member of the Jbook product line has a theorem statement. As features are composed, the theorem statement of what it means to be a correct compiler is refined by the addition of new invariants and the refinement of existing invariants.

5.3.3 Adding Proof Cases

Let \( T.S_{(c)} \) denote the conjunction of the invariants making up the theorem statement \( \text{T.S} \) for feature composition \( c \). If \( a \) is another feature, \( a+c \) must be shown to satisfy \( T.S_{(a+c)} \) — the (conjunction of the) invariants collected and refined by \( a+c \).
The structure of the compiler correctness proof \( T.P \) in Jbook is a list of cases appearing in an induction on the interpreter runs, i.e., in the initial state and each time an interpreter step has been performed. Feature \( G \) refines \( T.P_C \) by adding more cases and/or refining existing cases. Below we examine the refinements of \( T.P \) that are made by each of the \( \text{ExpI}, \text{StmI}, \text{ExpC} \) and \( \text{StmC} \) features.

The \( \text{ExpI} \) feature defines the imperative expressions of Java. Recall the recursive \( \text{ExpI} \) grammar definition:

\[
\text{Exp} \quad ::= \quad \text{Lit} \mid \text{Loc} \mid \text{Uop Exp} \mid \text{Exp Bop Exp} \\
\quad \mid \quad \text{Exp ? Exp : Exp} \mid \text{Asgn} \\
\text{Asgn} \quad ::= \quad \text{Loc} = \text{Exp}
\]

The proof for \( T \) is a case analysis using structural induction on the definition of expressions and of their compilation. The invariants \( \text{(reg)} \), \( \text{(begE)} \) and \( \text{(exp)} \) relate certain items (e.g., local Java variables and JVM registers) in the Java and JVM interpreters for \( \text{ExpI} \). If these invariants hold, the Java and JVM interpreter executions produce equivalent evaluation results. The proof \( T.P \) for \( \text{ExpI} \) is a list of 12 proof cases, one or more cases for each kind of expression showing the \( \text{ExpI} \) invariants are preserved [Stärk 2001].

The \( \text{StmI} \) feature introduces the imperative statements of Java. Its grammar refinement adds productions for Java statements; no \( \text{ExpI} \) productions are refined:

\[
\text{Stm} : = \quad ; \mid \text{Loc} = \text{Exp} ; \mid \text{Lab} : \text{Stm} ; \\
\quad \mid \text{break Lab} ; \mid \text{continue Lab} ; \\
\quad \mid \text{if (Exp) Stm else Stm} \\
\quad \mid \text{while (Exp) Stm} \mid \text{Block}
\]

(5)

The invariants that \( \text{StmI} \) adds are about statement executions, while the invariants of \( \text{ExpI} \) are about expression evaluations. Execution steps of the \( \text{ExpI} \) interpreter trivially preserve the \( \text{StmI} \) invariants, and vice versa, as these invariants relate sets of items that are disjoint. For the composed interpreters to satisfy the invariants of \( \text{StmI} \bullet \text{ExpI} \), \( \text{StmI} \) must add cases to \( T.P \), one or more for each production in (5), that prove the invariants of \( \text{StmI} \) are preserved; see cases 14-35 in [Stärk 2001]. Note that this induction on statement uses the proofs for the statement subexpression invariants as induction hypothesis.

The grammar refinement of feature \( \text{ExpC} \) adds expressions for static class fields, assignments to them, expression sequences and method invocations. The interpreter refinement of \( \text{ExpC} \) introduces frames and a frame stack and their values are related by the new invariant \( \text{(stack)} \). The second new invariant \( \text{(clinit)} \) relates the class initialization status of Java and JVM interpreter runs. As no \( \text{ExpI} \) and \( \text{StmI} \) interpreter step references or updates frames or the class initialization status, invariants \( \text{(stack)} \) and \( \text{(clinit)} \) are trivially satisfied by them. \( \text{ExpC} \) refines \( T.P \) with additional cases proving these two new invariants hold, one case for each kind of new expression. Since no new

---

1. In proof arguments, we tacitly use the fact that an ASM execution step is given by a set of guarded multiple assignments, in each step only the values of those locations (i.e., variables) that occur in a rule with a true guard may change, whereas the rest of the state remains unchanged. Thus, each time a new feature is introduced that adds a new invariant, that invariant is trivially preserved by each execution step that does not affect a location (variable) of the new invariant.
ExpC execution step affects any of the previous invariants, all the invariants hold for ExpC•StmI•ExpI: see cases 36-44 in [Stärk 2001].

StmC follows the above pattern: it adds no new invariants and refines T.P with additional proof cases, one or more for each production in its grammar refinement that adds method calls and returns.

Figure 7 lists compositions of features and the number of cases in T.P per composition. Each feature adds new cases (or refines existing ones, see below) in the proof of T.

<table>
<thead>
<tr>
<th>Composition</th>
<th>total # of cases in Proof of Theorem T</th>
</tr>
</thead>
<tbody>
<tr>
<td>j1 = ExpI</td>
<td>13</td>
</tr>
<tr>
<td>j2 = StmI•ExpI</td>
<td>35</td>
</tr>
<tr>
<td>j3 = ExpC•StmI•ExpI</td>
<td>44</td>
</tr>
<tr>
<td>j4 = StmC•ExpC•StmI•ExpI</td>
<td>54</td>
</tr>
<tr>
<td>Java1.0</td>
<td>83</td>
</tr>
</tbody>
</table>

Figure 7: Proof of Correctness

Cases can also be added as a result of refining invariants. In section 5.3.2, we showed the ExpE feature refined the abruption invariant (abr). As the existing proof cases for (abr) are not exceptions, their correctness remains unaffected by this refinement. However, ExpE adds new proof cases for the new invariants (exc) and (exc-clinit) which express the desired property for exceptions.

5.4 Refining Existing Proof Cases

Proof cases are also subject to refinement. The ExpI feature defines a case (in black font below) that shows the (exp) and (reg) invariant, which were introduced by ExpI, holds [Stärk 2001], namely (exp) for an expression evaluation and (reg) for the current values of Java local variables and the associated JVM registers:

Case 9: context(posn) = \( ^\alpha (\text{loc} = ^\beta \text{val}) \) and posn=\( \beta \):

Assume ... Hence invariant (exp) is satisfied in state n+1... and invariant (reg) is satisfied as well.

The invariant (fin) remains true since...

The StmE feature introduces try-catch-finally statements and a new invariant (fin) that deals with return addresses from finally code. As the return addresses from finally code are stored by the JVM in dedicated registers (not registers used by ExpI), it has also to be checked that every register assignment preserves this invariant (fin). Therefore the proof case concerning the values in JVM registers reg must be refined with additional proof text to show that the return addresses stored in reg for finally code are correct (in italic font above).

A larger example of theorem refinement that includes the addition and extension of both invariants and proof cases is presented in Appendix IV.

5.5 Further Structure

The ASM interpreter for Java programs and the compiler use a familiar object-oriented structure for implementing grammars. Each left-hand side of a production corresponds
to an abstract class and each right-hand side corresponds to one of its subclasses. The
inheritance hierarchy for expressions is rooted at abstract class called \texttt{Exp (expression)},
and it has concrete subclasses for literals (\texttt{Lit}), variables (\texttt{Loc}), unary expressions
(\texttt{Uop}), etc. Instances of these classes define an AST for a parsed expression [Batory
1998].

The Java interpreter defines an abstract method \texttt{interpret()} in the \texttt{Exp} class, and
all subclasses are obliged to provide implementations of this method (to interpret an
expression). Similarly, a compiler defines an abstract method \texttt{compile()} in the \texttt{Exp} class,
and all subclasses must provide implementations of this method (to compile an expres-
sion). Type checking ensures the methods are present in subclasses.

The proof of \( T \) in the Jbook is a sequence of cases within an induction on interpreter
runs. These cases largely correspond to the following: an ‘abstract’ theorem is defined
in the \texttt{Exp} class; all subclasses are obliged to define a concrete (i.e., fully elaborated)
theorem for each subclass. The ‘abstract’ theorem defines the invariants that are to hold
for all expressions. The ‘concrete’ theorems provide the proofs that these invariants
hold for particular expression types. This is the essence of structural induction (which
was used in the proof of \( T \)). In creating the original proof of the Jbook, some cases were
initially missed (and subsequently discovered). Type-checking would have automati-
cally reported the absence of missing cases. We will see in section 6 that type checking
might play a more expansive role in certifying theorems.

6 Generality of the Approach

An obvious question is: why does this work? We found in the Jbook an explicit repre-
sentation of a feature-based compositional verification structure. In a sense it did not
come as a surprise, since the major driving force for developing ASM models by step-
wise refinement had been “splitting the overall definition and verification problem into
a series of tractable subproblems” ([Stärk 2001] p7) for a complete (not some light-
weight) version of Java/JVM. The question remains whether we found an explicit fea-
ture-based formulation and proof of properties of interest because of the special charac-
ter of the domain of language compilation. We explain in this section the reasons which
make us believe that it is a general phenomenon that a clear compositional design struc-
ture goes together with a feasible structure of system invariants and their proofs.

Ideally, features only add new elements (e.g., ASMs, methods, classes, proofs). But
generally, this is not common. More typically, features add new elements and extend
existing elements, as we have seen in all the program representations used in the Jbook.
Such features have incremental (also called monotonic) semantics. As an aside, in twen-
ty years of building GenVoca product-lines, virtually all the features we have encoun-
tered have incremental semantics.

But there are domains where features have a more invasive impact by erasing the
definitions of existing elements (methods, ASMs, proofs) and replacing them with def-
initions that are specific to a composition of two or more features. That is, the replaced
definitions cannot be incrementally built. This is known as feature interaction: it is usu-
ally accompanied by an abrupt discontinuity in semantics where prior properties are no
longer valid. The telecommunications domain is replete with examples [Calder 2003].
Appendix III shows how element definitions can be replaced. [Liu 2006] is a general way to express feature interactions in a GenVoca model.

What can be said in full generality is the following: assume we are in a well-defined application domain where the domain expert knows well the relevant features. Then artifacts representing domain problem solutions, typically some code, as well as the statement and justification of properties of such problem solutions (read: invariants and proofs) typically have some structure. And within a structure, there are extension points or variation points where more structure can be added or existing structure can be replaced. Features exploit structure variability in that they modularize the structural changes of all program representations [Batory 1992][Batory 1998][Batory 2004][Kästner 2007].

This leads to the general question of mechanized support for verified developments. In many real-life software engineering problems a justification of the development (read: a proof of a certain system behavior) does not come in an automatic form, but builds upon the understanding of the subject matter by the engineer, as in Jbook\(^1\) and thus in this paper. In such an endeavor, ASMs help engineers formalize their programs and prove needed properties. The effort needed for manual proofs to convince humans is many times less than for comparable automated proofs,\(^2\) But manual certification of assembled theorems is only a provisional solution. The need for mechanized proofs for ASMs has been both recognized and accomplished for various case studies using theorem provers [Gargantini 2000][Goerigk 1996][Schellhorn 1998][Schellhorn 1998][Schellhorn 2007]. In particular, Schellhorn has shown that developing proofs incrementally (much like the incremental development in Jbook) simplifies the task of mechanically proving program properties [Schellhorn 1998].

Further progress for constructing feature-modularizing proofs may be made by examining what is assembled when features are composed:

- the text of a program’s source. This text must be compiled by a tool such as javac to verify that it is both syntactically correct and type correct.
- the text of a grammar’s specification. This text must be compiled by a tool such as javacc to verify it is both syntactically correct and well-formed (i.e., it type-checks according to the meta-grammar).
- the text of the program’s theorems. Here is where we need help: how do we know that the text constitutes a correct proof? If all program representations are treated similarly, what is the theorem counterpart to syntax and type checking?

Once the theorems are expressed in a machine manageable form, proof-checkers might be used. Theorems are written in a designated logic; a proof-checker certifies that the proof statements are well-formed in that logic. In effect, proof checking reduces to the type checking of terms that define the logic’s syntax, judgements, and rule schemes

---

1. As far as we know nobody in the theorem proving community up to now has accepted the challenge to mechanically verify the theorems proved in Jbook; much work has been done for restricted sublanguages, but not for the entire Java 1.0 (or the present Java) language. Therefore also the Jbook case study presented in this paper cannot (yet) be verified mechanically.
2. The reported ratio for two verification efforts was 1 to 4.
[Appel 2003]. So verifying a program of a product-line may be accomplished by assembling the program and its theorems, and using a proof checker to certify theorems automatically. Currently this is difficult to do: we cannot rely on different versions of a theorem prover to produce the same proofs (as different proofs may result as a consequence of different search strategies being used). Ideally, the incremental changes to proofs should not be dependent on particular proof technologies. In the case of Jbook, this seems to be the case: that a feature-based design of both code and proofs can lend itself to extensions that are straightforward to implement. Clearly, more examples like Jbook are needed.

Our approach is similar to verifying that the assembled source of a program is type-correct, i.e., assemble the program’s source and to see if it compiles without errors. Recent work shows how type safety properties of all programs of a product-line can be verified using SAT solvers [Thaker 2007]. This analysis may apply to proof text as well. Proof trees may have holes that are filled by refinements, e.g. when replacing an abstraction by a detailed machine, for which the axiomatic assumptions made for the abstraction have to be proved. Another example is the introduction of a sub-induction (e.g. on expressions) in a head induction (e.g. on statements). These ‘holes’ can be instantiated only by proof trees of a certain ‘shape’ (i.e., a theorem type). Guaranteeing that the ‘holes’ are instantiated properly is a problem of type-correctness.

7 Related Work

There is an enormous literature on verification. We limit our discussion of related work to that relevant to verifying software product lines.

The verification of product-lines using features using model-checkers was first studied by Krishnamurthi and Fisler in a series of papers (e.g., [Krishnamurthi 2001][Krishnamurthi 2004][Blundell 2004]). Properties of systems are often properties of individual features. They noted relationships between feature verification and open system verification, where information needed for system verification must be supplied by the set of features that define the target program. They developed specific algorithms and tools for computing propositional ‘interfaces’ of individual features, and for testing whether features violate system-wide properties. Our emphasis is on the feature-modularization of proofs for subsequent assembly, rather than techniques for automated formal verification.

Not all features are compatible; some features preclude or require the use of others in a composition. Verifying compositions of features is discussed in [Batory 2005], where feature models, grammars, and propositional formulas are related, and techniques for validating feature models are discussed.

Czarnecki used [Batory 2005] to show how feature models could be used to check the well-formedness of all products of a product line [Czarnecki 2006]. [Thaker 2007] is a follow-on work that showed how to verify the type correctness of a product-line.

Grammes and Gotzhein have studied a problem related to a product-line of SDL dialects [Grammes 2007]. A profile is a restriction of SDL to some sublanguage. Given an ASM interpreter for the semantics of full SDL and a profile, they can compute the interpreter for that profile and verify its correctness. (This computation may be charac-
terized by the removal of features). The verification step is done manually, but is believed that it is possible to be done automatically.

Hoare, Misra, and Shankar proposed a Verified Software Grand Challenge in 2005 with the goal of scaling verification to a million lines of code [Hoare 2005][Jones 2006]. Verification would be based on the text of a program and the annotations contained within it [Levens 2006]. We, like others, believe that verification must be intimately integrated with software design and modularity [Hunt 2006][Xie 2003]. Verifying product-lines, which requires the integration of design and verification, seems a fitting goal for a Verified Software Grand Challenge.

8 Conclusions

Providing warranties about programs is a long-standing goal of Computer Science. A pragmatic extension of this goal is to provide warranties for programs in a software product line. We make steps toward this extended goal by showing how features can integrate program verification and program design. A feature encapsulates program fragments that implement the feature’s functionality, as well as theorem fragments that prove the correctness of the feature’s behavior. Composing features yields both complete programs and their theorems. Our use of Jbook as a case study illustrates the feasibility of our approach.

Another contribution of this paper is the reinforcement that features offer a fundamental way to modularize programs. Disjoint communities (ASM and FOP) have independently recognized the utility of features to define complex programs in an incremental manner. We used the ASM Jbook case study to illustrate the power of features and to add theorems to the growing set of representations that are feature-refinable.

Although our work is preliminary, it provides us with new insights on feature-based verification. The next steps are to (a) evaluate the practicality of refining theorems, (b) certify assembled theorems by proof-checkers, and (c) see if certification can scale to all programs in a product line by exploiting recent advances in SPL verification. Further, the similarity of refinements of different program representations reinforces the possibility that general tools can be developed for refining all program representations, rather than developing unique tools to accomplish the same goals for different representations [Batory 2004].

Acknowledgements

We thank the K. Fisler, S. Krishnamurthi, C. Lengauer, H. Li, S. Trujillo and the anonymous referees for their helpful comments. This work was supported by NSF’s Science of Design Projects #CCF-0438786 and #CCF-0724979.

References


[BMW 2007] BMW: [www.bmwusa.com](http://www.bmwusa.com)


[Dell 2007] Dell Computers: [www.dell.com](http://www.dell.com)


Appendix I: Basics of FOP

A feature is an increment in program functionality. A software product line is a family of programs where no two programs have the same combination of features. Every program in a product line has multiple representations (e.g., source, documentation). When a feature is added to a program, any or all of the program’s representations may change. Below we informally sketch the first two generations of FOP, GenVoca and AHEAD, which have been used to build product-lines in many applications areas (e.g. [Batory 1992][Batory 2004]).

**GenVoca.** A GenVoca model of an SPL represents base programs as values (0-ary functions):

\[
\begin{align*}
  f & \quad \text{// base program with feature } f \\
  h & \quad \text{// base program with feature } h
\end{align*}
\]

Features are unary functions:

\[
\begin{align*}
  i*x & \quad \text{// adds feature } i \text{ to program } x \\
  j*x & \quad \text{// adds feature } j \text{ to program } x
\end{align*}
\]

where the operator \( \cdot \) denotes function composition.

The design of a program is a named expression:

\[
\begin{align*}
  p_1 = j*f & \quad \text{// } p_1 \text{ has features } j \text{ and } f \\
  p_2 = j*h & \quad \text{// } p_2 \text{ has features } j \text{ and } h \\
  p_3 = i*j*f & \quad \text{// } p_3 \text{ has features } i,j,f
\end{align*}
\]
The set of programs that can be defined by a GenVoca model is its product line. Expression optimization is program design optimization, and expression evaluation is program synthesis [Batory 2004][Batory 2005].

**AHEAD.** AHEAD generalizes GenVoca by revealing the internal structure of values and unary functions as tuples and modifications to tuples. Every program has multiple representations, such as source, documentation, bytecode, and makefiles. A GenVoca value is a tuple of representations of a program. For example, in a product line of parsers, a base parser \( f \) is defined by its grammar \( g_f \), Java source \( s_f \), and documentation \( d_f \). Program \( f \)'s tuple is \([g_f, s_f, d_f]\).

A GenVoca unary function maps a tuple of program representations to a tuple of extended representations using deltas. Suppose feature \( j \) extends a grammar by \( \Delta g_j \) (new rules and tokens are added), extends source code by \( \Delta s_j \) (new classes and members are added and existing methods are modified), and extends documentation by \( \Delta d_j \). The tuple of deltas for feature \( j \) is \([\Delta g_j, \Delta s_j, \Delta d_j]\), which we call a delta tuple.

The representations of a program are computed by tuple composition. The representations for parser \( p_1 \), which is produced by composing features \( j \) and \( f \), are:

\[
\begin{align*}
p_1 &= j \circ f; \quad \text{GenVoca expression} \\
&= [\Delta g_j, \Delta s_j, \Delta d_j] \circ [g_f, s_f, d_f]; \quad \text{substitution} \\
&= [\Delta g_j \times g_f, \Delta s_j \times s_f, \Delta d_j \times d_f]; \quad \text{composition}
\end{align*}
\]

That is, the grammar of \( p_1 \) is the base grammar composed with its extension \((\Delta g_j \times g_f)\), the source of \( p_1 \) is the base source composed with its extension \((\Delta s_j \times s_f)\), and so on.

Representations can have sub-representations, recursively. Every sub-representation can be modeled as a tuple, and can be transformed by delta tuples. In general, GenVoca values are nested tuples and functions are nested delta tuples, where the \( \circ \) operator recursively composes nested tuples. This is the essence of AHEAD [Batory 2004].

**Appendix II: Basics of ASMs**

*Abstract State Machines* provide a way to mathematically define the intuitive understanding of pseudo-code, extending Finite State Machines by “instructions” which operate on arbitrary structures.

**Machine Concept.** A (basic) ASM is a set of transition rules of form

\[
\text{If Condition then Updates}
\]

where the guard *Condition* is a first-order expression, denoting typically an event that has happened and/or a state of affairs that holds currently, and *Updates* is a set of array variable assignments (also called function updates) \( f(\exp_1, \ldots, \exp_n) := \exp \) with arbitrary expressions \( \exp_i, \exp \). In each step of such an ASM, all its transitions that can be fired (read: whose *Condition* is true in the current state) are executed simultaneously, changing as indicated by the *Updates* some array variable values (and only those), thus producing the next state (if the updates are consistent).

Also quantified rules of the following form are allowed, whose meaning, supporting synchronous parallelism and choice, should be obvious:

\[
\begin{align*}
&\text{Forall } x \text{ with } \text{Cond}(x) \text{ do rule}(x) \\
&\text{Choose } x \text{ with } \text{Cond}(x) \text{ do rule}(x)
\end{align*}
\]
The level of abstraction of an ASM (read: the structure upon which the machine rules operate) is determined by the functions that compose the expressions. These functions can be static or dynamic, the dynamic ones can be defined explicitly (so-called derived functions) or by the environment (so-called monitored functions) or by updates of the machine itself (so-called controlled functions), or they can be shared by the machine and (other agents in) its environment.

When dealing with multi-threaded distributed computations the notion of basic ASMs and the single computation steps performed by them does not change; what changes is the notion of runs. It is extended from runs where all the steps are ordered to asynchronous (formally: partial order) runs.

**Refinement Concept.** When refining an ASM, both its state (read: data structures) and its rules (read: computation steps) can be refined in combination. When relating abstract and refined runs, typically for stating and proving correctness, completeness and similar properties for the refinement, one has the freedom to define five features: a) the data structure refinement one wants to introduce, b) appropriate pairs of corresponding abstract and refined states of interest one wants to relate, c) segments of abstract and refined computation steps leading from one pair of corresponding states of interest to another one, d) sets of abstract and refined locations of interest one wants to compare, e) the equivalence properties one wants to establish.

For more detailed explanations we refer the reader to the (introductory chapter of the) ASM book [Börger and Stärk 2003].

**Appendix III: Replacement Refinements**

There is an additional refinement possibility in AHEAD: calls to \texttt{SUPER.m()} may be conditional. Consider the following refinement pattern:

\[
\text{void m() \{before; if (cond) SUPER.m(); after\}}\]

which is a blend of parallel addition and conservative extension. A special case of (6) that arises infrequently is when \texttt{cond} is always \texttt{false} (i.e., the original method is never called). This refinement is called replacement. The simplest known counter-example deals with element deletion in data structures. The element removal operation is:

\[
\text{void remove() \{ \_ \_ remove current element \_ \_\}}\]

When the feature of logical deletion is added to a data structure, elements are simply flagged deleted and are never removed. The logical deletion refinement of \texttt{remove()} is:

\[
\text{void remove() \{ set delete flag of element;}
\text{if (false) \{ \_ \_ remove current element \_ \_\}}\]

which is a replacement as the original method is not called. The logical deletion feature is not semantically equivalent to the original \texttt{remove()} method, though from an abstract viewpoint, by defining deletion to be equivalent with setting the deletion flag, an equivalence relation can be established between the two features.
Appendix IV: Complex Theorem Refinement

A feature can refine a theorem (statement and/or proof), namely by adding (Add) new and refining (Ref) existing invariants (Inv) and proof cases (Prf). We present an example that illustrates all of these possibilities.

Recall that an abruption is an interruption in flow control. Consider an abruption that is not an exception, say due to a return statement (which is similar to the remaining cases of a break or continue statement). If it occurs within a try block of a try-catch-finally statement and the corresponding target statement contains some try-catch-finally statement, then the Java semantics requires that all finally blocks between the return statement and its target have to be executed in innermost order before returning. To verify that this is correctly realized by the appropriately refined compiler (Fig.12.3 p164, which refines Fig.10.3 p153 and Fig.9.4 p144 in [Stärk 2001], feature StmE introduces a new invariant (fin) which states the correctness condition for return addresses from finally code that has to be executed when an abruption is encountered.

(fin) is a new condition for the newly introduced exceptions and try-catch-finally statements and thus is added to the list of invariants (Add-Inv). This triggers also a new proof part requiring new cases (Add-Prf), which are added to the existing proof (Jbook cases #76-80 for finally statements p199-201).

But the new invariant also refines the invariant that had already been imposed by the previously introduced features on abruptions that are not exceptions. In fact (fin) contains a refinement of the invariant for return statements (Ref-Inv) expressing that the correctness of the return address is preserved during the corresponding run segments for the finally code in the two interpreters executing the return statement. This triggers also a refinement of the proof cases (Ref-Prf) for return statements to guarantee that (fin) holds, namely adding the (fin)-related part to the original cases #48 (p191), #52 (top of p193), #53 (p193). Note that this refinement can be viewed as a conservative extension of the case of a return statement without finally code, because in the latter case the invariant (fin) is void.

Finally, an existing proof case is refined (Ref-Prf), which is discussed in section 5.4.

Appendix V: Two ASM Refinement Examples

We give two examples for non-trivial ASM refinements which illustrate the combined refinement of data structures and computation steps.

The first example is a case of conservative refinement. Let trustfulVM be the complete JVM interpreter, as defined in Part II of Jbook. Let verifyVM be the machine that verifies single bytecode methods, using appropriate data structures, as defined in Part III of Jbook. Then trustfulVM is conservatively refined by including a bytecode verifier which contains verifyVM as main component, as defined by Figure 8 For the refined machine, called diligentVM, one can prove the soundness and completeness of bytecode verification, combining appropriately the existing soundness proofs for the components trustfulVM and verifyVM, see Ch.17 of Jbook for the details.

The second example is a refinement of the execJava interpreter for single Java threads (defined in Ch. 1-6 of Jbook) to a machine that interpretes the concurrent execution of multiple Java threads, as defined in Ch.7 of Jbook. What the extension essen-
tially does is to add a scheduler that at each of its steps chooses a thread for execution by the execJava component, following its specific and independently definable and analyzable selection criteria. In it has been shown for the C# analogue of this construction how to prove properties of interest for such a thread handling model, using the ASM interpreter for C# defined in [Börger 2005].