Journal of Universal Computer Science, vol. 14, no. 12 (2008), 2059-2082
submitted: 1/9/07, accepted: 1/6/08, appeared: 28/6/08

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 recognized 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 existent 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.[1] 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.

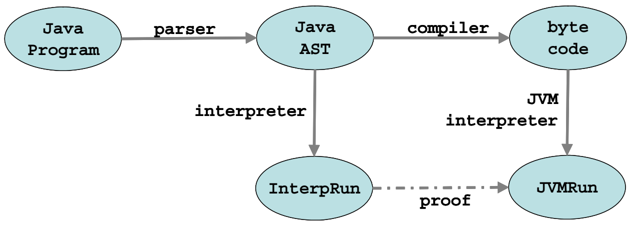

Figure 1: Jbook Organization

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 focussed 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:

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

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

```
Java1.0 = StmE•ExpE•ExpO•StmC•ExpC•StmI•ExpI                    (1)
```

where • denotes function composition. That is, `Java1.0` was incrementally developed in the Jbook by starting from base `ExpI` (which defines the Java sublanguage of imperative expressions, an interpreter of this sublanguage, a compiler, etc.), then `StmI` refines it, then `ExpC` refines `StmI•ExpI`, etc. In Jbook, only when the composition of `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 `JB` features that were given special names in the Jbook.

Jbook Term	*Composition*
$\mathtt{Java}_{\mathtt{I}}$	`StmI•ExpI`
$\mathtt{Java}_{\mathtt{C}}$	$\mathtt{StmC•ExpC•Java}_{\mathtt{I}}$
$\mathtt{Java}_{\mathtt{O}}$	$\mathtt{ExpO•Java}_{\mathtt{C}}$
$\mathtt{Java}_{\mathtt{E}}$	$\mathtt{StmE•ExpE•Java}_{\mathtt{O}}$
`Java`	`Java1.0 (see (1))`

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 `Java1.0` to produce sublanguages of Java. (A slightly different feature set than `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-and-matched, 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 `JB` 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 `ExpI` (i.e., I_{ExpI}) denotes the Jbook term $\texttt{execJavaExp}_{I}$.[1]

Our Term	*Jbook Term*	*Meaning*
G_i	$\texttt{syntax}_i$	language grammar
I_i	$\texttt{execJava}_i$	language interpreter
C_i	$\texttt{compile}_i$	language compiler
J_i	$\texttt{trustfulVM}_i$	virtual machine
T_i	$\texttt{theorem}_i$	theorem of compiler correctness

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 `JB` value `ExpI` are: the grammar for Java imperative expressions G_{ExpI}, the ASM definition of the expression interpreter I_{ExpI}, the ASM definition of the expression compiler C_{ExpI}, the ASM definition of the bytecode (JVM) interpreter J_{ExpI}, and the verification (theorem) representation T_{ExpI} which we will explain shortly. The tuple for program `ExpI` is $[G_{ExpI}, I_{ExpI}, C_{ExpI}, J_{ExpI}, T_{ExpI}]$.

A GenVoca function maps a tuple of program representations to a tuple of (in the Jbook called horizontally) refined representations. Feature `StmI` refines the base grammar by ΔG_{StmI} (new rules for Java statements and tokens are added), the language interpreter by ΔI_{StmI} (to implement the new statements), the compiler by ΔC_{StmI} (to compile the new statements), etc. `StmI`'s tuple is $[\Delta G_{StmI}, \Delta I_{StmI}, \Delta C_{StmI}, \Delta J_{StmI}, \Delta T_{StmI}]$.

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 $\texttt{Java}_I$ that has imperative expressions and statements is:

$$
\begin{aligned}
\texttt{Java}_I &= \texttt{StmI}\bullet\texttt{ExpI} \qquad \text{// GenVoca expression}\\
&= [\Delta G_{StmI}, \Delta I_{StmI}, \Delta C_{StmI}, \Delta J_{StmI}, \Delta T_{StmI}] \bullet \\
&\qquad [G_{ExpI}, I_{ExpI}, C_{ExpI}, J_{ExpI}, T_{ExpI}]\\
&= [\Delta G_{StmI}\bullet G_{ExpI}, \Delta I_{StmI}\bullet I_{ExpI}, \Delta C_{StmI}\bullet C_{ExpI},\\
&\qquad \Delta J_{StmI}\bullet J_{ExpI}, \Delta T_{StmI}\bullet T_{ExpI}]
\end{aligned}
$$

That is, the grammar of the $\texttt{Java}_I$ language is the base grammar composed with its refinement ($\Delta G_{StmI}\bullet G_{ExpI}$), the ASM definition of the $\texttt{Java}_I$ interpreter is the base defi-

1. In the Jbook also $\texttt{compiler}_i$ has a modular structure, being composed out of a compiler $\mathcal{E}$ for expressions, $\mathcal{S}$ for statements, and $\mathcal{B}$ for flow-control expressions.

nition composed with its refinement ($\Delta\mathtt{I}_{\mathtt{StmI}}\bullet\mathtt{I}_{\mathtt{ExpI}}$), 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 $\mathtt{T}$, as a representative example.

$\mathtt{T}_{\mathtt{ExpI}}$ denotes the theorem for the correctness of the `ExpI` compiler, i.e., the proof that interpreter runs of an `ExpI` program are equivalent to the JVM run of the compiled program. The refinement of this theorem by the `StmI` feature is denoted by $\Delta\mathtt{T}_{\mathtt{StmI}}$ in tuple `StmI`. The expression $\Delta\mathtt{T}_{\mathtt{StmI}}\bullet\mathtt{T}_{\mathtt{ExpI}}$ assembles the correctness theorem for the $\mathtt{Java}_{\mathtt{I}}$ language. Similarly for the languages $\mathtt{Java}_{C}$, $\mathtt{Java}_{O}$, $\mathtt{Java}_{\mathtt{E}}$ and `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 ($\bullet$) 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 $\Delta\mathtt{T}$ may refine its statement ($\Delta\mathtt{S}$) and/or its proof ($\Delta\mathtt{P}$). A composite theorem is produced by composing its subrepresentations, i.e., $\Delta\mathtt{T}\bullet\mathtt{T}=[\Delta\mathtt{S}\bullet\mathtt{S},\Delta\mathtt{P}\bullet\mathtt{P}]$. 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 $\mathtt{A}\rightarrow\mathtt{A}$ that maps an input artifact of a type `A` to a modified (usually extended) artifact of the same type (e.g., $\mathtt{source}\rightarrow\mathtt{source}$, $\mathtt{grammars}\rightarrow\mathtt{grammars}$, 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:

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

That is, if `condNew` (the condition of the new case) is not satisfied, invoke the original method, which is denoted by `SUPER.m()`. Otherwise execute the new actions. Exploiting a technique that is well-known from logic one can prove a theorem for a conservative extension ΔM 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 ΔT•T=[ΔS•S,ΔP•P]. Typically this involves a case distinction within an induction on runs of ΔM•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 ΔM-step, in which case the proof extension ΔP for the extended statement ΔS 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, typically coming with the same guard of the original ASM. In effect, the example rule (b) below is added "after" rule (a):

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

By ASM semantics, both are executed simultaneously if `cond` is satisfied, effectively extending the first rule to be:

```
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:

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

where either `before` or `after` could be null. Historically, a null `before` is called an *after-method*, a null `after` is a *before-method*, and non-null `before` and `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 semantically equivalent executions.[1]

The typical theorem refinement scheme for parallel addition is the conjunction, where T=[Sa,Pa] represents the theorem for rule (a) and ΔT=[ΔS,ΔP]=[Sb,Pb] the analogue 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 $\mathtt{G_{ExpI}}$ 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:

```
Exp     :=  Lit | Loc | Uop Exp | Exp Bop Exp
        |   Exp ? Exp : Exp | Asgn |Field
        |   Class.Field | Invk
Invk    :=  Meth(Exps) | Class.Meth(Exps)
Exps    :=  (Exp)+
Asgn    :=  Loc = Exp |Field = Exp
        |   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`, `StmI`, 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 (•) for statements and proofs is logical conjunction: `ΔT•T=[Sb∧Sa,Pb∧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., `Abruption`, `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 `Abruption`, where an *abruption* is an interruption in flow control.[1] Thus, the class hierarchy of Figure 4 is progressively revealed as features are composed. This is typical of FOP designs.

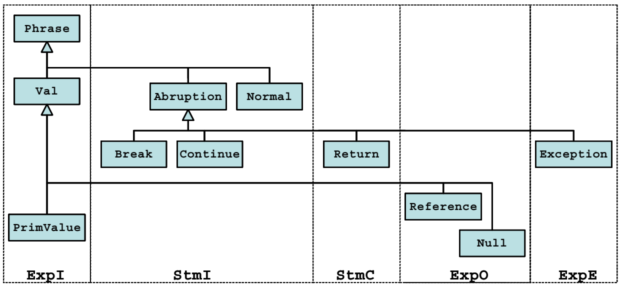

Figure 4: Refinement of Phrase Inheritance Hierarchy

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`):

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

Feature `ExpO` 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 `red 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 `T` defines the correctness of the Java compiler. The statement of `T` 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 `T` (denoted by `T.S`) and its proof (denoted by `T.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.

Invariant	Description
`(reg)`	the equivalence of local variables in the language interpreter and the associated registers in the JVM interpreter when both are in corresponding states
`(begE)`	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
`(exp)`	same as `(begE)` for a value returning termination of an expression evaluation
`(begS)`	same as `(begE)` except it applies to statement execution
`(stm)`	conditions for normal statement termination
`(abr)`	conditions for abrupted statement execution
`(stack)`	frame-stack equivalence condition
`(clinit)`	class initialization status equivalence condition
`(exc)`	conditions for exception statement execution

Figure 5: Invariants Used in Compiler Correctness Proofs

5.3.1 Adding New Invariants

`T.S` is a conjunction of invariants, which is subject to incremental refinement by adding further conjuncts. The conjunction is represented here as a list. `T.S` of the initial `ExpI` sublanguage has the `(reg)`, `(begE)` and `(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 `ExpI` concepts.

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

Similarly, the `ExpC` feature adds as further conjuncts the `(stack)` and `(clinit)` invariants to `T.S`; the `StmC` feature leaves `T.S` unchanged.

Given these features, Figure 6 lists their compositions and the invariants of `T.S` for each composition. The `T.S` for composition `j1` has three invariants; the `T.S` for `j3` and `j4` have eight. As Figure 6 shows, the set of invariants that define the statement of theorem `T` in the `JB` product-line varies from program to program. The remaining `JB` features introduce the remaining invariants of `T` for `Java1.0`.

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 `j3`, then the meaning of invariant `(reg)` ranges not only on the `ExpI` expressions, as in `j1`, but also on the new expressions defined by the `ExpC` grammar. The same remark applies to the theorem statement refinement of `stm` properties.

	j1	j2	j3	j4	Java1.0
(reg)	✓	✓	✓	✓	✓
(begE)	✓	✓	✓	✓	✓
(exp)	✓	✓	✓	✓	✓
(begS)		✓	✓	✓	✓
(stm)		✓	✓	✓	✓
(abr)		✓	✓	✓	✓
(stack)			✓	✓	✓
(clinit)			✓	✓	✓
(exc)					✓

where:

j1 = ExpI
j2 = StmI•ExpI
j3 = ExpC•StmI•ExpI
j4 = StmC•ExpC•
        StmI•ExpI

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 `(abr)` invariant, which is as follows:

if $restbody_n$/A=abr then <cond_1> (2)

That is, `<cond_1>` must hold when an abruption occurs. In section 5.2.1 we saw that feature `ExpE` extends the definition of an abruption to include exceptions. The `<cond_1>` of (2) applies *only* to abruptions that are *not* exceptions.

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

if $restbody_n$/A=abr *and abr is not an exception*
then <cond_1> (3)

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

if $restbody_n$/A=abr *and abr is an exception ...*
then <cond_2> (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 `T.S` for feature composition `C`. If `G` is another feature, `G•C` must be shown to satisfy $T.S_{(G\bullet C)}$ — the (conjunction of the) invariants collected and refined by `G•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 $\texttt{T.P}_{(C)}$ by adding more cases and/or refining existing cases. Below we examine the refinements of **T.P** that are made by each of the **ExpI**, **StmI**, **ExpC** and **StmC** features.

The **ExpI** feature defines the imperative expressions of Java. Recall the recursive **ExpI** grammar definition:

```
Exp       :=   Lit | Loc | Uop Exp | Exp Bop Exp
          |    Exp ? Exp : Exp | Asgn
Asgn      :=   Loc = Exp
```

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

The **StmI** feature introduces the imperative statements of Java. Its grammar refinement adds productions for Java statements; no **ExpI** productions are refined:

```
Stm :=         ; | Loc = Exp; | Lab : Stm;
          |    break Lab; | continue Lab;
          |    if (Exp) Stm else Stm
          |    while (Exp) Stm | Block                    (5)
```

The invariants that **StmI** adds are about statement executions, while the invariants of **ExpI** are about expression evaluations. Execution steps of the **ExpI** interpreter trivially preserve the **StmI** invariants, and vice versa, as these invariants relate sets of items that are disjoint.[1] For the composed interpreters to satisfy the invariants of **StmI•ExpI**, **StmI** must add cases to **T.P**, one or more for each production in (5), that prove the invariants of **StmI** are preserved; see cases 14-35 in [Stärk 2001]. Note that this induction on statements uses the proofs for the statement subexpression invariants as induction hypothesis.

The grammar refinement of feature **ExpC** adds expressions for static class fields, assignments to them, expression sequences and method invocations. The interpreter refinement of **ExpC** introduces frames and a frame stack and their values are related by the new invariant **(stack)**. The second new invariant **(clinit)** relates the class initialization status of Java and JVM interpreter runs. As no **ExpI** and **StmI** interpreter step references or updates frames or the class initialization status, invariants **(stack)** and **(clinit)** are trivially satisfied by them. **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`.

Composition	total # of cases in Proof of Theorem `T`
`j1 = ExpI`	13
`j2 = StmI•ExpI`	35
`j3 = ExpC•StmI•ExpI`	44
`j4 = StmC•ExpC•StmI•ExpI`	54
`Java1.0`	83

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(pos_n) = $^{\alpha}$(loc = $^{\beta}$ val) and pos_n=β:
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 `Exp` (expression), and it has concrete subclasses for literals (`Lit`), variables (`Loc`), unary expressions (`Uop`), etc. Instances of these classes define an AST for a parsed expression [Batory 1998].

The Java interpreter defines an abstract method `interpret()` in the `Exp` class, and all subclasses are obliged to provide implementations of this method (to interpret an expression). Similarly, a compiler defines an abstract method `compile()` in the `Exp` class, and all subclasses must provide implementations of this method (to compile an expression). 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 `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 automatically 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 representation 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 stepwise refinement had been "splitting the overall definition and verification problem into a series of tractable subproblems" ([Stärk 2001] p7) for a complete (not some lightweight) version of Java/JVM. The question remains whether we found an explicit feature-based formulation and proof of properties of interest because of the special character 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 structure 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 twenty years of building GenVoca product-lines, virtually all the features we have encountered 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 definitions 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 usually 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 characterized 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

[Albahari 2001] Albahari, B., Drayton, P., Merril, B.: *C# Essentials*. O'Reilly and Associates, 2001.

[Appel 2003] Appel, A., Michael, N., Strump, A., Virga, R.: “A Trustworthy Proof Checker”, *J. of Automated Reasoning*, 2003.

[Batory 1992] Batory, D., O’Malley, S.: “The Design and Implementation of Hierarchical Software Systems with Reusable Components”. *ACM TOSEM*, October 1992.

[Batory 1998] Batory, D., B. Lofaso, B., Smaragdakis, Y.: “JTS: Tools for Implementing Domain-Specific Languages”. *ICSR 1998*.

[Batory 2004] Batory, D., Sarvela, J., Rauschmayer, A.: “Scaling Step-Wise Refinement”. *IEEE TSE*, June 2004.

[Batory 2005] Batory, D.: “Feature Models, Grammars, and Propositional Formulas”. *SPLC 2005*.

[Blundell 2004] Blundell, C., Fisler, K., Krishnamurthi, S., van Hentrenrck, P.: “Parameterized Interfaces for Open System Verification of Product Lines”. *ASE 2004*.

[BMW 2007] BMW: `www.bmwusa.com`

[Börger 1995] Börger, E., Rosenzweig, D.: “The WAM - Definition and Compiler Correctness”. In *Logic Programming: Formal Methods and Practical Applications, Studies in Computer Science and Artificial Intelligence*, 11, North-Holland, 1995

[Börger 1999] Börger, E., Schulte, W.: “Initialization Problems for Java”. *Software—Concepts & Tools*, 20(4), 1999.

[Börger 2000] Börger, E., and Schmidt, J.: “Composition and Submachine Concepts for Sequential ASMs”. *CSL 2000*.

[Börger and Stärk 2003] E. Börger, E., Stärk, R.: *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[Börger 2003] Börger, E.: “The ASM Refinement Method”, *Formal Aspects of Computing*, 2003.

[Börger 2004] Börger, E., Stärk, R.: “Exploiting Abstraction for Specification Reuse. The Java/C# Case Study”, in *Formal Methods for Components and Objects: Second International Symposium* (FMCO 2003 Leiden), 42-76, 2004.

[Börger 2005] Börger, E., Fruja, G., Gervasi, V., and Stärk, R.: “A High-Level Modular Definition of the Semantics of C#”. *Theoretical Computer Science*, Vol. 336 #2-3, 2005.

[Calder 2003] Calder, M., Kolberg, M., Magill, E., Reiff-Marganiec, S.: “Feature Interaction: A Critical Review and Considered Forecast”, Computer Networks, #41, 2003.

[Czarnecki 2000] Czarnecki, K. and Eisenecker, U.: *Generative Programming Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000.

[Czarnecki 2006] Czarnecki, K., Pietroszek, K.: “Verification of Feature-Based Model Templates Against Well-Formedness OCL Constraints”. *GPCE 2006*.

[Dell 2007] Dell Computers: `www.dell.com`

[Fruja and Börger 2006] Fruja, N., Börger, E.: “Modeling the .NET CLR Exception Handling Mechanism for a Mathematical Analysis”. *Journal of Object Technology*, Vol. 5#3, 2006.

[Fruja 2004] Fruja, N.: “Specification and Implementation Problems for C#”. *ASM 2004*.

[Fruja and Börger 2006] Fruja, N.: “Type Safety of C# and .NET CLR”, Ph.D. Thesis, ETH Zurich, 2006.

[Gargantini 2000] Gargantini, A., Riccobene, E.: “Encoding Abstract State Machines in PVS”, *Abstract State Machines: Theory and Applications*, Springer-Verlag, 2000.

[Goerigk 1996] Goerigk, W., et al.: “Compiler Correctness and Implementation Verification: The Verifix Approach”, *Int. Conf. on Compiler Construction*, Proc. Poster Session of CC 1996.

[Grammes 2007] Grammes, R., Gotzhein, R.: “SDL Profiles — Formal Semantics and Tool Support”. *FASE 2007*.

[Hoare 2005] Hoare, T., Misra, J., Shankar, N.: “The IFIP Working Conference on Verified Software: Theories, Tools, Experiments”. `vstte.ethz.ch/report.html`, October 2005.

[Hunt 2006] Hunt, G., et al.: “Sealing OS Processes to Improve Dependability and Security”. Microsoft Research Technical Report MSR-TR-2005-135, April 2006.

[Java 2008] Java Card Language Specs: `java.sun.com/products/javacard/specs.html`

[Jones 2006] Jones, C., O’Hearn, P., Woodcock, J.: “Verified Software: A Grand Challenge”. *IEEE Computer*, April 2006.

[Kästner 2007] Kästner, C., Apel, S., Batory, D.: “A Case Study Implementing Features Using AspectJ”, *SPLC* 2007.

[Kiczales 1991] Kiczales, G., des Rivieres, J., Bobrow, D.: *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Krishnamurthi 2001] Krishnamurthi, S., Fisler, K.: “Modular Verification of Collaboration-Based Software Designs”. *FSE 2001*.

[Krishnamurthi 2004] Krishnamurthi, S., Fisler, K., Greenberg, M.: “Verifying Aspect Advice Modularly”. *ACM SIGSOFT 2004*.

[Levens 2006] Leavens, G., et al.: “Roadmap for Enhanced Languages and Methods to Aid Verification”. GPCE 2006

[Liu 2006] Liu, J., Batory, D., Lengauer, C.: “Feature Oriented Refactoring of Legacy Applications”, *ICSE 2006*.

[Pohn 2005] Pohl, K., Bockle, G., v.d. Linden, F.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer 2005.

[Schellhorn 1998] Schellhorn, G., Ahrendt, W.: “The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV”, *Automated Deduction — A Basis for Applications III*, Kluwer Academic Publishers, 1998.

[Schellhorn 1998] Schellhorn, G.: "Verification of Abstract State Machines", Ph.D. Thesis, University of Ulm, 1999.

[Schellhorn 2007] Schellhorn, G. et al.: "A Systematic Verification Approach for Mondex Electronic Purses Using ASMs". *Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis.* LNCS 2007 (to appear).

[Stärk 2001] Stärk, R., Schmid, J., Börger, E.: *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

[Stärk 2004] Stärk, R., Börger, E.: "An ASM Sepcification of C# Threads and the .NET Memory", *Abstract State Machines 2004*, LNCS Vol. 3052.

[Stärk 2005] Stärk, R.: "Formal Specification and Verification of the C# Thread Model", *Theoretical Computer Science*, Vol 343, 2005.

[Thaker 2007] Thaker, S., Batory, D., Kitchin, D., Cook, W.: "Safe Composition of Product Lines". *GPCE 2007*.

[Taha 1997] Taha, W., Sheard, T.: "Multi-Stage Programming with Explicit Annotations", *PEPM 1997*.

[Woodcock 2007] Woodcock, J., Banach, R.: "The Verification Grand Challenge", *JUCS* Vol 13#5, 2007.

[Xie 2003] Xie, F., Browne, J. "Verified System by Composition from Verified Components". *ACM SIGSOFT/FSE 2003*.

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):

```
f         // base program with feature f
h         // base program with feature h
```

Features are unary functions:

```
i•x       // adds feature i to program x
j•x       // adds feature j to program x
```

where the operator • denotes function composition.

The *design* of a program is a named expression:

```
p1 = j•f       // p1 has features j and f
p2 = j•h       // p2 has features j and h
p3 = i•j•f     // p3 has features i,j,f
```

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 Δg_j (new rules and tokens are added), extends source code by Δs_j (new classes and members are added and existing methods are modified), and extends documentation by Δd_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:

$$
\begin{aligned}
p_1 &= j \bullet f && \text{; GenVoca expression}\\
&= [\Delta g_j, \Delta s_j, \Delta d_j] \bullet [g_f, s_f, d_f] && \text{; substitution}\\
&= [\Delta g_j \bullet g_f, \Delta s_j \bullet s_f, \Delta d_j \bullet d_f] && \text{; composition}
\end{aligned}
$$

That is, the grammar of p_1 is the base grammar composed with its extension ($\Delta g_j \bullet g_f$), the source of p_1 is the base source composed with its extension ($\Delta s_j \bullet s_f$), 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 $\bullet$ 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

```
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(exp1,...,expn):=exp` with arbitrary expressions `expi`, `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:

```
Forall x with Cond(x) do rule(x)
Choose x with Cond(x) do rule(x)
```

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 `SUPER.m()` may be conditional. Consider the following refinement pattern:

```
void m() {before; if (cond) SUPER.m(); after;}                    (6)
```

which is a blend of parallel addition and conservative extension. A special case of (6) that arises infrequently is when `cond` is always `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:

```
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 `remove()` is:

```
void remove() {set delete flag of element;
      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 `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 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-

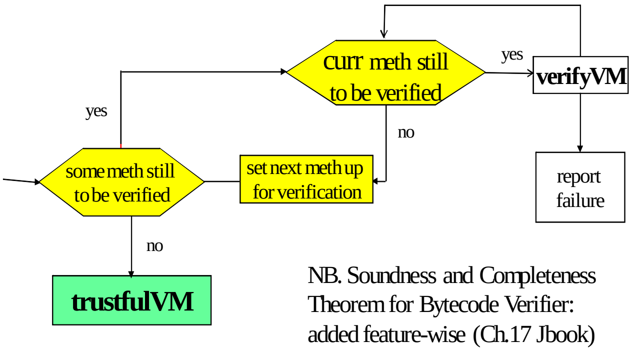

Figure 8: Adding the ByteCode Verifier

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].