On The Modularization of Theorems for Software Product Lines

Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
batory@cs.utexas.edu

Egon Börger
Dipartimento di Informatica
Università di Pisa
I-56127 Pisa, Italy
boerger@di.unipi.it

Abstract

A goal of software product lines is the economical synthesis of programs in a family of programs. In this paper, we explain how theorems about program properties can 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 explain how features modularize both programs and theorems. By composing features, the source code and theorems for a program are synthesized. Generated theorems may then be certified manually or automatically using a proof checker, opening a new line of research in verification.

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 [7][17][32]. Sadly, what distinguishes software products is the absence of meaningful warranties [25]. While great strides have been made in verification over the last ten years, there are few results on verifying *software product-lines (SPLs)* [16][28][29][39].

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 [24][40]. In this paper, we explore an approach that suggests how feature modularization can scale verification to product-lines of programs. We bring together results from previously unrelated communities: *Abstract State Machines (ASM)* and *Feature-Oriented Programming (FOP)*. ASM is a rigorous method for program specification and verification. FOP is a design methodology and compositional technology for program synthesis. ASM and FOP both use step-wise refinement to construct programs and specifications. Our case study is the 2001 JBook [37] 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 [9]. More recent work examined C# with similar results [13][18][19] [20].

Although ASM and FOP were conceived independently (their roots trace back to the early 1990s), both use *features* — increments in functionality — as a modularization centerpiece. JBook and FOP use features to modularize grammars and programs in an identical way. The contributions of our paper are justifying this claim and more importantly showing how features modularize JBook theorems (both their statements and proofs) on the correctness of program properties. By composing features, complete grammars, programs, and theorems are synthesized.

Our results are not limited to JBook or compilers; they are meaningful in the context of SPLs where each program of an SPL 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 can be synthesized, exactly like other program representations. Generated theorems may then be certified manually or automatically using a proof checker. Consequences of these ideas are opening new lines of research in verification.

2 An Overview of the JBook Product Line

JBook [37] 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, a complete grammar, language interpreter, compiler, and bytecode interpreter for Java 1.0 are produced.

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

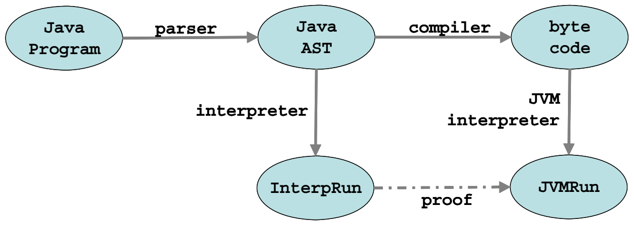

Figure 1. JBook Organization

JBook was not developed with product lines in mind. It focussed on the definition and verification of a single interpreter and compiler for the Java 1.0 language. To give JBook an SPL architecture, we present a series of more elaborate FOP models.

GenVoca is a model of product-lines: base programs are values (constant functions) and features are functions that map programs to refined programs [3]. A GenVoca model of JBook is an algebra `JB`, where each element of `JB` is a JBook refinement 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 constant `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`:

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

where • denotes function composition. That is, `Java1.0` was incrementally developed 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. Only when the composition of `Java1.0` was complete were manual proofs of correctness developed. *We want to show how correctness proofs can be synthesized at each composition step.*

Note: Figure 2 lists compositions of `JB` features that were given special names in the JBook.

JBook Term	*Composition*
$\texttt{Java}_{\texttt{I}}$	`StmI•ExpI`
$\texttt{Java}_{\texttt{C}}$	$\texttt{StmC•ExpC•Java}_{\texttt{I}}$
$\texttt{Java}_{\texttt{O}}$	$\texttt{ExpO•Java}_{\texttt{C}}$
$\texttt{Java}_{\texttt{E}}$	$\texttt{StmE•ExpE•Java}_{\texttt{O}}$
`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, but these sublanguages are not very interesting. A logical generalization of `JB` is to add new language constructs: updating to Java 1.5, support for state machines [5] and Lisp quote/unquote metaprogramming constructs [38]. Features can then 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 [4][5].

3 AHEAD Representation of JBook SPL

AHEAD is a generalization of GenVoca that exposes different representations of programs and reveals how features refine each of these representations by composition [5]. 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., $\texttt{I}_{\texttt{ExpI}}$) denotes the JBook term $\texttt{execJavaExp}_{\texttt{I}}$.[2]

3.1 Program Representations

Every program has multiple representations: source, documentation, bytecode, makefiles, etc. A GenVoca constant is a vector of representations for a base program. The repre-

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.

2. $\texttt{compiler}_{\texttt{i}}$ has different names in the JBook. $\mathcal{E}$ denotes the compiler for expressions, $\mathcal{S}$ for statements, and $\mathcal{B}$ for flow-control expressions.

Our Term	*JBook Term*	*Meaning*
G_i	$syntax_i$	language grammar
I_i	$execJava_i$	language interpreter
C_i	$compile_i$	language compiler
J_i	$trustfulVM_i$	virtual machine
T_i	$theorem_i$	theorem of compiler correctness

Figure 3. Name Correspondences

sentations of the `JB` constant `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 byte-code (JVM) interpreter J_{ExpI}, and the verification (theorem) representation T_{ExpI} which we will explain shortly. Program `ExpI`'s vector is $[G_{ExpI}, I_{ExpI}, C_{ExpI}, J_{ExpI}, T_{ExpI}]$.

A GenVoca function maps a vector of program representations to a vector of 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 vector is $[\Delta G_{StmI}, \Delta I_{StmI}, \Delta C_{StmI}, \Delta J_{StmI}, \Delta T_{StmI}]$.

The representations of a program are computed by vector composition, where corresponding components are composed. The grammar, interpreter, compiler, etc. representations of the Java sublanguage $Java_I$ that has imperative expressions and statements is:

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

That is, the grammar of the $Java_I$ language is the base grammar composed with its refinement ($\Delta G_{StmI} \bullet G_{ExpI}$), the ASM definition of the $Java_I$ interpreter is the base definition composed with its refinement ($\Delta I_{StmI} \bullet I_{ExpI}$), and so on. In general, the representations of a program are synthesized by taking a GenVoca expression, replacing each term with its corresponding vector, and composing vectors.

3.2 Theorems

Theorems of 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_{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 ΔT_{StmI} in vector `StmI`. The expression $\Delta T_{StmI} \bullet T_{ExpI}$ synthesizes the correctness theorem for the $Java_I$ language. More on theorem refinement in Section 5.3.

3.3 Nested Vectors

Program representations have subrepresentations, and recursively, subrepresentations may have subrepresentations. Hierarchical containment relationships are expressed by allowing each term of a vector to be a vector that can be refined. In general, the composition operator (•) that we use recursively composes nested vectors. This is the essence of AHEAD [5].

As an example, theorems have a vector structure. Theorem `T` has a statement `S` and a proof `P`; `T`'s vector is `[S,P]`. A theorem refinement ΔT may refine its statement (ΔS) and/or its proof (ΔP). A composite theorem is produced by composing its subrepresentations, i.e., $\Delta T \bullet T = [\Delta S \bullet S, \Delta P \bullet P]$. Examples of nested representations of code are given in [5].

4 What is a Refinement?

A feature is a transformation that maps an input artifact to a modified (usually extended) artifact. The transformation is 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 [11] where AHEAD uses three. One is *conservative extension*: (a) define the condition for the new case, (b) define a new ASM machine to add the extra behavior, and (3) 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.

A second form of ASM refinement is *parallel addition*: (a) define a new ASM to add the extra behavior, and (b) guard the new ASM with the same conditions of the original ASM. In effect, rule (b) 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, this is expressed by the 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* [27]. See Appendix I for one more case.

Figure 4 shows the frequency of different kinds of refinements used in building the *AHEAD Tool Suite (ATS)* [5]. Interestingly before and after methods were most common, followed by around methods. Conservative extensions were infrequent. Also included in Figure 4 are the corresponding numbers for a legacy application (`FRBDB` [26]) that was retrofitted with a feature design. ATS represents one extreme where features had a premeditated design; `FRBDB` represents another where features were an after-thought. Both exhibit similar distributions.

Refinement Name	Frequency in ATS	Frequency in FRBDB	Code Pattern
`before`	26	142	`before; Super.m();`
`after`	22	124	`Super.m(); after;`
`around`	5	63	`before; Super.m(); after;`
`conservative`	4	13	`if (!condNew) SUPER.m(); else {...}`

Figure 4. Frequency of Primitive Refinements

It is hard to get comparable statistics for JBook, as ASM is a parallel rule-based language. As best as we can tell, before refinements and conservative extensions are common in JBook, and after and around are uncommon. The reason is that after refinements are largely hidden in ASMs because of their parallel-execution nature; sequential execution is encoded in state transitions rather than in straight-line code. In any case, both AHEAD and JBook use comparable ASM/method refinement techniques.

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 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, these simple concepts apply to theorems as well.

5 Refinement of Artifacts

We now document the feature-refinement of grammars, code, and theorems used in the JBook, and note their similarity to feature-oriented program development.

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:

```
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 refines this grammar by adding object fields and method calls (indicated in *`red italic`* font above). `ExpC` refines productions `Exp` and `Asgn` with additional right-hand sides, and introduces new productions `Invk` and `Exps`. Exactly the same technique was used in AHEAD to modularize and refine grammars [5].

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.

5.2.1 Refining Inheritance Hierarchies

JBook calls the result of executing an expression or statement a `Phrase`. Initially, `ExpI` defines a simple inheritance hierarchy rooted at `Phrase` (Figure 5). The only subclass is `Val`, and it has many different subclasses (`boolean`, `byte`, `short`, etc.) which are depicted by a single class `PrimValue`.

Feature `StmI` adds subclasses to `Phrase` that represent the possible results of executing imperative statements (e.g., `Break`, `Continue`, and `Normal`). Feature `StmC` adds 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.[3] Thus, the class hierarchy of Figure 5

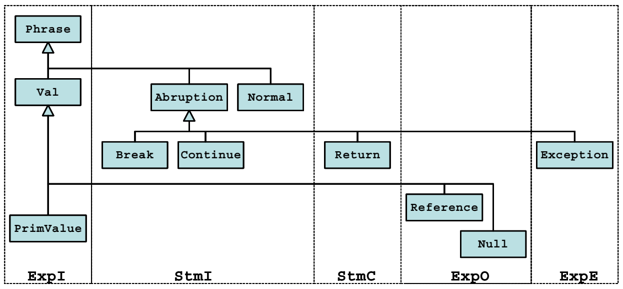

Figure 5. Refinement of Phrase Inheritance Hierarchy

is progressively revealed as features are composed. This is typical of FOP designs.

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. Feature `StmC` defines the ASM actions (in `black` font) taken when a method is exited:

```
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 constructors. The refinement adds the definition in *`red italic`* font above. This change can be easily expressed as a refinement of Java code.

5.3 Refining Theorems

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 6. *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.

3. `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.

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 execute 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 execution
`(begS)`	same as `(begE)` except it applies to statements
`(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 6. Invariants Used in Compiler Correctness Proofs

5.3.1 Adding New Invariants

`T.S` is a list of invariants. The `T.S` of the `ExpI` sublanguage is defined by the `(reg)`, `(begE)` and `(exp)` invariants. The remaining invariants of Figure 6 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 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 only `ExpI` and `StmI` concepts.

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

Given these features, Figure 7 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 7 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`.

	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 7. Statement of Correctness

5.3.2 Refining Existing Invariants

Program invariants are also subject to refinement. A sketch of the abruption `(abr)` invariant is:

```
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 *red italics*). `(2)` becomes:

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

Second, **`ExpE`** introduces invariant `(exc)` to `T.S.` to cover the case where an abruption *is* an exception:

```
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 correctness theorem. 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 set of invariants of `T.S` for feature composition `C`. If `G` is a feature, `G•C` must be shown to satisfy $T.S_{(G•C)}$ — the invariants collected and refined by `G•C`.

The structure of `T.P` in JBook is a list of cases. 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 **`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. variables) in the interpreter and JVM ASMs for **`ExpI`**. If these invariants hold, the interpreter and JVM executions are producing the same results. The `T.P` for **`ExpI`** is a list of proof cases, one or more cases for each kind of expression showing the **`ExpI`** invariants are preserved ([37] p179-184).

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.[4] 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. 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, and expression sequences. **`ExpC`** introduces frames and a frame stack to the language and JVM interpreters respectively, and their values are related by a new invariant `(stack)`. A second new invariant `(clinit)` relates the class initialization status of interpreter and JVM runs. As no **`ExpI`** and **`StmI`** interpreter step references or updates frames or the class initialization status, invariants `(stack)` and `(clinit)` are satisfied. **`ExpC`** refines `T.P` with additional cases proving these two new invariants hold, one case for each kind of new expression. Since no **`ExpC`** execution step affects any of the previous invariants, all the invariants hold for **`ExpC•StmI•ExpI`**.

`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 8 lists compositions of these 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 8. Proof of Correctness

4. In proof arguments, we tacitly make use of the fact that an ASM execution step is 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. Therefore, 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.

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 a new proof case for the new invariant **`(exc)`** which expresses the desired property for exceptions.

5.4 Refining 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`**, hold for variable assignment ([37] p183):

> ***Case 9: context(pos_n) = $^{\alpha}$(loc = $^{\beta}$ val) and pos_n=$^{\beta}$:***
> **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 statement return addresses. As the return addresses from **`finally`** code are stored by the JVM in dedicated registers, it has also to be checked that every register assignment preserves the invariant. Therefore the case concerning variable assignment must be refined with additional proof text (in ***red 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 II.

5.5 Further Structure

The ASM interpreter and compiler use a familiar object-oriented structure. Each right-hand side of a production corresponds to an abstract class and each left-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 [4].

The 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. 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, cases were initially missed (and subsequently discovered). Type-checking may have automatically reported the absence of missing cases. We will see in Section 7 that type checking can play a much more expansive role in certifying theorems.

5.6 Recap and Insight

An obvious question is: why does this work? 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 or monotonic semantics. To the best of our knowledge, in 20 years of building GenVoca product-lines, virtually all 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 usually accompanied by an abrupt discontinuity in semantics where prior properties are no longer valid. The telecommunications domain is replete with examples [14]. Appendix I shows how element definitions can be replaced. [31] is a general theory of how feature interactions can be integrated into a GenVoca model.

In summary, proofs, invariants, code and grammars have a structure. And within a structure, there are extension points or variation points [15] 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 [3][4][5][26].

6 Validation

The previous sections document the results of our validation. We spent many weeks (over several months) reviewing the JBook details to isolate the contents of individual features (ASM definitions, grammars, and theorems). The task was not difficult and sometimes tedious as the relevant part of the JBook on which we focussed covers 200 pages, including the correctness proof which occupies 37 pages.

We were pleased to find 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 step-wise refinement had been "splitting the overall definition and verification problem into a series of tractable subproblems" ([37] p7) for a complete (not some lightweight) version of Java/JVM. The explicit feature-based formulation and proof of the properties that were exposed confirmed our belief that a clear compositional design structure goes together with a feasible structure of system invariants and their proofs.

7 Future Challenges

Feature-modularizing proofs is only a first step toward the goal of automatically verifying programs in software product lines. Further progress can be made by examining what is synthesized 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?

In JBook and this paper, the proof analysis builds upon the understanding of the subject matter by the engineer. 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.[5] But manual certification of generated 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 [21][22][33][34][35]. In particular, Schellhorn has shown that developing proofs incrementally (much like the incremental development in JBook) simplifies the task of mechanically proving program properties [34].

Once the theorems are expressed in a machine manageable form, proof-checkers can 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 [2]. So verifying a program of a product-line is accomplished by generating the program and its theorems, and using a proof checker to certify theorems automatically. Doing this would be a meaningful step forward: it opens a new line of research that would further expose the structure (interfaces and variation points) of theorems that are refined by features, and it would provide a way to verify programs of a product-line automatically using proof checkers.

The approach outlined above is conceptually no different than verifying that the generated source of a program is type-correct, i.e., generate 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 [16][39]. This analysis should 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.

8 Related Work

There is an enormous literature on verification. For lack of space, we limit our discussion of related work to that relevant to verifying software product lines.

Not all features are compatible; some features preclude or require the use of others in a composition. Verifying compositions of features is discussed in [6], where feature models, grammars, and propositional formulas are related, and techniques for validating feature models are discussed.

Czarnecki used [6] to show how feature models could be used to check the well-formedness of all products of a product line [16]. [39] is a follow-on work that showed how to verify the type correctness of a product-line.

Collaborations are a fundamental way to express large-scale refinements of object-oriented programs [5]. Krishnamurthi and Fisler have explored the modular verification of collaborations using model checkers. Their work showed that predicates describing properties of state machines are refined by features [28][29].

Hoare, Misra, and Shankar proposed a Verified Software Grand Challenge in 2005 with the goal of scaling verification to a million lines of code [23][25]. Verification would be based on the text of a program and the annotations contained within it [30]. We, like others, believe that verification must be intimately integrated with software design and

5. The reported ratio for two verification efforts was 1 to 4 [8][33].

modularity [24][40]. Verifying product-lines, which requires the integration of design and verification, seems a fitting goal for a Verified Software Grand Challenge.

9 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 presented first steps toward this extended goal by showing how features 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.

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 verification and opens new lines of research. The next steps are to (a) evaluate the practicality of refining theorems, (b) certify generated 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 [5].

Acknowledgements. We thank Christian Lengauer, Harry Li, Salvador Trujillo for their helpful comments. We also thank Christian Kästner for the `FRBDB` statistics. This work was support in part by NSF's Science of Design Project #CCF-0438786.

10 References

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

[2] A.W. Appel, N. Michael, A. Strump and R. Virga. "A Trustworthy Proof Checker", *J. of Automated Reasoning*, 2003.

[3] D. Batory and S. O'Malley. "The Design and Implementation of Hierarchical Software Systems with Reusable Components". *ACM TOSEM*, October 1992.

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

[5] D. Batory, J.N. Sarvela and A. Rauschmayer. "Scaling Step-Wise Refinement". *IEEE TSE*, June 2004.

[6] D. Batory. "Feature Models, Grammars, and Propositional Formulas". *SPLC 2005*.

[7] BMW. `www.bmwusa.com`

[8] E. Börger and D. Rosenzweig. "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

[9] E. Börger and W. Schulte. "Initialization Problems for Java". *Software—Concepts & Tools*, 20(4), 1999.

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

[11] E. Börger. "The ASM Refinement Method", *Formal Aspects of Computing*, 2003.

[12] E. Börger and R.F. Stärk. "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.

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

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

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

[16] K. Czarnecki and K. Pietroszek. "Verification of Feature-Based Model Templates Against Well-Formedness OCL Constraints". *GPCE 2006*.

[17] Dell Computers. `www.dell.com`

[18] N.G. Fruja and E. Boerger. "Modeling the .NET CLR Exception Handling Mechanism for a Mathematical Analysis". *Journal of Object Technology*, Vol. 5#3, 2006.

[19] N.G. Fruja. "Specification and Implementation Problems for C#". *ASM 2004*.

[20] N.G. Fruja. "Type Safety of C# and .NET CLR", ETH Zurich, 2006.

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

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

[23] T. Hoare, J. Misra and N. Shankar. "The IFIP Working Conference on Verified Software: Theories, Tools, Experiments". tinyurl.com/nrhdl, October 2005.

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

[25] C. Jones, P. O'Hearn and J. Woodcock. "Verified Software: A Grand Challenge". *IEEE Computer*, April 2006.

[26] C. Kästner, S. Apel and D. Batory. "A Case Study Implementing Features Using AspectJ", *SPLC* 2007.

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

[28] S. Krishnamurthi and K. Fisler. "Modular Verification of Collaboration-Based Software Designs". *FSE 2001*.

[29] S. Krishnamurthi, K. Fisler and M. Greenberg. "Verifying Aspect Advice Modularly". *ACM SIGSOFT 2004*.

[30] G.T. Leavens, et al. "Roadmap for Enhanced Languages and Methods to Aid Verification". GPCE 2006

[31] J. Liu, D. Batory, and C. Lengauer. "Feature Oriented Refactoring of Legacy Applications", *ICSE 2006*.

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

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

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

[35] G. Schellhorn 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).

[36] G. Schellhorn. "ASM Refinement Preserving Invariants". *ASM 2007*.

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

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

[39] S. Thaker. "Design and Analysis of Multidimensional Program Structures". M.Sc. Thesis, Department of Computer Sciences, The University of Texas at Austin, 2006.

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

Appendix I: 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.

Appendix II: 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.

Consider an abruption that is not an exception, say due to a `return` 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 [37]), 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 5.3.3.