A Tool-Supported Integration of BON and JML

Richard Paige and Liliya Kaminskaya

Technical Report CS-2001-04

July 1, 2001

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

A Tool-Supported Integration of BON and JML

Richard Paige[1] and Liliya Kaminskaya[2]

[1] Department of Computer Science, University of York,
Heslington, York YO10 5DD, United Kingdom. paige@cs.york.ac.uk
[2] Department of Computer Science, York University,
4700 Keele Street, Toronto, Ontario M3J 1P3, Canada. liliyak@cs.yorku.ca

Abstract. We describe a tool-supported integration of an object-oriented formal method, BON, with an object-oriented formal modelling language, JML. The integration is both *artifactual* – carried out so as to exploit JML's existing and planned tool support – and *effectual*, in order to provide a graphical syntax and process support for JML. The integration is characterised using the meta-method of [13, 14] which provides a general approach for combining software development methods and modelling languages. A CASE tool is described that supports the integrated techniques.

1 Introduction and Motivation

Object-oriented (OO) modelling languages, such as UML [3], BON [27], Object-Z [23], and JML [9] are seeing increased use and research interest. The most widely used of these is UML, which presents a broad collection of modelling constructs, and is supported by compatible processes and tools. More specialised modelling languages, such as JML (aimed at specifying Java applications) and BON (particularly useful for specifying Eiffel applications, but more specifically for reliable, seamless, and reversible software development) see use in specific contexts and software development projects. In general, we are seeing two disparate types of modelling languages for OO development: general-purpose languages, such as UML; and specialised languages, such as BON and JML. It is the latter that we focus on in this paper.

BON and JML each have the following properties.

- They have their origins in formal specification languages. Each language includes documentation, in the form of *contracts* (pre- and postconditions of methods, and invariants of classes), which is never separated from the modelling abstractions themselves.
- They are relatively small languages that provide only a subset of the modelling constructs available in UML.
- Each approach has limitations. BON, in particular, provides no analytic tool support, e.g., for reasoning. Nor is guidance provided with the BON process on how to implement BON detailed designs in Java. JML, on the other hand, is a purely text-based modelling language, and provides no compatible process.

Much recent research has been carried out on *method integration*, the process of combining two or more independent methods to produce a new one. Method integration

has been proposed as an appropriate way to remove limitations with methods, to develop new methods, to acquire use of new modelling techniques and tools, and to assess the complementary nature of software development techniques. It also provides a rationale and mechanisms for constructing *simpler, smaller* software development methods that tackle one specific class of development problem: if a technique, modelling construct, or tool is not present or available in a method, then the method can be integrated with another that possesses the desired mechanisms.

In this paper, we present a loose tool-supported integration of BON and JML, in order to address the limitations of each approach. The integration is loose in two senses: the BON and JML languages can be used independently, without change, or they can be used in an integrated manner; and, the tools are integrated via use of file-sharing and APIs. In the terminology of Stirewalt and Dillon [24], the integration is both *artifactual* (it is done to exploit tools) and *effectual* (it is done so as to carry out tasks that could not be carried out with one or both of the separate techniques). The approaches are integrated using the meta-method of [13, 14], which provides a general approach for combining methods and modelling languages. We emphasize that the integration produces a new method: we consider integration of modelling languages and development process issues as well.

1.1 Overview

In Section 2, we provide an overview of BON and JML, focussing on the modelling languages, but also explaining the recommended, iterative BON development process. This process will be generalised to use the JML modelling language and its tools. The meta-method of [13, 14] is also explained. In Section 3, we integrate BON and JML. The complementary nature of the approaches is explained, the modelling languages are integrated, and the BON process generalised. We discuss complications with integrating the modelling languages, particularly due to the expressive differences between BON and JML (e.g., due to multiple inheritance, covariance, and genericity). We also discuss validation of the integration. Section 4 describes tool support for the integrated method, provided via a CASE tool for BON. Finally, in Section 5, we discuss lessons learned about integration, and consider future work.

2 Background

We provide a brief overview of BON and JML, focussing on explaining the notational differences, so as to provide justification for the complementary nature of the approaches. As well, we explain the approach we take to integrating BON and JML, via the meta-method of [13, 14].

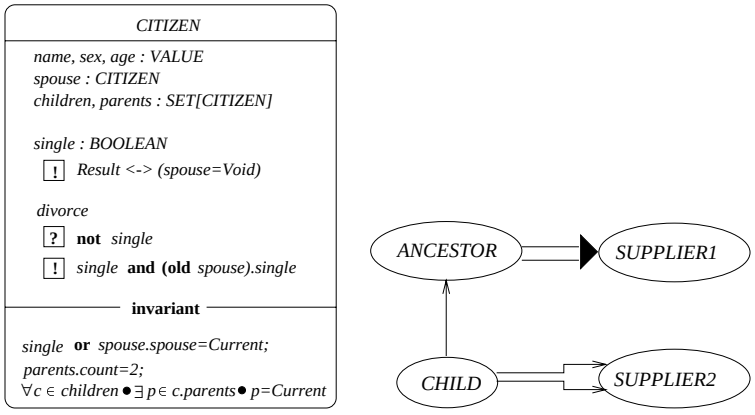
2.1 BON

BON is an OO method possessing a recommended process as well as a graphical language for specifying object-oriented systems. The language provides mechanisms for

specifying classes and objects, their relationships, and assertions (written in first-order predicate logic) for specifying the behaviour of routines and invariants of classes.

The fundamental construct in BON is the class. A class has a name, an optional class invariant, and zero or more features. A feature may be an *attribute*, a *query*—which returns a value and does not change the system state—or a *command*, which does change system state but returns nothing. Fig. 1(a) contains an example of a BON model for the interface of a class $CITIZEN$. Features are in the middle section of the diagram (there may be an arbitrary number of feature sections, annotated with names of classes that may access the features). Features may optionally have contracts (pre- and postconditions) written in the BON assertion language. Class invariants may also be specified; these capture properties that must be true whenever an instance of the class has one of its features called by a client. The assertion syntax is a dialect of first-order predicate logic, with sugar introduced (e.g., $Current$, $Result$) to make it easier to specify object-oriented systems.

In Fig. 1(a), class $CITIZEN$ has six attributes, one query, and one command. For example, $single$ is a $BOOLEAN$ query while $divorce$ is a parameterless command that changes the state of an object. $SET[G]$ is a generic predefined class with generic parameter G and the usual operators (e.g., $\in$, add). The class $SET[CITIZEN]$ thus denotes a set of objects each of type $CITIZEN$.



(a) Citizen interface (b) BON relationships

Fig.1. BON syntax for interfaces and relationships

BON models consist of classes organized in *clusters* (drawn as dashed rounded rectangles that may encapsulate classes and other clusters). Classes and clusters interact via two general kinds of relationships.

– **Inheritance:** Inheritance defines a subtype relationship between a child and one or more parents. The inheritance relationship is drawn between classes $CHILD$ and

$ANCESTOR$ in Fig. 1(b), with the arrow directed from the child to the parent class. In this figure, classes have been drawn in their compressed form, as ellipses, with their interface details hidden.

– **Client-supplier:** there are two client-supplier relationships, association and aggregation. Both relationships are directed from a *client* to a *supplier*. Association depicts reference relationships, while aggregation depicts subobject (or part-of) relationships. Client-supplier relationships can be drawn between classes and clusters; recursive rules are given in [27] to explain the meaning of cluster relationships. The aggregation relationship is drawn between classes $CHILD$ and $SUPPLIER2$ in Fig. 1(b), whereas an association is drawn from $ANCESTOR$ to class $SUPPLIER1$.

BON also provides notation for dynamic models, showing the communication between objects. This notation is very similar to collaboration diagrams in UML [3]. BON is similar in many respects to the core of UML.

2.1.1 The BON process The BON development process is iterative, risk-driven, and idealised; compatibility with the BON process is defined in terms of producing a required set of document deliverables, including class diagrams, dynamic diagrams, scenarios and charts, etc. Each task in the process has a set of input sources, produces a set of deliverables, and is controlled by acceptance criteria, which take into account the risk of proceeding. Each task may be iterated several times; feedback may occur from successive tasks to preceding tasks. Unlike the Rational Unified Process [8], the BON process is not use-case driven, but it is architecture-centric; use-cases can but are not required to be applied in the process's early stages. The process also emphasises using contracts to capture the behaviour of modelling abstractions, instead of statecharts. The process is sketched in Fig. 2.

Task	Description
1	Delineate system borderline.
2	List candidate classes.
3	Select classes and group into clusters.
4	Define classes and their features.
5	Sketch system behaviours using dynamic models.
6	Define public features and contracts.
7	Refine system.
8	Factor out common behaviour.
9	Complete and review system.

Fig.2. The BON idealised process

Task #7, refinement, involves taking a preliminary design and finding new design classes, as well as new features for existing classes; it is intended to iterate with Task #6. Task #8 is step aimed at producing reusable classes – it is sometimes called generalisation. A BON static model is first produced as a deliverable from Task #6. The

integration of BON and JML will extend this process by an additional, potentially iterated task introduced in between Task #6 and Task #7.

2.2 JML

The Java Modelling Language, JML, is a behavioural interface specification language that is tailored to Java [9]. It is an ASCII-based specification language that can be used to specify Java modules. It can be used as a stand-alone specification language, capturing constraints on methods, classes, and interfaces. It can also be used in combination with Java, allowing contracts to be embedded as comments within Java code as an aid to verification and debugging. JML supports a number of features common to specification languages, such as quantifiers, specification-only variables, and frame conditions. As well, it provides features that are useful in specifying Java classes and interfaces.

Fig. 3 provides an example of an abstract class specification in pure JML: the unbounded stack.

```
public abstract class UnboundedStack {

 /*@ public model JMLObjectSequence contents
   @    initially: contents != null && contents.isEmpty();
   @*/

 //@ public invariant: contents != null

 public abstract void pop();
 /*@ public_normal_behavior
   @ requires: !contents.isEmpty();
   @ modifiable: contents;
   @ ensures: contents.equals(\old(contents.trailer()));
   @*/

 public abstract void push(Object x);
 /*@ public_normal_behavior
   @ modifiable: contents;
   @ ensures: contents.equals(\old(contents.insertFront(x)));
   @*/

 public abstract Object top();
 /*@ public_normal_behavior
   @ requires: !contents.isEmpty();
   @ ensures: \result == contents.first();
   @*/
}
```

Fig.3. JML specification of an unbounded stack

Abstract values of stack objects are specified by the model data field `contents`; this field does not have to be implemented in Java code, i.e., it is a specification-only variable. The `initially` clause provides an abstract initialisation for `contents`. The class invariant specifies properties that must hold true in each visible state. The three methods, `pop`, `push`, and `top` have their usual preconditions, postconditions, and frames (the latter given by `modifiable` clauses). An omitted frame (e.g., in `top`) indicates that no fields can have their state modified. In postconditions, the syntax `\old(E)` can be used to evaluate expression `E` in the prestate. As well, JML distinguishes between reference equality (`equals`, used in the postcondition of `pop`) and value equality (`==`, used in `top`). This JML specification could be placed in a `.java` file for a Java implementation of an unbounded stack.

The JML constructs that we have seen so far are similar to those provided with BON. JML provides a number of constructs, useful for specifying Java modules, that are not provided with BON, such as *depend clauses* (for expressing field dependencies), *history constraints*, and *exceptions*, which are particularly critical for specifying Java propgrams, where exception handlers are widely used.

Several tools have been developed for JML, including a JML type checker, a JML run-time assertion checker, and a documentation generator [10]. The run-time checker is currently limited to checking preconditions of methods and whether a class invariant holds at run-time. JML is also partially supported by the Extended Static Checker (ESC) [4] (the syntax of ESC/Java is very similar to JML), and by the LOOP tool [2]. LOOP is a special-purpose compiler that incorporates the semantics of Java and JML. The output of LOOP is a set of logical theories for the theorem provers PVS [12] and Isabelle.

JML was chosen for the integration with BON because of its utility in developing Java programs (no guidance is provided with BON for producing Java code from models), because of its existing tool support, and with the intent of leveraging future tool support as well.

2.3 A method for formal method integration

A method for formal method integration was presented in [13, 14]. It describes a precise approach to integrating software development methods, via combining the modelling languages of the methods, and by relating their development processes. At least one of the methods being combined has to be formal, in the sense that its modelling language must have a formal grammar specifying its syntax, and a formal semantics for modelling constructs is provided. The method of [13, 14] is general, in that it can integrate any number of methods. It emphasizes combining modelling languages while providing assistance in linking and combining processes. The usefulness of the method has been validated on a number of integration case studies, combining industrially applicable languages and methods such as UML (with a compatible process), Z, BON, Structured Analysis, and Larch. In turn, these integrations have been applied to a number of case studies in software development.

The method is based on the construction of heterogeneous languages [13], and on defining relationships between tasks in processes. We first give a brief overview of heterogeneous languages and their construction, and then discuss the steps of the method

(which will indicate how relationships between processes can be defined and documented).

2.3.1 Heterogeneous languages and bases Modelling languages play a critical role in software development methods, for writing descriptions of requirements, architectures, designs, and programs, and in documenting the development process. Modelling languages (which consist of a notation as well as structural well-formedness rules, i.e., a grammar or a metamodel) play a key part in how we integrate methods: we combine languages as the first step in combining methods. A *heterogeneous language* is composed from several different languages and is used to write heterogeneous specifications.

Definition 1. A *heterogeneous specification* is composed from parts written in two or more different modelling languages.

As the name suggests, heterogeneous languages are an integration of both notation and metamodel. Heterogeneous languages are useful for a number of reasons: for extending expressive capabilities of individual languages; for producing simpler specification languages [28]; for writing simpler specifications than might be produced using a single language [28]; for ease of expression; and because they have been proven successful in practice. Heterogeneous languages are ideally applied in a setting where the complementarity of the separate languages is evident [17], and when the languages being combined can be used together without significant alteration, in part so as to not alienate users of the individual languages.

A semantics for a heterogeneous specification can be given by formally defining the meaning of the composition of partial specifications in some notation. If we combine two languages, where one is formal and the other informal, we can formally define specification compositions by formalizing the semantics of the informal notation in the formal one. This approach lets specifiers treat the compositions as if they were written in the formal language. More generally, we may want to build a heterogeneous language from more than two formal or informal languages. In this case, we can construct a *heterogeneous basis* [13], a set of languages and translations between languages, which can be used to give a formal semantics to heterogeneous specifications via translation to a homogeneous specification. The language used to provide the semantics for a heterogeneous specification may be one of the languages used to write the specification, or it may be an entirely different language altogether. This allows semi-formal languages, e.g., data flow diagrams or variants of object modelling languages, to be given an appropriate semantics depending on the context in which they are used. This is consistent with the work of Baresi and Pezzé [1] on formalising families of languages, such as those available in Structured Analysis.

Fig. 4 depicts an example of a heterogeneous basis. A preliminary version was first presented in [13]. It has since been extended to include further languages, such as BON [18] and now JML.

In Fig. 4, the arrows represent syntax-directed translations that have been defined between the languages. The arrow $\dashrightarrow$ between languages represents a partial translation. A translation may be partial in one of two senses: there may exist constructs in the source language that are inexpressible – and thus, are not translated – in the target

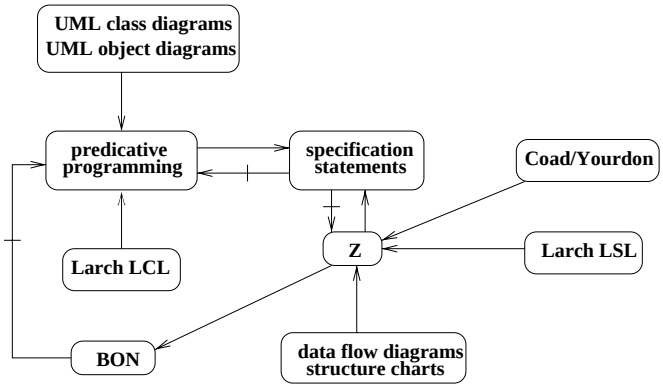


Fig.4. Heterogeneous basis

language; or, a translation of every construct in the metamodel of the source language is simply not presented or currently available. Translations are given in terms of the metamodel of a source language: each construct that appears in the metamodel of the source language is mapped to a construct or set of constructs in the metamodel of the target language. In this manner, translations can be defined for text-based languages (they are expressed in terms of constructs that appear in a context-free grammar), and for visual languages.

Translations between the formal languages are documented in [13, 14]. Mappings from elements of UML to predicative programming are in [16]. A mapping from Z to BON is given in [18]. The basis is extendible (as we discuss in the next section), which is critical in defining a general method for formal method integration.

Constructing a heterogeneous basis is usually hard, and requires careful understanding of the semantics and interpretations of all the constructs in the individual languages, as well as a means to resolve syntactic and semantic incompatibilities and differences in expressiveness across the languages. General discussion on these issues is in [21].

2.3.2 A method for formal method integration We recap the steps of the method here for the sake of completeness.

1. **Ensure complementarity of the methods.** It must be demonstrated that the methods being integrated are complementary. This is done in order to provide motivation and intent for the integration. Patterns of method complementarity are discussed in [17]. Generally, methods may be complementary in terms of their modelling languages, their processes, and their supplementary tools, but other rationales for integration may be provided as well, e.g., for non-technical reasons, such as to best make use of developer expertise and familiarity.
2. **Select a base method.** A base method provides a process that is to be supported and complemented by other (invasive) methods. Selecting a base method is aimed at assisting integrators in determining the roles that the separate methods can play in the integrated approach.

3. **Choose invasive methods.** The processes of invasive methods augment, are embedded in, or are interleaved with that of the base method. The invasive methods are chosen to complement the base methods, through notation, through process, through tools, or through user preferences. The invasive methods may overlap with the base method; that is, the invasive methods may duplicate notations or process tasks provided by the base method. In choosing the invasive methods, it must also be decided how to reconcile these overlaps, for example, by restricting the use of a method, or by synthesizing the documentation deliverables produced.
4. **Construct or extend a heterogeneous basis.** This is accomplished by defining translations, or by adding languages from the base and invasive methods to an existing one. At this point, a *basis language* can be selected. This language is chosen from the heterogeneous basis and is used to provide a formal semantics to heterogeneous specifications. This semantics will be used when reasoning about heterogeneous specifications; thus, the basis language should be chosen to make the desired kind of reasoning as simple as possible to do. Note that the basis language does not have to be fixed permanently once it is chosen; the heterogeneous basis and its translations can be used to change it later on. One might want to do this to be able to make use of new tools.
   It may be determined that the languages being combined overlap syntactically or semantically. The language integrator must decide how to deal with such overlap, e.g., by restricting use of one notation.
5. **Define how the individual processes cooperate.** It is informally described how the processes of the methods are to work together in the new method. Process cooperation can be specified using activity diagrams (an example is in Section 3.5), or workflows. Two forms of cooperation are particularly common.
   - **Generalisation.** The process of the base method is generalized to use heterogeneous notations constructed from those of the base and invasive methods. Effectively, notations are added to an existing method, and its process is generalized to use the new notations.
   - **Interleaving of processes.** Relationships between the process of the base method and the processes of the invasive methods are defined. Examples of relationships include the following.
     - *Linking* of processes, by defining a translation between notations of different methods. An example is in [18], in which a translation from Z to the Business Object Notation links the Z ‘established strategy’ with the BON method.
     - *Replacement* of entire process steps in a base method by process steps of an invasive method. The invariant in such a replacement is that the steps being added must do at least the tasks of the steps they are replacing.
     - *Supplementation* of process. Specific phases of one process are identified and are supplemented by phases from a second process. The integration of Z and predicative programming in [15] provides an example; the Z established strategy is supplemented by refinement rules for reasoning about time and space.
     - *Parallel use* of processes, by describing relationships that interleave the use of two or more separate processes.

6. **Guidance to the user.** Hints, examples, and suggestions on how the integrated method can be used is provided.

The meta-method has been applied in a number of examples. We refer the reader to [13, 18, 16] for instances. Each example integration is also illustrated with an example of applying the integrated method in a non-trivial case study.

3 Integrating BON and JML

We now describe the integration of BON and JML, following the method presented in the previous section. The integration will generalise the BON process, by adding JML to it. To carry this out, we will show how to translate BON to JML, thereby allowing us to combine BON and JML models and to introduce JML into the BON process.

3.1 Ensure complementarity

For any integration of languages or methods, it is critical to justify the complementary nature of the techniques, so as to help justify the usefulness of the integration. BON and JML are complementary techniques in the following ways:

- BON provides a CASE tool-supported graphical language for modelling, whereas JML is text-based. By integrating BON and JML, we provide a tool-supported graphical syntax for JML.
- JML provides richer tool support for reasoning than does BON. Integrating the techniques will allow developers to use the JML type checker, run-time assertion checker, static checkers like ESC/Java, and, eventually, theorem provers like PVS and Isabelle with their BON models.
- JML provides a richer set of modelling constructs for modelling, particularly mechanisms for specifying exceptional behaviour and dependencies. BON provides no such techniques.
- BON is a method, whereas JML is a modelling language. Integrating BON and JML effectively adds process support to JML, thus enriching it to a full software development method.
- By integrating BON and JML, we also provide a method, with tool support, that can help in producing Java programs from BON models: the integration tailors BON for developing Java programs.

3.2 Select a base method

The base method in the integration will be BON, in part because it provides a process that spans requirements analysis, through detailed design, generalisation, and validation. The BON process will be generalised with the use of JML and its tools. Tools for BON will be applied throughout tasks 1-5 of the suggested BON process from Fig. 2, while tools for JML will be used in task 6, and in a new task that will be added later.

3.3 Select the invasive techniques

The invasive technique in this integration is JML. The BON process will be extended to make use of JML specifications, and thereafter JML tools, during task 6 of the suggested BON process.

3.4 Extend a heterogeneous basis

The next step is to combine the languages of the techniques of interest. We will thus add JML to the heterogeneous basis presented in Fig. 4. The extension will occur by defining a rigorous translation from to JML. The details of the translation are quite long, and are presented in full in [7]. We provide an overview here. The translation will be presented in terms of the metamodels of BON and JML.

BON and JML are both based on OO concepts, and are thus founded on the specification of classes, features of classes, properties of features and classes, and relationships on classes. We describe the translation from BON to JML in terms of these four basic specification concepts.

3.4.1 Translating classes All BON classes (except primitives, and generic sets) are translated into JML public interfaces; this is done since a BON class cannot contain feature implementations, only feature specifications. A BON class may be annotated with stereotypes, e.g., deferred, effective, reused, interfaced, persistent, root, etc. These stereotypes are in most cases dropped in translation to JML: the deferred stereotype is captured by the JML notion of a public abstract interface; effective, reused, external, and interfaced stereotypes in BON are given only to aid the reader – these could be and are translated as comments. The persistent stereotype indicates that instances of the BON class persist between executions of the system. We translate this for now by adding a comment to the resultant JML class indicating that instances should persist. This could be implemented in Java using *Serializable*, but this is not the only mechanism that can be used for persistence. The BON root stereotype indicates a class from which execution of a resultant system may begin. This class is mapped to a public interface with a method called `main`.

BON built-in types (e.g., $INTEGER$, $REAL$, $STRING$, $BOOLEAN$) are mapped to their JML equivalents (i.e., `int`, `float`, `String`, `boolean`). The only significant change needs to be made with BON generic sets, i.e., $SET[G]$. These must be mapped to JML's `JMLObjectSet`. This latter type is less restrictive than BON generic sets, in that it allows heterogeneous sets of objects, whereas with $SET[G]$, all objects in the set must be of compatible types. We discuss translating arbitrary generic types in Section 3.4.6.

3.4.2 Translating features A feature in BON is either an attribute (representing part of the state of an object), a query, or a command. Attributes in BON are translated into JML `model instance`s; the keyword `model` indicates that the instance need not appear in an implementation. By default, the translation assumes an attribute is a specification-only variable; the developer can remove this tag if desired. Queries in

BON are side-effect free functions, and thus they are translated into JML `pure` functions. Commands in BON can change the state of an object, but cannot return a value. Thus, they are translated to JML procedures, with a `void` return type.

BON classes may also possess *deferred* features, which are routines that are to be implemented by child classes. These are translated to JML `abstract` methods.

Each feature in BON has a list of accessors, client classes that have permission to call the feature. This list may allow any clients to access the feature (i.e., *public* access), may allow only routines local to the class to access the feature (i.e., *private* access), or may allow certain specifically named clients to access the feature. The former two kinds of access lists are mapped directly to JML's `public` methods and `protected` methods[1]. The last kind of access list is inexpressible in JML and in Java: it is not possible to restrict access to a method to a list of specific classes. Thus, to translate these access lists, we first (a) check that the BON model obeys the information hiding rules (the BON-CASE tool, described in Section 4, guarantees this); and (b) translate the feature to a `public` one, thereby relaxing the specification. This is not ideal, but it is the only simple way to translate such BON constructs to JML. Of course, this effects round-trip engineering – if we were to reverse-engineer a BON model from an automatically generated JML specification, we would lose full export policy information. The integration of BON and JML has effectively suggested to us that when developing Java programs from BON, we should restrict use of information hiding in BON to ANY and $NONE$ access.

3.4.3 Translating properties of features and classes A BON feature may have a precondition and postcondition, and a class may have an invariant. As well, features may possess a frame, indicating the attributes that may be changed by the routine. These constructs exist in JML as well, and thus the mapping is direct: each BON construct is translated to its JML equivalent. This will require some minor syntactic adjustments (e.g., replacing BON **old** expressions with JML `\old()` expressions). A full description of the syntactic rewriting of expressions and assertions is given in [7].

This direct translation is possible, in part, because the semantics of pre- and postconditions in BON is identical to their semantics in JML. It is important to note that the BON semantics for contracts says nothing about the behaviour of a call made in a state that does not satisfy the precondition: any behaviour is acceptable (including nontermination). JML allows specification of exceptional behaviour. As this is a stronger specification of behaviour than is made in BON, it is sound to map BON feature pre- and postconditions to JML method pre- and postconditions.

A BON class invariant can be mapped directly to a JML `public invariant`. The two constructs have the same semantics, with respect to when the invariant must be true, and when it can be (temporarily) false.

Fig. 5 shows the JML translation of the invariant of $CITIZEN$ shown in Fig. 1(a).

3.4.4 Translating relationships BON possesses three kinds of class relationships: inheritance, association, and aggregation. The latter two relationships, which define

[1] We use `protected` instead of `private` because BON's *NONE* access means that while clients cannot access the feature, child classes may make use of it.

```
public invariant: single() || spouse.spouse == this;
public invariant: parents.count == 2;
public invariant: (\forall (Person c) (children.has(c))
                  ==> (\exists (Person p) (c.parents.has(p))
                    ==> (p==this)))
```

Fig.5. JML translation of $CITIZEN$ invariant

has-a and *part-of* relations between client and supplier classes, are translated directly to attributes of the client class. This accurately captures the precise semantics of association (which indicates that the client object refers to one or more instances of the supplier class), but does not adequately capture the semantics of aggregation. With aggregation, a supplier object is part of the "whole" client object: this part cannot be shared, nor can a client exist without its supplier part. Thus, we must add to the invariant of the client the clause $supplier\_object \neq null$. This is still not sufficient. But JML (and Java, for that matter) is not expressive enough to distinguish part/whole and has-a relationships. Thus, developers must take great care when using JML specifications that result from translating BON aggregation relationships that they do not misuse supplier objects, e.g., by allowing multiple references to a subobject. The integration of BON and JML suggests to us that we should avoid use of aggregation in BON, whenever we develop JML specifications or Java programs.

Inheritance in BON defines a subtyping relationship. It is mapped into the JML `extends` relationship. When a feature is inherited, it can be renamed (via BON's **rename** clause) and **redefine** (i.e., its behaviour can be changed). In translating to JML, we automatically apply all renamings that appear in the BON specification. Thus, child classes will use the renamed version of a feature, and this renamed version appears in the interface of the JML child class. Any occurrences of the old name of the feature that appears in contracts will be replaced by the new name.

Redefined features are translated to overridden methods in JML. BON redefinition allows the modification of a contract: preconditions can be weakened and postconditions can be strengthened; this is the strong behavioural subtyping requirement of Liskov and Wing. JML places the same requirements on redefinition. To enforce this "subcontracting", BON allows preconditions to be weakened only by disjoining new assertions (via a **require else** clause), and postconditions can be strengthened only be conjoining new assertions (via an **ensure then** clause). This BON syntax is mapped directly to JML's `also` clauses, which have the same meaning.

3.4.5 Example Here is a small example demonstrating the translation. BON class $EMPLOYEE$ is shown in Fig. 6. It would be translated into the JML specification shown in Fig. 7.

(JML actually allows invariant clauses to appear anywhere in the interface of a class. We include it at the end of the class because it simplifies automatic translation from BON.) In Fig. 6, $gross\_salary$ is exported only to the class $INLAND\_REVENUE$, whereas in the JML specification, it is public. But because the remainder of the JML

class $EMPLOYEE$
feature $\{ANY\}$
$name : STRING$
$age : INTEGER$
feature $\{INLAND\_REVENUE\}$
$gross\_salary : INTEGER$
feature $\{ANY\}$
$net\_salary(tax\_rate : INTEGER) : REAL$
require $tax\_rate \geq 0 \wedge tax\_rate \leq 100$
ensure $Result = gross\_salary - (tax\_rate/100) * gross\_salary$
invariant $age \geq 0 \wedge name \neq Void$
end

Fig.6. Employee in ASCII BON

```
public interface Employee {

   /*@ public model instance String name; @*/

   /*@ public model int age; @*/

   /*@ public_normal_behavior
     @ requires: tax_rate >= 0 && tax_rate <= 100;
     @ ensures: \result==gross_salary-(tax_rate/100)*gross_salary;
     @*/
   public pure float net_salary(int tax_rate);

   /*@ public model int gross_salary; @*/

   /*@ public invariant: age >= 0 && name != null @/
}
```

Fig.7. Employee in JML

specification has been automatically translated from BON, and because we know that the BON specification obeyed the information hiding model, we know that only interface `InlandRevenue`, of all automatically generated clients, will try to use this field. However, a more expressive notion of information hiding, along the lines of BON would be useful in JML (and in Java, for that matter).

We now discuss complications in translating BON to JML, particularly with mapping multiple inheritance, genericity, and covariant feature redefinition.

3.4.6 Complications and expressiveness The potential for problems arises in translating four BON constructs into JML.

1. *Multiple inheritance.* In BON, a class may have one or more parents, thus defining one or more subtype relationships. In JML and Java, multiple inheritance is only permitted on interfaces. Fortunately, a BON class cannot possess an implementation (i.e., only feature specifications can be given), and so each BON class is mapped to a JML interface, and thus multiple inheritance in BON can be translated directly to multiple interface inheritance in JML. This only works because JML permits interfaces to contain fields, whereas in Java only method interfaces are allowed in interfaces.
   We experimented with using automatic delegation, via the Jamie tool [26], for mapping BON multiple inheritance into single inheritance (with use of a private field). This would not work since Jamie requires method implementations in order to apply the delegation design pattern, and JML does not support method implementations. This information can be added to Java method bodies integrated with the JML specifications, but since the JML toolset does not examine method bodies, this is not entirely useful since it will not permit accurate reasoning. When JML and Java are combined, and expressed in PVS using the LOOP tool, then accurate reasoning will be possible.
2. *Generic types.* BON supports generic classes, e.g., $SET[G]$, etc., while JML and Java do not[2]. The most commonly used BON generic classes, e.g., $SET[G]$, can be translated to JML built-in object types, e.g., `JMLObjectSet`. Object types allow heterogeneous collections of data to be stored, and thus they are less restrictive than the BON classes. User-defined generic classes cannot be mapped automatically to JML without user assistance: generic parameters would have to be replaced by use of `Object`, strong typing would be lost, and formerly homogeneous data structures would be translated to heterogeneous data structures. The current implementation of the BON to JML translation allows use of generic classes in BON diagrams, but warns the user that problems will arise when they are translated to JML: generic parameters are not mapped to `Object` types, though this could be implemented easily. Currently, translated generic classes are not accepted by JML tools.
3. *Covariant redefinition.* BON is a covariant language, in the sense that signatures of features can be covariantly redefined as they are inherited: types that appear in a routine signature can be replaced with a subtype. This applies to parameters, result types of functions, and types of attributes. JML and Java are strictly no-variant

[2] Though as announced at JavaOne 2001, Java 1.5 will provide support for generics, and so we expect JML will eventually provide the same.

(with the exception, in Java, of arrays which are contravariant). A developer translating BON to JML may choose not to apply covariant redefinition (which they may very well do if they anticipate implementing their specification in Java). The BON-CASE tool, discussed in the next section, allows developers to covariantly redefine routines. When translating to JML, a covariantly redefined routine is translated to an overloaded method. This does not permit use of polymorphic attachments and dynamic dispatch on the method.

4. *Information hiding.* We have already discussed the differences in the information hiding models of BON and JML. A list of feature client classes that differs from $NONE$ or ANY in BON is mapped to `public` in JML. This is not satisfactory, so we envision that perhaps the JML toolset could be extended to check that the information hiding model is obeyed (i.e., that only permitted clients are accessing methods and attributes), and suitable comments could be embedded in JML specifications.

In constructing the translation from BON to JML, we have had to address the expressive differences – both syntactic and semantic – between the two languages. As a result, due to the similarities between JML and Java, we have captured precisely the core difficulties that will arise when trying to translate a BON specification into JML, and therefore have revealed potential problems in mapping a BON model into Java.

We have not proven the soundness of the translation. This is challenging in part because only a partial formal semantics has been defined for BON (see [19]) and the semantics for JML is still evolving. Soundness will be challenging to establish, given the nature of the two modelling languages, which provide many modelling constructs. Still, we have confidence in the validity of the translation for two reasons.

- We have systematically described how each BON construct is to be translated to JML. While this description is only semi-formal, it is precise enough to give a reader confidence in its soundness. Such a presentation will be useful in later verifications of the translation.
- We have applied the JML checker to a number of translated BON specifications (both specifications we expect to be accepted by the checker, and those we expect to not be accepted). In all cases, the expected results corresponded with the results provided by the checker.

3.5 Define the integrated process

The next step in combining BON and JML is provide an integrated development process that captures how the techniques are to be used together. In general, if we are integrating two methods that each provide their own processes, we will be required to define how these processes interact. This will involve, typically, addition of process steps from one method to another method, replacement of process steps, defining synchronisation points between parallel processes, or defining where one process will leave off during development, and where a second process will commence. However, JML is a modelling language and provides no *de facto* process. Thus, we must explain how the BON process is to be *generalised* to make use of JML.

The BON process was described in Section 2.1.1. Task #6 of the process, defining public features and contracts, is the first place wherein precise assertions are added to features and classes. It is at this step where we choose to extend the process to use JML.

The generalised process is sketched in Fig. 8, using an activity diagram. New tasks have been introduced between defining public features and contracts, and refining the system. In these new tasks, we produce JML specifications from BON (automatically, using the BON-CASE tool described in the next section), and apply the JML checker to them. The checker will provide feedback on the specification. For now, this feedback is limited to syntax and type checking, but eventually it will be possible to reason about JML specifications using theorem provers like PVS and Isabelle. Any feedback provided by the checker that reveals flaws or errors in the specification can be fed back in to the BON specification. The **Refine system** task is an extension of a similarly named task from the original BON process. There are two paths that the new task can generate. In one path, we take the feedback provided by the JML checker and revise the BON model. We can then apply task #7 again, and iterate this process until we are satisfied with the BON specification. At this point, we continue with the standard BON process, factor out common behaviour, complete the system, and generate code – likely in Eiffel. In the second path, we remain within the JML/Java setting. After producing and checking JML specifications, we continue the development by refining the JML specifications to Java code. Thus, the generalised process can effectively help us target two possible implementation languages using BON: Eiffel and Java.

The process we have shown relies on tools for checking JML specifications, and for producing BON models in the first place. We next describe a tool, called BON-CASE, which not only supports the construction of BON models, but also helps to automate task #7 in the generalised process of Fig. 8.

4 Tool Support for the Integrated Method

The BON-CASE tool [7] is an open-source CASE tool for BON. It is implemented in JFC/Swing, atop the GEF framework. The CASE tool (version 1.1) supports the full BON notation, including static diagrams (classes and interfaces, clusters, relationships, and assertions), and dynamic diagrams (objects, messages, and scenarios). Its persistence mechanism supports storing diagrams in the ASCII dialect of BON, XML, and JPEG, amongst other formats (which we discuss shortly). The CASE tool implements a large portion of the BON metamodel, which was formally specified and partly validated in [20]. Thus, BON models produced using the CASE tool satisfy the syntactic well-formedness constraints of BON, as well as a number of lightweight semantic constraints (e.g., that feature calls are made according to the information hiding model). The exact list of metamodel constraints that are implemented can be found in [7].

Fig. 9 gives an overview of the tool's user interface for producing static diagrams. A static diagram, without class interface details, can be produced using one drawing canvas, and when the user desires to add interface details (e.g., feature names, contracts, signatures, etc.) to a class, a further dialog is displayed. The tool also supports class interfaces, using the notation in Fig. 1(a), as well as collaboration and use-case diagrams.

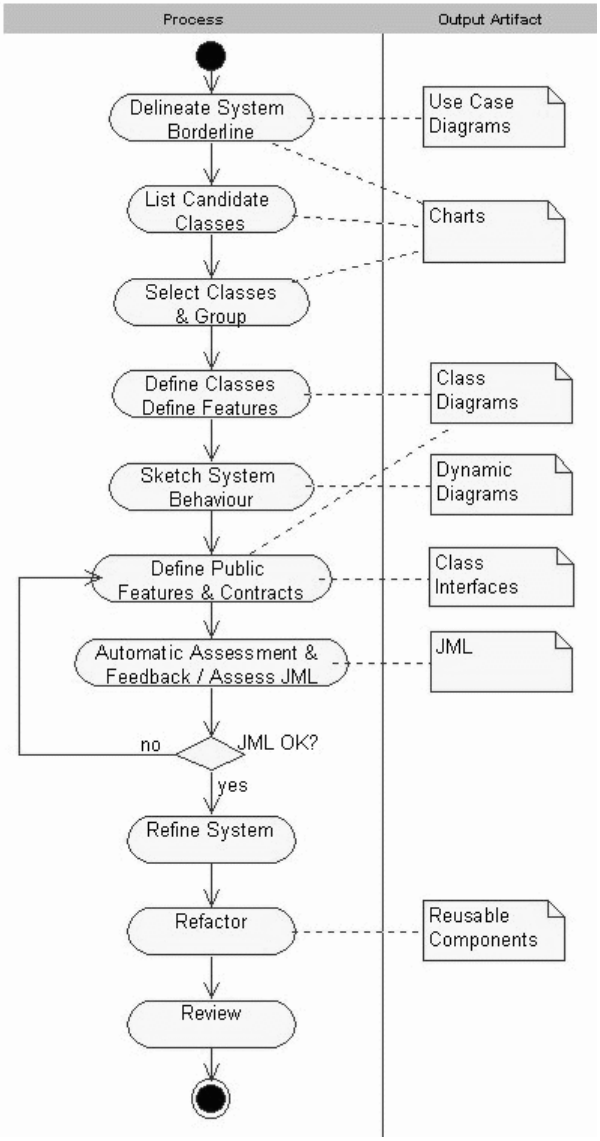


Fig.8. Generalised process using activity diagrams

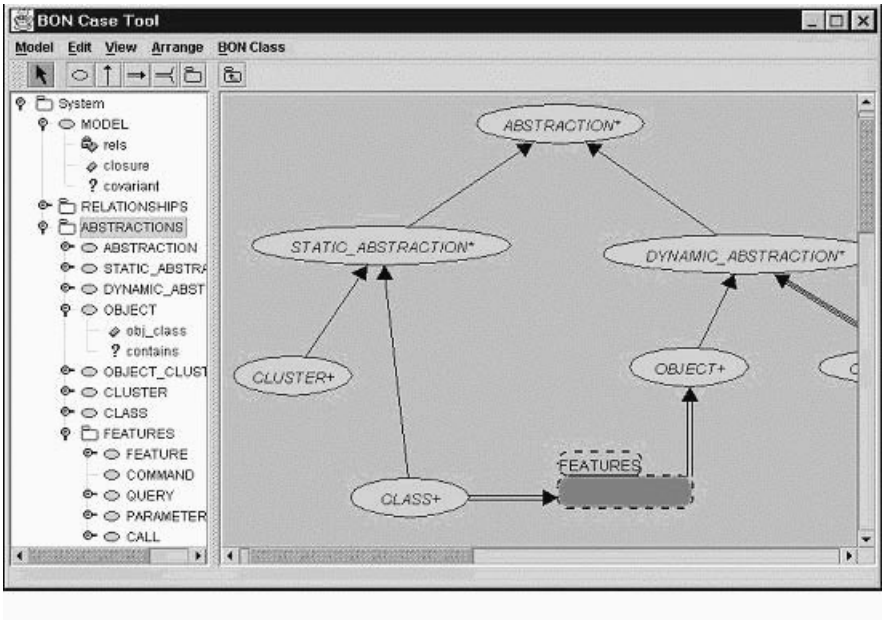


Fig.9. Screen shot of the BON-CASE tool

The abstract package architecture of the tool is shown in Fig. 10. The main components of the tool are the diagram editor, the BON parser (which generates abstract syntax trees), the code generator (which is an abstract interface that is implemented by specialised code generators for target languages) and the metamodel, which encapsulates the well-formedness constraints on BON models.

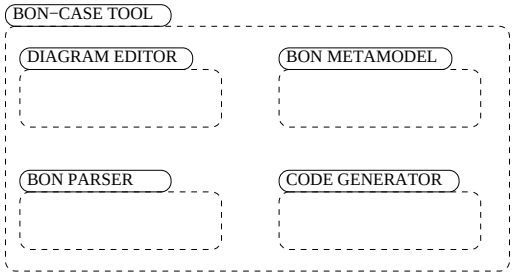


Fig.10. Architecture of the BON-CASE tool

A critical component of the tool is its code generation engine. The code generator, designed using the Template pattern [5], abstracts the code generation process from concrete implementations of abstract syntax tree walkers. Thus, it is straightforward to add new code generators to the tool without affecting the other subsystems. The engine

was recently extended with generators for producing Java and Eiffel code skeletons (assertions in the BON models are transformed to iContract assertions or Eiffel assertions directly). The architecture of the code generator package is shown in Fig. 11. A redesign of this package is currently being considered, based on the use of the Visitor pattern as suggested in [24]. We are considering this redesign in part because the concrete elements (the abstract syntax tree) being accessed by tree walkers are stable and unlikely to change over time, and because the Visitor pattern will lead to easier extension and future tool integration.

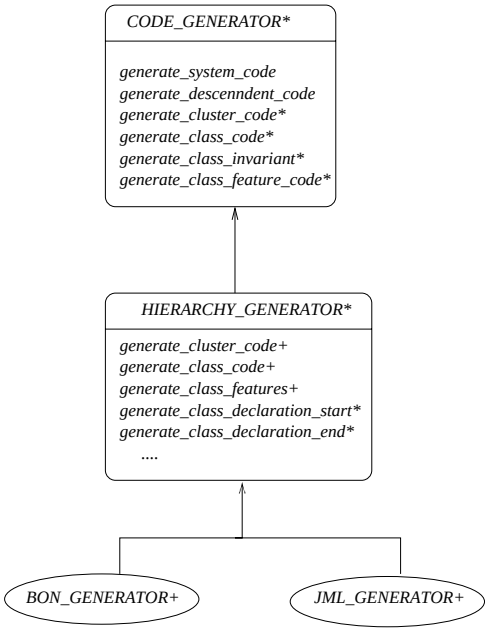


Fig.11. The code generator cluster

BON-CASE provides a code generator for JML, which implements the translation rules discussed in Section 3. The JML code generator is implemented by first generating ASCII BON from a graphical BON model. In this process, an abstract syntax tree is produced. The JML code generator, implemented using the ANTLR tool [22], walks the abstract syntax tree and emits JML code. A number of examples of generated JML specifications can be found in [7]; these automatically generated specifications have been processed and checked by the JML checker.

The BON-CASE tool can be used during the first six tasks of the generalised BON process. The JML code generator can be applied during task #7, and the results of applying the JML checker to the automatically generated JML code can be fed back in to the BON process in task #8. The feedback provided by the JML checker is, of course, given in terms of JML syntax, but the relative closeness of this syntax to that of

BON does not make it difficult to utilize this feedback in modifying the original BON models. Adding reverse engineering facilities to BON-CASE will help here (we are currently working on adding these facilities). We note as well that, at least informally, the semantics of BON and JML have much in common (e.g., in terms of contracts, reference types, classes, etc.) so that we can expect and have received useful feedback from the JML checker about errors or problems with our BON models.

5 Discussion and Lessons Learned

Method, modelling language, and tool integration can be done for a number of reasons: to increase the expressive power of software development languages; to add a process to a modelling language; to generalise, extend, or improve a process through addition of tasks from other processes; to help smooth the adoption of new techniques in a project; and to acquire use of hitherto unavailable tools.

The integration of BON and JML presented in this paper fits under the last category. The primary motivation for defining the translation from BON to JML, and for implementing the translation within the BON-CASE tool, was to be able to use the JML toolset with BON. Previously, BON had only very limited support for reasoning about specifications; this integration provided the means to use JML's existing tools – and, inevitably, future tools – for reasoning. However, there were other reasons for carrying out the integration as well.

- *Targetting Java code.* BON is particularly useful when applied in concert with the Eiffel programming language. However, much recent software development has been carried out with other languages, particularly Java and C++. We desired to make it easier for developers to build Java applications while still making use of BON. Since JML is a modelling language for Java applications, defining an integration of BON and JML provides methodological support for producing Java applications from BON specifications. We mention that the current version of the BON-CASE tool, on generation of Java code from BON models, is capable of generating iContract specifications as well, and it will be easy to extend the tool to generate Java 1.4 code.
- *Graphical views of JML specifications.* JML is strictly an ASCII-based language. For building large systems, graphical views are particularly helpful, and are popular with developers. This integration provides, effectively, a visual front-end for JML. This will be particularly helpful in producing large JML specifications. We point out that the integration and the BON-CASE tool provides a simple way to produce JML interfaces quickly: a BON model is drawn using BON-CASE. Then, JML interfaces are automatically generated. The JML interfaces can then be extended with desired details that are more efficient to add using an editor.

A critical issue in integrating methods and languages is in dealing with differences in expressiveness. BON and JML provide different modelling constructs, some of which are inexpressible in the other language. For example, BON provides an aggregation (part-of) relationship that cannot be directly mapped to JML; it also provides renaming and covariant redefinition that are inexpressible in JML. JML provides `depends`

relations and exceptional behaviour specifications that have no equivalent in BON. In practical terms, we have to decide how to deal with these differences when using the languages together. Our perspective on this issue is as follows.

- It is absolutely critical to know where problems may arise in translating from one language to another. We have catalogued the core constructs in BON and JML that are not directly translatable to equivalent constructs, and which must first be refined in order to be translatable. Given this catalogue, tool builders will know where problems will arise in implementing code generators.
- Restricting use of a modelling language to enable translation is a useful technique, but it is not always possible, nor is it always convenient. Developers should not be forced to restrict use of a modelling language just so as to permit translation. Any method integration technique should allow developers to best select how to deal with differences in expressiveness.
- It must be determined by the developers how to best refine a BON model that cannot be expressed in JML, to a specification that can be translated. Developers will have to rely on their own experience and skill in order to carry this out.

We learned the following important lessons about method integration and tool support for integration techniques in the course of this work.

- If we are not able to prove the soundness of a translation (perhaps due to an incomplete formal semantics for one or both of the languages), then we can be convincing about the translation's reasonableness by presenting the translation in a systematic fashion. This is an appropriate technique for large-scale translations (involving languages with many features) as well as languages with an incomplete formal semantics.
- The Template design pattern is useful in defining an extendible mechanism for loosely integrating tools via file-sharing. The pattern is also an appropriate mechanism for a low-cost implementation of an extendible heterogeneous basis. The pattern was used in implementing the BON-CASE tool's code generator component, and it was found to be useful in extending the tool to automatic generation of Eiffel code as well as iContract-annotated Java code.
- It is typically difficult, if not impossible, to define complete translations between languages that are semantics-preserving, because of the differences in expressiveness of the languages. A semantics-preserving translation may be definable on a subset of a language.
- A useful language translation will be: *structure preserving* (i.e., a homomorphism in the sense of [11]), wherein the translation will be monotonic over language combinators; *refinement preserving*; and *semantics preserving* on an identified (though not necessarily strict) subset of the source language.

We have taken a very pragmatic approach to integration in this paper: we desired to use the JML toolset with BON, and we developed and implemented a translation to effect this. We have not yet proven the soundness of the translation, nor the correctness of its implementation. Our experiments with the JML checker have given us greater confidence in the correctness of the translation. Also, the systematic way in which we

have structured and presented the translation adds to our confidence. It would be beneficial to have a proof of soundness for the translation. This is made more challenging by the size of the BON and JML languages, and the relative imprecision that still remains in the semantics of each language.

A key limitation with the integration – and the implementation in the BON-CASE tool – is the inability to reverse the translation, i.e., to take manually modified JML specifications and reverse engineer a BON specification from it. Currently, changes in JML specifications have to be manually inserted into the original BON specification. The BON-CASE tool currently does not support reverse engineering, though the infrastructure to allow this is present in its design. We are currently defining a reverse mapping, from JML to BON, and plan to implement it along with other reverse engineering facilities in the tool in the near future. We are also experimenting with extending the basis and the tool with further languages, e.g., Object-Z written in LaTeX source, and Object-Z documents expressed in the XML markup of [25]. The latter, in particular, should be reasonably straightforward to implement since the CASE tool already supports generation of XML documents.

References

1. L. Baresi and M. Pezzé. Toward formalising structured analysis. *ACM Trans. Soft. Eng. and Method.* 7(1):80-107, Jan. 1998.
2. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *Proc. TACAS'01*, LNCS 2031, Springer-Verlag, April 2001.
3. G. Booch, J. Rumbaugh, and I. Jacobson. *The UML Reference Guide*, Addison-Wesley, 1999.
4. The Extended Static Checker ESC/Java. Compaq System Research Centre. `www.research.digital.com/SRC/esc/Esc.html`.
5. E. Gamma, R. Helm, J. Vlissides, R. Johnson. *Design Patterns*, Addison-Wesley, 1995.
6. E.C.R. Hehner. *A Practical Theory of Programming*, Springer-Verlag, 1993.
7. L. Kaminskaya. *Combining Object-Oriented Software Development Methods: an Integration of BON and JML*. MSc thesis, Department of Computer Science, York University, May 2001.
8. P. Kruchten. *The Rational Unified Process* (Second Edition), Addison-Wesley, 2000.
9. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: a behavioural interface specification language for Java. Technical Report 98-06i, Department of Computer Science, Iowa State University, Feb. 2000.
10. G.T. Leavens, K.R.M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, ACM Press, October 2000.
11. W. McUmber and B. Cheng. A general framework for formalizing UML with formal languages. In *Proc. International Conference on Software Engineering 2001*, IEEE Press, May 2001.
12. S. Owre, J. Rushby, and N. Shankar. PVS: a prototype verification system. In *Proc. CADE-11*, LNCS 607, Springer-Verlag, 1992.
13. R.F. Paige. A meta-method for formal method integration. In *Proc. Formal Methods Europe 1997*, LNCS 1313, Springer-Verlag, September 1997.
14. R.F. Paige. Pure formal method integration via heterogeneous notations. *Formal Aspects of Computing* 10(3), June 1998.
15. R.F. Paige. Comparing extended Z with a heterogeneous notation for reasoning about time and space. In *Proc. ZUM '98*, LNCS 1439, Springer-Verlag, September 1998.

16. R.F. Paige. Integrating a program design calculus with a subset of UML. *The Computer Journal* 42(2), March/April 1999.
17. R.F. Paige. When are methods complementary? *Information and Software Technology* 41(3), February 1999.
18. R.F. Paige and J.S. Ostroff. From Z to BON/Eiffel. In *Proc. Automated Software Engineering 1998*, IEEE Press, October 1998.
19. R.F. Paige and J.S. Ostroff. Developing BON as an industrial-strength formal method. In *Proc. World Congress on Formal Methods*, LNCS 1709, Springer-Verlag, September 1999.
20. R.F. Paige and J.S. Ostroff. Metamodelling and conformance checking with PVS. In *Proc. Fundamental Aspects of Software Engineering 2001*, LNCS 2029, Springer-Verlag, April 2001.
21. R.F. Paige and L. Kaminskaya. A Tool-Supported Integration of BON and JML. Technical Report CS-TR-2001-04, Department of Computer Science, York University, July 2001.
22. T. Parr and R. Quong. ANTLR: a predicated-LL(k) parser generator. *Software - Practice and Experience* 25(7), 2000.
23. G. Smith. *The Object-Z Specification Language*, Kluwer, 2000.
24. K. Stirewalt and L. Dillon. A component-based approach to building formal analysis tools. In *Proc. International Conference on Software Engineering 2001*, IEEE Press, May 2001.
25. J. Sun, J.S. Dong, J. Liu, and H. Wang. Z Family on the Web with their UML Photos. Technical Report TRA1-01, School of Computing, National University of Singapore, January 2001.
26. J. Viega, P. Reynolds, and R. Behrends. Automating Delegation in Class-Based Languages. In *Proc. TOOLS-USA 2000*, IEEE Press, August 2000.
27. K. Walden and J.-M. Nerson. *Seamless Object-Oriented Software Architecture*, Prentice-Hall, 1995.
28. P. Zave and M. Jackson. Conjunction as composition. *ACM Trans. on Software Engineering and Methodology*, 2(4), 1993.