JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 1, No. 3
Special issue: TOOLS USA 2002 proceedings

BON-CASE: An Extensible CASE Tool for Formal Specification and Reasoning

Richard Paige
Department of Computer Science, University of York, York, U.K.
Liliya Kaminskaya
AMDocs Inc., Toronto, Ontario, Canada
Jonathan Ostroff, Jason Lancaric
Department of Computer Science, York University, Toronto, Canada

We describe BON-CASE, an extensible tool for the BON modelling language. The tool's support for formal techniques – in particular, for formal specification of contracts and frames, as a platform for verification of Eiffel code, for lightweight integration with existing reasoning tools (such as type checkers, theorem provers, and static assertion checkers), and its extensible architecture – is discussed. We present the infrastructure provided with the tool, and support being added to the tool for reverse engineering and the *single model principle*, as well as for checking the consistency of static and dynamic views of a system.

1 INTRODUCTION

Tool support for building high-integrity software systems is of increasing importance, especially given the size and complexity of typical industrial-scale applications, e.g., aircraft navigation or engine control systems. In this domain, tools are of particular importance for supporting *testing* of applications, for *generating code* automatically from abstract models of system properties, and for *reasoning* about correctness and robustness of models and implementations of systems.

The first two characteristics – testing and code generation – are supported by existing CASE tools for object-oriented modelling languages, e.g., Rational Rose (though admittedly, many CASE tools are limited to generation of code stubs from models). The last characteristic – reasoning about models of systems – is well-supported by automated theorem provers such as PVS [16], model checkers such as SMV, and static assertion checkers such as ESC/Java [4] and the JML toolset [12]. In general, however, there is limited support for all three elements in existing tools. One example of this is the U2B macro package [24], which translates UML models in Rational Rose to B abstract machines [1], thus enabling the use of B tools for reasoning indirectly about UML models. This package aims at adding formal techniques to UML-based development, although in doing so it introduces an impedance mismatch between UML's object-oriented models and B's abstract machines. A different approach is offered by the USE tool [22], which provides a simulator for OCL constraints applied to UML models; it does not provide support for the production of executable code.

In this paper, we present and discuss the design and characteristics of the BON-CASE tool. BON-CASE is a CASE tool supporting the BON object-oriented modelling language [25], and which is designed to be *extensible* for the purposes of supporting formal techniques, particularly reasoning via lightweight integration with existing analytic tools, such as theorem provers and static checkers. It differs from related work by its implementation in an open-source tool framework, its design-for-extensibility, and its emphasis on supporting existing formal techniques, rather than addition of formality to semi-formal modelling languages. As such, the tool's design emphasizes the following elements.

- *Support for formal specification.* The tool supports pre- and postconditions of routines, as well as class invariants, written using the BON assertion language, a dialect of first-order predicate logic. The tool provides the infrastructure for supporting automatic verification and reasoning via integration with existing tools.
- *Extensible code generation template.* To allow a variety of tool-supported reasoning mechanisms to be applied to BON models, a code generation template provides an abstract interface to the code generation process. Reasoning tools can be integrated with BON-CASE by implementing the template.
- *Single model principle.* BON supports the single model principle [15], which states that consistent software development deliverables should be ensured. We discuss the principle more precisely in Section 2, and discuss ongoing work towards completing and furthering its support in BON-CASE in Section 5.
- *Partially validated metamodel.* BON-CASE provides a metamodel component, implementing the well-formedness constraints on BON models. The metamodel has been through a partial formal validation process using the PVS theorem prover [18]. This is discussed further in Section 4 and Section 6.

The paper provides an overview of the features of BON-CASE, its design, and its extensible support for formal reasoning. It also provides a detailed discussion on ongoing work on consistency checking of the different views of a system that can be represented in BON (in Section 6).

BON versus UML

The Unified Modelling Language (UML) [3] is a standard language for describing systems. UML is founded on the use of disparate, independently constructed views of a system, including: the static view, describing the classes and class relationships that make up the system; the dynamic view, depicting the objects, collaborations, and messages that are sent between objects; the behavioural view, where state charts are used to describe how an individual object reacts to a message; the use case view, which depicts system behaviour from an external, user-oriented perspective; and the physical view, showing relationships between software and hardware components.

UML, while not designed exclusively for use in modelling high-integrity systems, can be tailored for such systems. This could entail using its constraint language, OCL [26], as well as providing restrictions on the different diagrams to be used, in order to establish consistency of views.

It is beyond the scope of this paper to provide a detailed comparison of BON and UML. We refer the reader to [17], which contrasts the two languages for the purposes of building high-integrity systems. An important question to address, however, is why we chose to support BON in our CASE tool instead of UML.

One answer comes from the intended domain of application. BON has been designed from the start to support formal specification techniques, which have proven to be useful, if not essential, in designing such systems. *Design-by-contract*, the use of which is inherent with BON, is applied throughout the BON process for capturing constraints on conceptual models of the problem domain, and for capturing constraints on designs. UML permits use of design-by-contract, via OCL, but the constraint language is only loosely integrated with the graphical modelling language, and supporting processes for the language do not always make use of OCL. As well, there are syntactic and semantic complications with using OCL (e.g., with respect to automated type checking, the use of flattened collections, etc., some of which are intended to be resolved with OCL 2.0 [14]). Thus, we say that BON supports and emphasizes the use of formal specification, while UML *permits* the use of formal specification techniques.

A second reason for using BON instead of UML in this domain is its support for the single model principle [15]. The principle will be discussed in more detail shortly, but the main impact of it is that it supports checking the consistency of deliverables – i.e., models, code, and testing documentation – produced during development. Consistent deliverables are essential when building high-integrity systems.

2 OVERVIEW OF BON

BON, due to Waldèn and Nerson [25], is an object-oriented method possessing a recommended development process as well as graphical and textual languages for specifying object-oriented systems. It emphasizes the use of formal specifications of classes. The language provides mechanisms for specifying classes and objects, their relationships and interactions, and assertions, written in first-order predicate logic, for specifying the behaviour of routines and invariants of classes. BON is syntactically and semantically compatible with Eiffel: it is designed so that BON class diagrams can be seamlessly and directly mapped [13] to Eiffel programs, providing that assertions are suitably refined and implemented in the Eiffel assertion language. As well, BON is designed to be automatically reversible from Eiffel programs. It has been used successfully with a number of other languages, such as Smalltalk and Java.

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 changes system state but returns nothing. In [25], attributes are treated as parameterless queries without assertions; we distinguish attributes for efficiency reasons, and to make it easier to generate code (particularly JML and Java; see Sections 3 and 5). Fig. 1 contains an example of a BON model for the interface of a class $CITIZEN$.

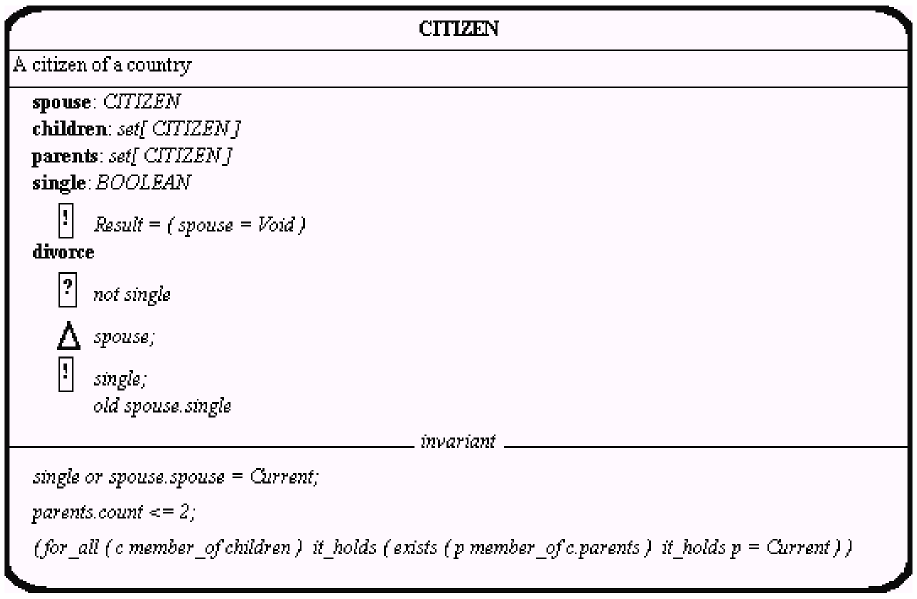


Figure 1: Class $CITIZEN$

Preconditions and postconditions of features are indicated using **?** and **!**, respectively. We have modified the syntax for expressing feature parameters to use Eiffel's notation. As well, we have introduced a notion of a frame – the Δ or **modifiable** clause – to BON class interfaces. The Δ clause specifies a bunch of attributes that may be changed by the feature; attributes not appearing in a frame cannot be changed. The class invariant specifies properties that must be true before and after any client-side call to a feature of the class.

BON class diagrams consist of one or more classes organized in *clusters* (drawn as dashed rounded rectangles that may include classes and other clusters). Classes and clusters interact via two general kinds of relationships. The relationships are drawn in Fig. 2 (the updated syntax for aggregation, also supported by EiffelStudio, is depicted in this diagram, and is supported by BON-CASE).

- **Inheritance:** Inheritance defines a subtyping relationship between a child and one or more parents. It is drawn from class $CHILD$ to class $ANCESTOR$ in Fig. 2.

- **Client-supplier:** there are two client-supplier relationships, association (drawn between $ANCESTOR$ and $SUPPLIER1$) and aggregation (drawn between $CHILD$ and $SUPPLIER2$). Both relationships are directed from a *client* to a *supplier*.

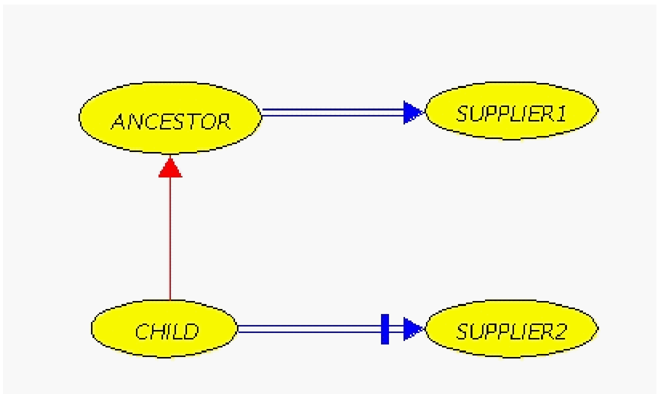
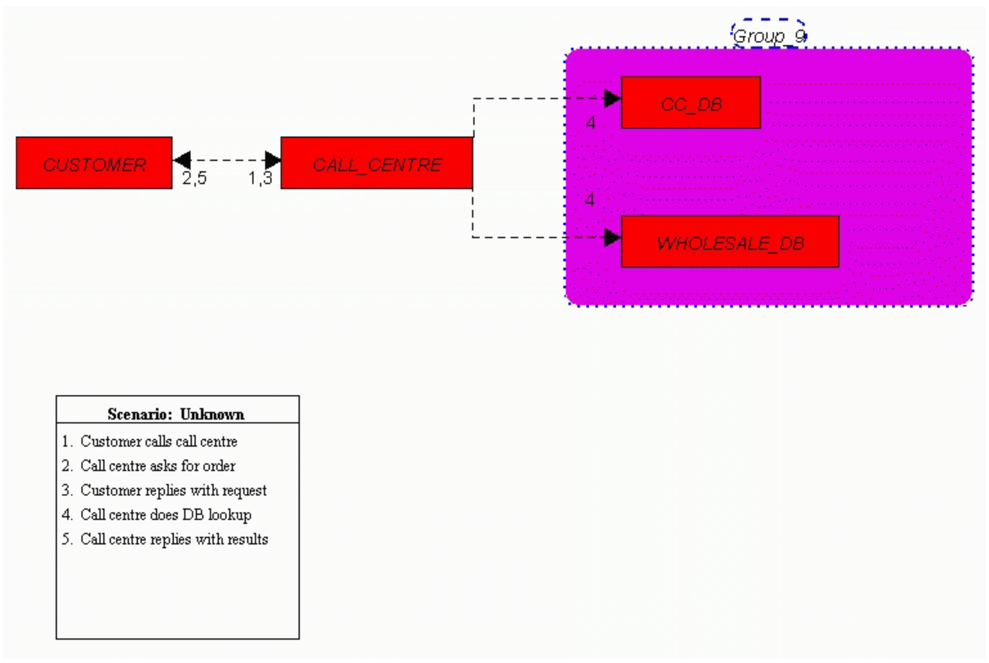


Figure 2: Notation for inheritance, association, and aggregation

BON also provides notation for dynamic diagrams, showing the messages passed between objects, in a manner akin to UML's collaboration diagram. Fig. 3 shows an example, generated using BON-CASE. Numbers that annotate messages are cross-referenced to a scenario box, detailing the purpose of the message. Messages in dynamic diagrams are visual representations of feature calls.



Scenario: Unknown

1. Customer calls call centre
2. Call centre asks for order
3. Customer replies with request
4. Call centre does DB lookup
5. Call centre replies with results

Figure 3: BON dynamic diagram

Single model principle

UML provides five views of a system; these views are each depicted with separate diagrams. This *multiple model* approach can lead to inconsistency: information presented in one diagram may contradict information in another diagram. For example, a collaboration diagram might show an object of type $DATABASE$ receiving a message $enable(n)$ under the condition $n \geq 0$, whereas a class diagram of the same system might show that $DATABASE$ cannot receive $enable(n)$ messages, or that it can only receive such a message under the contrary condition that $n < 0$. Such inconsistencies must be identified and resolved when using UML before the system is implemented and delivered. Some will be captured by the UML metamodel and tools which implement the metamodel, but not all inconsistencies can be caught in this way. The inconsistent guards as described above would not be prevented by an implementation of the metamodel, since to do so would require a theorem to be proven.

BON supports the *single model principle*. The design of BON-CASE provides the infrastructure to implement the principle; work is ongoing on completing this implementation. A full discussion of the principle is beyond the scope of this paper; the report [15] explains it in detail. The basic idea is as follows. As with multiple model approach, languages and methods that support the single model principle also support multiple views of the system. However, the principle aims to ensure consistency of views either by automatic construction or by rigorous analysis. A language that follows the single model principle has several characteristics. It is *seamless* and *reversible*, in that its modelling abstractions can be used throughout the development process and models can be produced automatically from code. It is *wide-spectrum*, applicable to analysis, design, and implementation. It provides *conceptual integrity*: it uses a small number of powerful descriptions that work together to help describe the software product, and it provides one good way to describe every construct of interest. Finally, it provides mechanisms for automatically establishing or checking the consistency of views. The desirability of having a modelling language that satisfies these characteristics when building high-integrity systems has been argued in detail in [15]. Essentially, the argument is that following the principle provides greater assurance and support for building products and having the document deliverables remain consistent.

We discuss existing and planned support for the single model principle in BON-CASE in detail in the sequel.

3 OVERVIEW OF BON-CASE

BON-CASE is an open-source CASE tool for BON. Its design emphasizes the support of *formal specification*, via preconditions, postconditions, and class invariants, as well as *formal reasoning and analysis*, via integration with existing formal methods tools. It is intended to help support rigorous object-oriented analysis and design, particularly for high-integrity systems with substantial robustness and reliability requirements. In this section, we provide an overview of the tool and its features, focusing on its support for formal techniques. In successive sections, we describe its design (aimed at making it straightforward to integrate BON-CASE with a variety of formal analysis tools) and its existing support for reasoning.

Diagrams and user interface

BON-CASE supports the two main types of BON diagrams: *class diagrams* and *dynamic diagrams* (also known as collaboration diagrams). It also supports *use case diagrams*, adopted from UML. Use case diagrams can be applied in several ways, e.g., as *rough sketches* and as abstractions of dynamic diagrams. This is discussed further in Section 3.4.

Fig. 4 shows a screenshot of the main user interface for BON-CASE. The tool provides a typical CASE tool framework implemented in a model-view-controller style. In the left-most view is detailed the contents of a *project*. A project may include several diagrams, including static diagrams, use case diagrams, and dynamic diagrams. There may be several instances of each diagram type, though only one instance of each type may be active for editing. New instances may be added at any time. A project may also include supporting documents, e.g., a to-do list, mockups, test plans. Supporting documents may be text files or JPEG or GIF images. The right-hand view contains details of the currently active project element in the left-hand view.

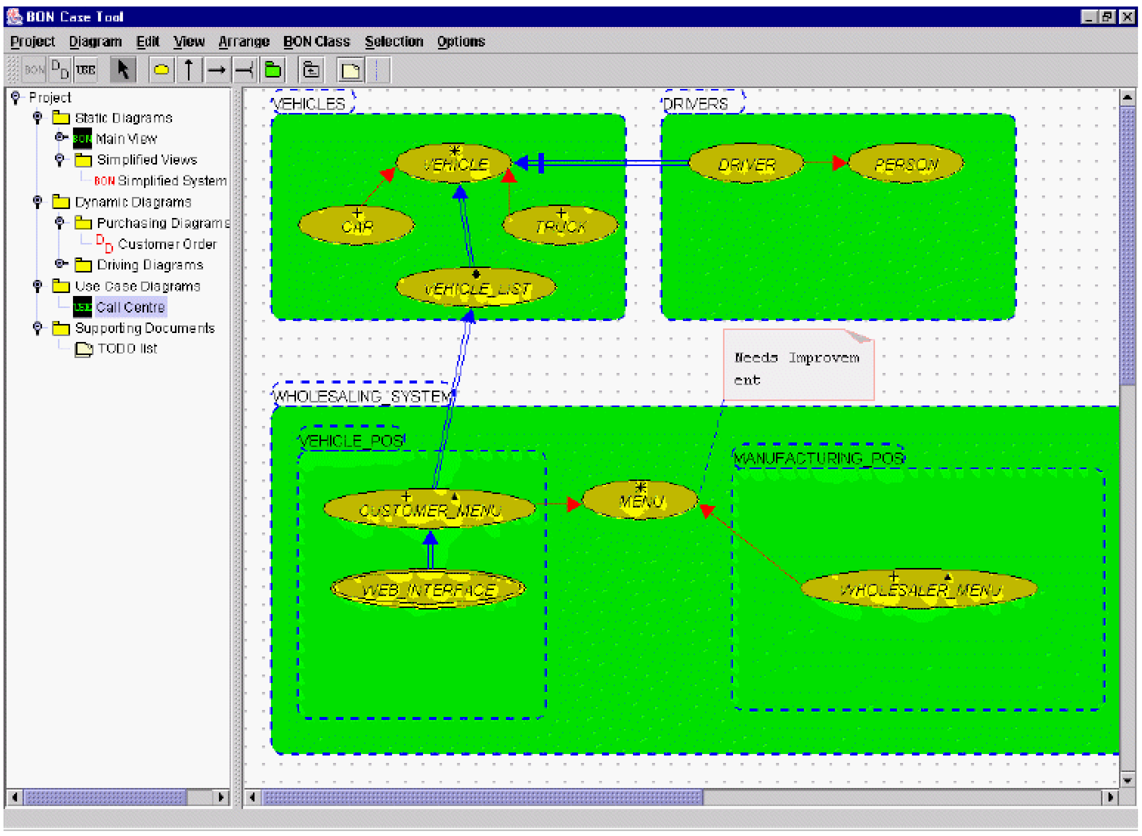


Figure 4: Main user interface for BON-CASE

In Fig. 4, a static diagram is currently active in the right frame, and a use case diagram has just been selected in the left frame. The right frame depicts the compressed BON view of each class, i.e., an ellipse. Clusters (collections of classes and other clusters) are also drawn in a compressed form. All BON relationships are supported by the tool, and relationships can be drawn between classes and clusters. All the typical drawing facilities of CASE tools are supported, e.g., resizing, moving, labelling edges, edge remapping.

Diagrams can include notes, informal comments about the diagram. The note icon on the tool bar is used to add new notes. Existing notes can be edited simply by double-clicking on the note. This will spawn a copy of the Jext editor [9], which can then be used to enter the text to be contained within the note. A similar mechanism can be used to add Eiffel code to classes. The **BON Class** menu item provides an option to add Eiffel code to any BON class. Selecting this option will again spawn a copy of Jext, which provides Eiffel syntax highlighting.

Class specifications

Each class in BON may contain a specification, detailing the attributes, queries, and commands possessed by the class, as well as assertions (e.g., pre- and postconditions and class invariants). This information is included in the BON *class interface*, and it can be added to a class in BON-CASE by using the edit specification facilities. The full BON assertion language, as defined in [25], is supported for writing assertions. These assertions will be used by code generators when automatically producing Eiffel or Java code from the BON diagrams. The full BON assertion language (and thus, class interfaces) are not supported in EiffelStudio.

A novelty with BON-CASE is that it can depict BON class interfaces, both by themselves, and also in static diagrams. For example, in Fig 4, the class $VEHICLE$ could be replaced by its class interface (by using the toggle details feature of the Selection menu). Fig. 1 showed an example of a class interface produced using BON-CASE.

Dynamic diagrams

BON-CASE provides support for BON's dynamic (collaboration) diagrams, depicting objects and the messages sent between them. An example of a dynamic diagram created using BON-CASE was shown in Fig. 3. Single and multiple-receiver messages can be depicted, as can concurrent messages, e.g., in a multi-threaded application. A scenario box will automatically be generated by BON-CASE, documenting the messages that are sent and their order. Consistency between dynamic diagrams and class diagrams is discussed in the sequel.

Use case diagrams

BON-CASE supports a subset of use case diagrams as described in UML [3]. It supports actors, use cases, and basic dependencies between use cases, i.e., *includes* and *extends* relationships. An example of a use case diagram created with BON-CASE is in Fig. 5.

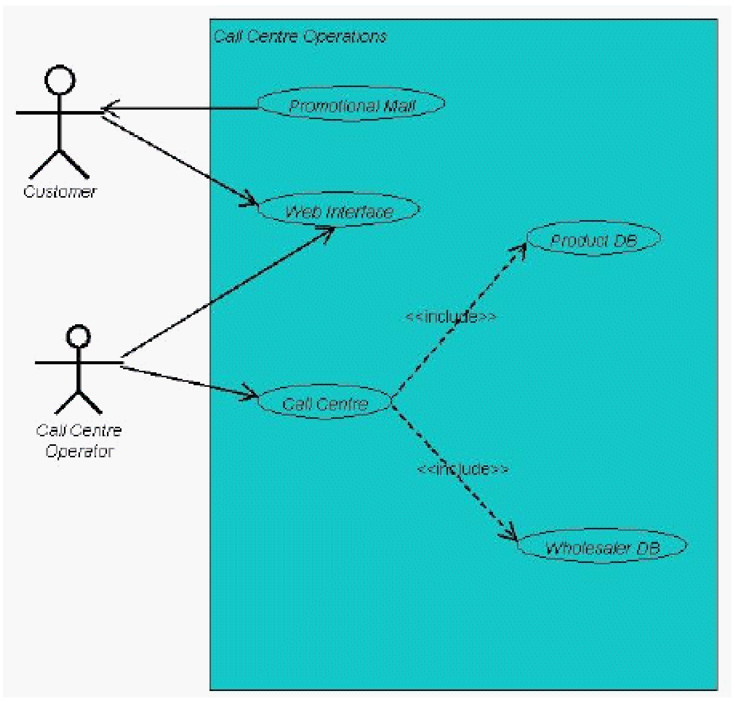


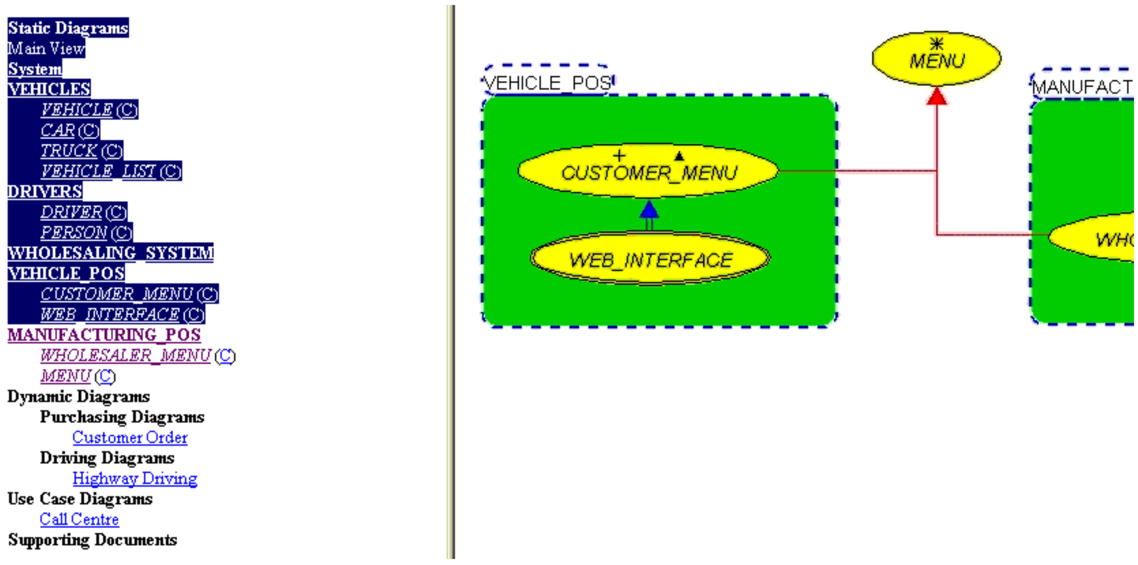
Figure 5: Use case diagram produced using BON-CASE

Documentation generation

A critical feature of any CASE tool is its support for generating browsable, understandable documentation for a project, its models, and their relationships. BON-CASE will automatically produce a suite of browsable HTML files containing the different diagrams associated with a project. Each project will have an index HTML file generated for it, containing links to documentation for each diagram in the project. This is illustrated by Fig. 6.

The documentation is accessible via any browser, and is presented in two frames in a way that attempts to be consistent with the BON-CASE tool interface. Clicking on an item in the left frame brings up the item's details in the right-most frame, in much the same way as BON-CASE.

Both compressed and detailed views of classes are available (the **C** next to the name of each class in Fig. 6 is a link to Eiffel source code for the class).



Static Diagrams
Main View
System
VEHICLES
    VEHICLE (C)
    CAR (C)
    TRUCK (C)
    VEHICLE_LIST (C)
DRIVERS
    DRIVER (C)
    PERSON (C)
WHOLESALING_SYSTEM
VEHICLE_POS
    CUSTOMER_MENU (C)
    WEB_INTERFACE (C)
MANUFACTURING_POS
    WHOLESALER_MENU (C)
    MENU (C)
Dynamic Diagrams
    Purchasing Diagrams
        Customer Order
    Driving Diagrams
        Highway Driving
Use Case Diagrams
    Call Centre
Supporting Documents

Figure 6: Documentation produced using BON-CASE.

Code generation

BON-CASE currently supports the generation of code in Eiffel, Java, textual BON, and JML. We discuss JML generation in Section 5, as it provides a link with formal specification and analysis techniques. The generation of Eiffel from BON is straightforward, except with respect to contracts: some BON contracts – e.g., those that make use of quantifiers over infinite domains – are not executable. The code generator for Eiffel will still produce code for these non-executable expressions. It will be left to the developer to refine such assertions so that they can be executed. We are currently experimenting with Eiffel agents to assist in this process.

The code generator for Java places certain restrictions on which elements of BON can be translated to their semantic equivalents in Java. For example, aggregation in BON has no equivalent in Java; the relationship is mapped to a field in Java. Multiple inheritance in BON cannot be directly translated into Java, except in the case where all parent classes are deferred (i.e., are semantically equivalent to Java interfaces). BON-CASE will attempt to generate Java code, but if the BON static diagram includes multiple inheritance, code generation will fail. We are exploring further proposed mechanisms for mapping multiple inheritance into Java.

The Java code generator will translate BON assertions into iContract [11] assertions. iContract is a preprocessor for Java that provides design-by-contract features, such as pre/postconditions and class invariants. Since these assertions are embedded as comments in the generated Java code, the code is compliant with all existing Java compilers as well as the iContract preprocessor.

4 DESIGN AND IMPLEMENTATION OF THE TOOL

BON-CASE is implemented in JFC/Swing, atop the GEF graph drawing framework [7]. It makes use of the Jext editor for writing notes and Eiffel code. The CASE tool implements a substantial portion of the BON metamodel, which was formally specified and partly validated in [20]. The exact list of metamodel constraints that are implemented can be found in [10]. Some constraints are implemented within the tool's user interface, others as separate routines in a metamodel component that are applied to a BON model as a whole as a model is updated and changed.

The abstract architecture of the tool is shown in Fig. 7. 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.

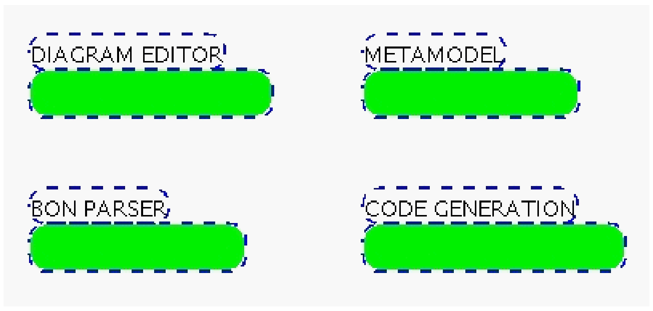


Figure 7: Architecture of the BON-CASE tool

A key component of the tool is its code generation engine. The design of the code generator makes use of the Template pattern [6]; it 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 architecture of the code generator package is shown in Fig. 8. The abstract interface to the code generator defines the process of code generation: that BON classes, interfaces, associations, inheritance, etc., must all be translated in some order. Implementations of this interface provide specific translations in concrete languages. Adding a new code generator resolves to implementing the code generation interface.

The code generation interface provides a lightweight mechanism to loosely integrate BON-CASE with a variety of formal reasoning tools. It should be contrasted with filters as supported in EiffelStudio. Filters are a user-level and syntax-based mechanism for mapping BON into different languages. Our approach requires changing the implementation to add a new code generator, but is more flexible and general, e.g., in supporting semantic checking during translation. In the next section, we discuss one implemented integration, via the Java Modelling Language (JML) [12], and its supporting tools.

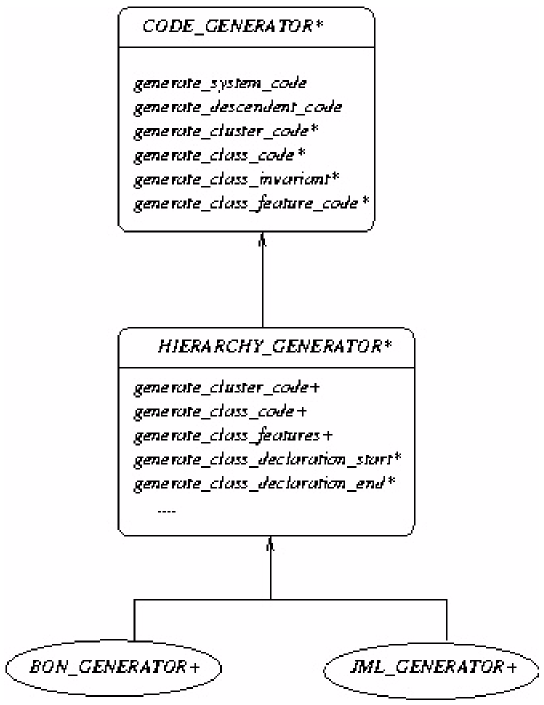


Figure 8: The code generator cluster

5 SUPPORT FOR REASONING

A key requirement in the development of BON-CASE was to provide a framework by which to support formal techniques, particularly specification and reasoning. Formal specification techniques are provided directly by BON. Reasoning techniques are currently provided via integration of BON-CASE with the Java Modelling Language toolset (JML) [12]. This integration is currently implemented via a code generator that translates BON static diagrams into JML specifications.

JML

The Java Modelling Language, JML, is a behavioural interface specification language tailored to Java [12]. 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. Fig. 9 provides an example of an abstract class specification in pure JML: the unbounded stack.

Abstract values of stack objects are specified by the model (specification-only) data field `contents`. The `initially` clause provides initialisation for `contents`. The class invariant specifies properties that must hold true in each visible state. JML distinguishes between reference equality (`equals`, used in the postcondition of `pop`) and

```
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();
   @*/
}
```

Figure 9: JML specification of an unbounded stack

value equality (`==`, used in `top`). The remaining JML constructs used in Fig. 9 are similar to those provided with BON. JML also 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 programs.

Several tools have been developed for JML, including a JML type checker, a JML run-time assertion checker, and a documentation generator [12]. 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], a special-purpose compiler. The output of LOOP is a set of logical theories for the theorem provers PVS and Isabelle, which can thereafter be fed to the provers in order to reason about the JML models.

Translating BON to JML

A translation from BON to JML is defined in [10]. The translation are defined in terms of the BON metamodel: elements in the metamodel are mapped to constructs in the JML context-free grammar. This maps directly to the Template pattern used in the code generator component of BON-CASE, thus simplifying implementation. The fundamentals of this translation are straightforward: BON classes are mapped to JML interfaces; queries are mapped to `pure` (side-effect free) JML methods; commands and attributes are mapped to JML procedures and fields. BON assertions are mapped to their JML equivalents. Associations are mapped to JML fields, and inheritance to JML's `extends`. And redefined features in BON are translated into overridden methods in JML. Complications arise with translating the following BON constructs. In most cases, we have had to compromise preserving semantics in order to allow the BON models to be translated and reasoned about. We also note that many of these complications will arise in mapping BON models to languages other than JML (particularly Java), and in creating profiles for UML.

- *Multiple inheritance.* JML is a specification language for Java, and as such it does not support multiple implementation inheritance. However, a BON class contains no feature implementations, only pre- and postconditions. As well JML interfaces – unlike Java – can include attributes. Thus, if each BON class is translated to a JML interface, then multiple inheritance in BON can be translated to multiple interface inheritance in JML, which is supported.

- *Aggregation.* JML does not support aggregation (part-of) relationships. This BON relationship is translated to a field with the additional invariant clause $\mathit{field} \neq \mathit{null}$ in the client. This is not a semantics-preserving translation (since it allows multiple clients to share supplier parts - thus, it is more akin to aggregation in UML [3]), but it is as close as we can currently get with JML.

- *Generic types.* JML does not support generic types, and so the BON-CASE tool will not translate these constructs correctly. However, it will translate the class $SET[G]$ in to JML's `JMLObjectSet`, which supports heterogeneous sets of objects. We anticipate changes being made to JML with the advent of Java 1.5, which will support generic types.

- *Covariant redefinition.* In BON, features can be covariantly redefined in child classes. JML, like Java, is no-variant. But JML does support overloading of methods based on signatures. Thus, covariantly redefined features in BON are translated in to overloaded methods. This compromise allows the diagram to be translated and reasoned about, but it does not correctly support dynamic dispatch.

- *Information hiding.* JML supports only public, private, and protected features. If a BON feature is not private, then it is translated to a public feature in JML. Selective exports as in BON cannot be expressed in JML.

BON-CASE provides a code generator for JML, which implements the translation rules suggested above. The JML code generator works by first internally generating an abstract syntax tree for a BON model. The JML code generator then walks the abstract syntax tree and emits JML code. A number of examples of JML specifications can be found in [10]; these automatically generated specifications have been processed and checked by the JML checker. The processing carried out has been syntax and type checking, primarily, although we have annotated the generated JML specifications with Java code in order to carry out run-time assertion checking with the JML tool suite as well. Type errors discovered by the JML checker have been assessed and used to manually correct the BON specifications. Thus, we have been able to use the JML checker to analyze our BON specifications, as well as to give us greater assurance that the translation from BON to JML has been implemented correctly.

Currently, the JML tool suite provides a run-time assertion checker, a type checker, and a documentation generator. Ongoing work on JML, in collaboration with the LOOP project and the ESC/Java project, has focused on developing more sophisticated tool support. In particular, work is proceeding on the use of automated theorem provers – particularly Isabelle and PVS – for reasoning about JML specifications. This is being carried out under the auspices of the LOOP project, based on their translation of a subset of Java in to PVS and Isabelle theories. We envision making use of this work for reasoning about specifications generated automatically from BON-CASE as well.

Extension of formal reasoning techniques

Extension of BON-CASE to alternative mechanisms of reasoning resolves to implementing the template for code generation. This defines a translation that will produce an embedding of a BON specification in an alternative formalism. This means providing a small set of classes that define how BON concepts – e.g., classes, associations, inheritance, attributes – are to be represented in the alternative formalism. We are currently working on the definition of translators from BON to Object-Z [23], B machines [1], and PVS, with the PVS translation the furthest along in development.

In [19], a refinement calculus for Eiffel was presented. This calculus lets a developer take a BON class with pre/postconditions, and refine it to an Eiffel class, while producing a proof that the Eiffel program satisfies the BON specification. This provides a mechanism for formally verifying Eiffel programs. A set of proof obligations are defined that would need to be discharged in order to verify that an Eiffel program satisfies a BON specification. We aim to provide support for this process with BON-CASE. Support will be partially provided by the aforementioned code generator for PVS. We envision the proof obligations that arise during a refinement process to be automatically translated in to PVS conjectures, and thereafter the PVS system can be used to automatically or semi-automatically discharge them, or to provide counterexamples. The proof obligations in [19] have been designed, in particular, to lend themselves to automation via PVS.

6 ONGOING WORK, THE SINGLE MODEL PRINCIPLE, AND VIEW CONSISTENCY

The single model principle (hereafter abbreviated as *SMP*) was discussed in some detail earlier. We are attempting to implement the principle within the framework of BON-CASE. This poses a number of research challenges and complexities, and work on supporting the principle should be considered as ongoing. We now discuss some of these challenges, while at the same time discuss ongoing and future work with BON-CASE.

Reverse engineering

A critical element of the SMP is the need to keep code and models consistent. We aim at supporting this in part by providing forward generation of code from models, and reverse engineering of models from code. The former is implemented in the tool, and an alpha release of the tool that implements reverse engineering of BON models from Eiffel code has recently been made available.

Generation of alternative views

Another key element of the SMP is that different views of a system should be consistent by construction, or tools and algorithms should be provided by which consistency can be checked. It may be desirable to support further views of a system via automatic generation, beyond those provided currently with BON-CASE. For example, it may be desirable to provide a finite state machine view in order to depict how an object reacts to a message appearing in a dynamic diagram. We are currently exploring the forward generation of state machines from BON static and dynamic diagrams, focusing on the use of pre- and postconditions to drive the translation. This will likely follow the work of Graham on SOMA [8].

Consistency checking

The previous sub-section discussed the provision of new views of a system via automatic generation, guaranteeing consistency. It is also desirable to be able to check the consistency of existing views. The BON metamodel, currently implemented in BON-CASE, provides a number of rules for guaranteeing that static and dynamic diagrams are consistent. However, the metamodel implementation cannot check all elements of consistency. Consider a static diagram containing a number of classes with pre- and postconditions, and a dynamic diagram depicting a number of messages. A message in BON represents a feature call; thus, for a message to be valid, the receiving object must provide a corresponding feature, and it must be accessible to the client that is sending the message. However, the precondition of the feature being invoked *must be true* when the message is sent; otherwise the message is illegal. The precondition can only be true providing that previous messages that have been sent have left the system in a state satisfying the precondition.

This last type of consistency constraint is not present in the BON metamodel implemented in BON-CASE. To check that a precondition of a feature is enabled requires theorem proving technology in its full generality. In [21], a necessary and sufficient condition for static and dynamic diagram consistency (where static diagrams include pre/postconditions) is presented, and it is shown how to prove that the condition holds using the PVS theorem prover [16]. This latter work shows how to prove the consistency of a suite of static and dynamic diagrams by generating a collection of PVS theories.

7 CONCLUSIONS

We have given an overview of the functionality and design of BON-CASE, an extensible and powerful CASE tool for producing BON models and for supporting formal reasoning about such models. In particular, BON-CASE currently supports a lightweight integration of BON with JML, so as to effect use of JML's formal techniques and tools for reasoning. In integrating BON-CASE with JML, we encountered a number of challenges, which will arise when integrating the tool with further formal techniques. The key problem that we encountered with providing formal reasoning techniques for BON was in precisely defining the translation from BON to JML. Two points are worth noting about such translations, since these issues will arise when integrating new formal techniques with BON-CASE.

- It is typically difficult to define complete translations between specification languages that are also semantics-preserving, because of the differences in expressiveness of the languages. A semantics-preserving translation may be definable on a subset of a language (as is the case with mapping BON to JML). By presenting the translation systematically and exhaustively, we can provide greater confidence that the translation is actually sound. It is also advantageous if we can make use of tools, e.g., checkers, simulators, and theorem provers, to help validate the translation by analyzing its results.
- A useful language translation will be:
  - *semantics preserving* on a not necessarily strict subset of the source language
  - *refinement preserving* [5]. That is, if in a source language a specification A can be implemented by a specification B, then the translation of A should be implemented by the translation of B as well. The translation of BON to JML preserves refinement at the level of methods of classes.
  - *structure preserving*: the architectural style of a specification in the source language should be preserved in the translation. The translation of BON to JML is structure preserving.

A limitation with BON-CASE and its support for formal reasoning is the inability to reverse the translation, i.e., to take 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. We are currently defining a reverse mapping, from JML to BON, and plan to implement it once the reverse engineering of BON models from Eiffel programs is stable. In terms of further support for formal techniques, we are also aiming to extend the tool with further target languages, e.g., Object-Z documents expressed in XML. The latter, in particular, should be straightforward to implement since the CASE tool already supports generation of XML. As well, we are considering how to apply this work to UML. Because the BON-CASE tool provides a separate package implementing presentation style, and a further package for implementing the metamodel, it should be possible to produce a version of the tool applicable to UML modelling – or which can generate UML diagrams as a view of the BON models – and automatic generation of JML specifications, as well.

REFERENCES

[1] J.-R. Abrial. *The B Method*, Cambridge Press, 1996.

[2] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML, *Proc. TACAS 2001*, LNCS 2031, Springer-Verlag, 2001.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The UML User Guide*, Addison-Wesley, 1999.

[4] D. Detlefs, K. Leino, G. Nelson, and J. Saxe. Extended Static Checking, SRC Research Report 159, Compaq Research, 1998.

[5] C. Fischer. How to combine Z with a process algebra, *Proc. Z User Meeting 1998*, LNCS 1493, Springer-Verlag, 1998.

[6] E. Gamma, R. Helm, J. Vlissides, and R. Johnson. *Design Patterns*, Addison-Wesley, 1995.

[7] GEF Project Group. GEF: Java Library for Connected Graph Editors. http://gef.tigris.org, 2001.

[8] I. Graham. *Requirements Engineering and Rapid Development*, Addison-Wesley, 1998.

[9] R. Guy. Jext 3.0 User Manual. http://www.jext.org, 2001.

[10] L. Kaminskaya. *Combining Object-Oriented Software Development Methods: an Integration of BON and JML*, MSc Thesis, York University, May 2001.

[11] R. Kramer. iContract - the Java Design by Contract tool. *Proc. TOOLS USA 1998*, IEEE Press, 1998.

[12] G. Leavens, A. Baker, and C. Ruby. Preliminary Design of JML, Technical Report 98-06j, Department of Computer Science, Iowa State University, revised May 2000.

[13] B. Meyer. *Object-Oriented Software Construction*, Second Edition, Prentice-Hall, 1997.

[14] OCL 2.0 Revision Working Group. Response to the UML 2.0 OCL RfP, Revision 1.3, http://www.omg.org, March 2002.

[15] J. Ostroff and R. Paige. The Single Model Principle, CS-TR-2001-06, Department of Computer Science, York University, August 2001.

[16] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. The PVS Language Reference 2.4. Computing Science Laboratory, SRI International, September 2001.

[17] R. Paige and J. Ostroff. A Comparison of BON and UML, *Proc. UML 1999*, LNCS 1723, Springer-Verlag, 1999.

[18] R. Paige and J. Ostroff. Metamodelling and Conformance Checking with PVS, *Proc. Fundamental Aspects of Software Engineering 2001*, LNCS 2029, Springer-Verlag, 2001.

[19] R. Paige and J. Ostroff. ERC: an Object-Oriented Refinement Calculus for Eiffel, CS-TR-2001-05, Department of Computer Science, York University, August 2001.

[20] R. Paige and J. Ostroff. A Proposal for a Lightweight Rigorous UML-Based Development Method for Reliable Systems, *Proc. Workshop on Practical UML-Based Rigorous Development Methods*, LNI-7, German Society, 2001.

[21] R. Paige, J. Ostroff, and P. Brooke. Checking the Consistency of Class and Collaboration Diagrams using PVS, *Proc. Rigorous Object-Oriented Methods 2002*, BCS, March 2002.

[22] M. Richters and M. Gogolla. Validating UML Models and OCL Constraints, *Proc. UML 2000*, LNCS 1939, Springer-Verlag, 2000.

[23] G. Smith. *The Object-Z Specification Language*, Kluwer, 2000.

[24] C. Snook and M. Butler. Using UML class diagrams for constructing B specifications, Technical Report, Department of Electronics and Computer Science, University of Southampton, 2000.

[25] K. Waldèn and J.-M. Nerson. *Seamless Object-Oriented Software Architecture*, Prentice-Hall, 1995.

[26] J. Warmer and A. Kleppe. *The Object Constraint Language*, Addison-Wesley, 1999.

ABOUT THE AUTHORS

Richard Paige (paige@cs.york.ac.uk) is a lecturer at the University of York, York, UK, where he works with the High-Integrity Systems Group and is a co-leader of the Software and Systems Modelling Team (with Andy Evans). He completed his PhD in Computer Science at the University of Toronto in 1997.

Liliya Kaminskaya (liliya.kaminskaya@amdocs.com) is a software engineer at AMDocs Toronto. She completed her MSc in Computer Science at York University, Toronto, in 2001.

Jonathan Ostroff (jonathan@cs.yorku.ca) is an associate professor at York University, Toronto, Canada, where he leads research on object-oriented design, formal methods, and real-time software development.

Jason Lancaric (jlancar@cs.yorku.ca) is a software engineer working on the BON-CASE project. He completed his BSc in Computer Science at York University, Toronto, in 2001.