BON Development Tool

Ali Taleghani
Jonathan Ostroff
York University, Toronto, Canada
{alitaleg, jonathan}@cs.yorku.ca

Abstract

Modeling languages enable developers to view large systems via diagrammatic notations, which can increase understanding. We present the plan for an Eclipse plug-in that is a CASE Tool for BON. Our tool supports class and collaboration diagrams, fully automated forward and reverse engineering and additionally ensures the consistency between separately constructed class and collaboration diagrams.

1 Introduction

Modeling and design methodologies such as UML [1] have been receiving a lot of attention since they offer important modeling facilities by using diagrammatic notations. Using models can give developers a quick and easy-to-grasp overview of a complex system, but it has also become apparent that the inability to express finer details about a system is a limitation. Textual languages have therefore been used to enrich modelling languages. The Object Constraint Language [2] can, for example, be used in conjunction with UML to describe constraints within OO models.

Further, different views of a system give the developer different perspectives of the same system [3]. Having different views of one system, however, introduces a new problem. Consistency checking between views of a system is necessary to ensure that information described in one model is not contradicted by information described in another model [4].

In this paper, we introduce an Eclipse [13] plug-in called BON-Development-Tool (BDT), which is a BON [5] CASE Tool. BDT allows the construction of class diagrams and collaboration diagrams. It further integrates tightly with the Eiffel-Development-Tool [6] and will allow for seamless forward and reverse engineering. Finally, BDT will enable its users to check the consistency of information presented in dynamic and static diagrams.

BDT is designed to help programmers at every step of the software development process. Forward engineering capabilities allow the developer to start with the construction of class models and the tool will generate program code automatically. Automatic reverse engineering on the other hand, constructs BON class diagrams from program code. As a result, code and model are kept in-sync at all times. Once the user has a better understanding of the system in construction, collaboration diagrams can be constructed. Finally, when (partial) static and dynamic diagrams have been produced, the programmer can use the tool to confirm that the two separately constructed views are consistent and that the specified message sequence can be executed.

We are currently in the process of developing and completing BDT. We have completed the graphical part of BDT and are concentrating on forward and reverse engineering and consistency checking of our tool.

This paper is organized as follows. In Section 2, the BON model is outlined and small examples of static and dynamic views are given. Section 3 outlines the main features and strengths of BDT. Section 4 introduces the notion of consistency between static and dynamic BON diagrams and explains how we plan to accomplish this task. Finally, conclusion and future work follow.

2 BON

BON is a modeling language, which enables the user to specify classes, objects and their relationships within an object-oriented environment. In addition to relationships, assertions (written in first-order predicate logic) can be used to specify the behavior of routines and invariants of classes [4].

There were several reasons for developing a CASE tool for BON. First, it was important to develop a tool that integrates tightly with the Eiffel-Development-Tool [6]. This, however, does not mean that BON (and BDT) cannot be used with other OO programming languages. Second, it provides support for static and dynamic views of a system – the two views we are interested in. Third, BON supports the notion of pre- and postconditions for routines (a routine is the equivalent of a Java method) and class invariants without the need to add any secondary textual language. Finally, if used in an educational environment, we believe that BON is simpler and easier to understand than other modeling languages.

2.1 BON Static Model

The fundamental construct in a BON static diagram is a class. A class has a name, an optional class invariant and zero or more features (features can be attributes, queries or commands) [5]. Classes can be viewed in two forms: In the compact view a class is represented by an ellipse with the class name in the center of the ellipse. The second view consists of a rounded rectangle with all features and invariants of the class visible. Figure1 shows both views. Classes can be further organized in clusters, which make construction of larger systems easier.

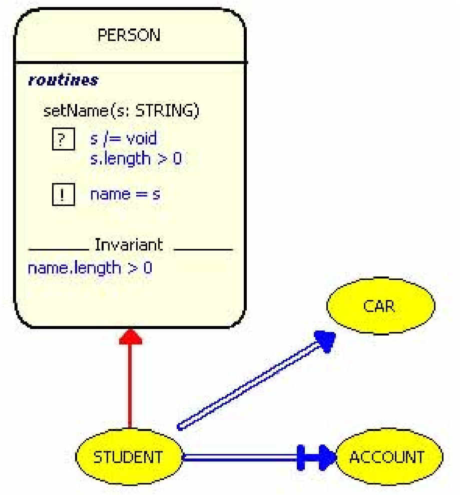

Figure 1: Screenshot of a Class View in BDT.

Classes and clusters interact via two kinds of relationships:

1) **Inheritance:** Inheritance is simply defined as the inclusion in a class, called the *child,* of operations and contract elements defined in other classes, its *parents* [5]. Inheritance may translate differently depending on the object-oriented language used, so the definition is kept very general in BON. Inheritance in BON is represented as a single line with an arrow towards the super class.

2) **Client-supplier:** The user has two options for a client-supplier relationship, association and aggregation. Association refers to reference relationships and aggregation to sub object (subpart) relationships [4]. Association relationships are represented by a double line with an arrow towards the supplier (e.g. CAR in fig1). The representation of aggregation is similar to that of association, but with an additional perpendicular line close to the arrow (e.g. ACCOUNT in fig1). Figure1 shows all three possible relationships.

2.2 BON Dynamic Model

BON also provides notation for collaboration diagrams, which show the communication between objects. Collaboration diagrams consist of rectangles representing runtime objects and arrow lines between them representing messages sent from one object to other object(s). An example of a BON collaboration diagram is given in Figure2. Messages are numbered for two purposes: First, they represent *time* in the scenario - that is the order in which calls are made. Second, they correspond to entries in a *scenario box* where the role of each call may be described using free text [5]. Each message corresponds to a feature call.

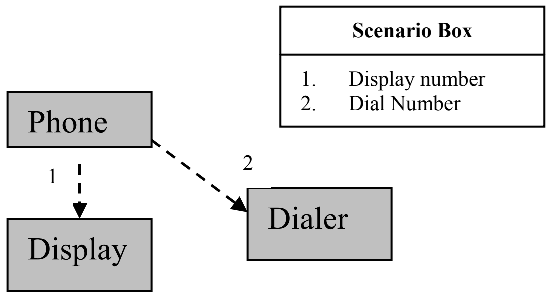

Figure2: Screenshot of a Collaboration Diagram in BDT

3 Overview of BDT

In this section we will provide an overview of our tool and its most important features. Many of these features have already been implemented and others we are currently working on.

3.1 BDT Structure

BDT provides the environment to construct class diagrams and collaboration diagrams. As a standalone, diagramming plug-in, a BDT-project can include several dynamic and static diagrams. As part of EDT (or JDT [7]), only one class diagram can be modified at a time. This ensures that code and class diagrams are always consistent. The user can choose which classes and clusters are shown in the class diagram at any time.

3.2 Static Diagrams

BDT allows the user to construct class diagrams with any level of detail. When a new class is added to the diagram, the user has to specify the class name only. Next, the user can, using an easy-to-use interface, enter additional information about a class. This information can include class invariants, comments and features (attributes, commands and queries). For commands and queries, the user can further specify pre- and postconditions and *modifies* clauses.

An important advantage of BDT is its information hiding capabilities. Classes can be shown in the compact or expanded view. Further, expanded views can be configured to hide any features, invariants or comments that are not needed (these will not be deleted, but just hidden from the view). In figure1 for example, the user might choose not to show the invariant and can hide it easily.

3.3 Dynamic Diagrams

BDT also provides support for constructing collaboration diagrams showing object-interactions. For each object, the name of the object must be specified. Arrows represent messages and each message has a unique number. Numbers are placed on each arrow closer to the receiver of the message. Messages can be added, deleted and edited using the scenario box.
Consistency between dynamic and static diagrams is discussed in the sequel.

3.4 Forward and Reverse Engineering

One of the main goals of this project was to enable the developer to have consistent code and static diagrams. It was important to us to have code automatically generated from diagrams and vice versa. Generation of Eiffel code from BON is straightforward. Following the model-control-view paradigm in designing our system made this task easier. Each model of a class is inspected and the corresponding code is generated. We have designed our tool to detect as many errors as possible during the construction of the diagram and therefore decrease the dependency on the parser and compiler. Constructing diagrams from code is slightly more challenging since there are many constructs in a programming language such as Eiffel that will not be used in diagrams (diagrams abstract from the details) and need to be projected out.

Forward and reverse engineering to and from Java using BON places more restrictions since certain elements in BON do not have a semantic equivalent in Java [8]. Aggregation and multiple- inheritance are two of the major differences. Aggregation can be modeled by a simple reference, but multiple-inheritance cannot be directly translated into Java, except in cases where all super classes are deferred (interfaces). Currently, we are investigating the Eclipse Modelling Framework (EMF) [9] for use in this step. EMF allows the generation of code from Rational Rose model files, annotated Java interfaces, or an XML schema definition [9].

4 Checking for Consistency

Collaboration diagrams are predominantly constructed independently of class diagrams and are often simply assumed to be consistent [3]. We are interested in providing the developer with a tool that detects inconsistencies in the two views. Our method mainly relies on BON's notation, which allows the specification of pre- and postconditions.

An inconsistency can arise in a number of ways (discussed below), but this paper mainly discusses one source. The final tool will check for all sources for inconsistency, but it is important to realize that our tool is not complete. It is possible to miss inconsistencies that exist in the diagram and code. On the other hand, BDT is sound and inconsistencies reported are "real" inconsistencies.

In the next sections we provide a short overview of what we think are the main issues when checking for consistency and how we plan to tackle these problems.

4.1 Checking Collaboration Diagrams Against Class Diagrams

Our goal is to be able to check the consistency of one or more BON collaboration diagrams against one BON class diagram. In addition, we want to report where an inconsistency arises if the tool detects one.

There are several steps when checking the consistency between BON collaboration and class diagrams [4]:

1. The diagrams is syntactically correct
2. Each object appearing in the collaboration diagram has a corresponding class in the class diagram
3. Each message in the collaboration diagram has a corresponding routine
4. The sender of a message must be the client of the receiver (supplier) in the class diagram
5. The routines that are called (from 3) must be enabled, i.e. their preconditions must be true. The precondition can only be true if the sequence of previous calls has established a system state that satisfies the precondition.

As mentioned above, we will concentrate on Point5, as it seems to be the most challenging piece in checking for consistency. The next section provides an overview of our intended approach.

4.2 Checking Feature Preconditions

In order to ensure that a specified message sequence in a dynamic diagram is consistent with the static diagram two conditions must be checked [4]:

1) A developer specified initial state, *init,* must satisfy the precondition of the first message
2) For all Messages Msg(i) for i > 1, the state created by *init* and messages Msg(1) to Msg(i-1) must satisfy the preconditions of Msg(i).

The tool will first verify that *init* satisfies the precondition of the first message and then traverse through the remaining messages. For each message it will ensure that the current system state satisfies the precondition of the message. Every time a message is executed, its postcondition will alter the current system state. As a result, the current system state is a sequential composition of all postconditions of messages that have been executed.

Sequential composition is defined as follows [4]:

$$P;Q = \exists s' \bullet P[s := s'] \wedge Q[olds := s']$$

where s' is an intermediate state.

Let *#messages* represent the number of messages in the dynamic diagram, *msg(i).pre* and *msg(i).post* represent the pre- and postconditions of message *i* and *init* the initial state. Then our tool must check the following condition:

$$\forall i > 1 \wedge i \leq \#messages \bullet$$
$$(init; msg(1).post; \ldots; msg(i-1).post) \rightarrow msg(i).pre$$

The sequential composition and resulting textual substitution ensure that the transition of variables to different values as messages are sent is reflected in the Boolean expression that will be passed to the theorem-prover.

Currently we are experimenting with simple systems and adding complexity as we go along. Central to the consistency checking of our system is the automatic theorem-prover Simplify [10]. An important property of this theorem-prover is that it is refutation-based: to prove a formula P, it tries to satisfy the negation of P [10]. This important feature can be used to provide valuable feedback to the user. Another important advantage of Simplify is that it is available for various platforms as is Eclipse. We have had good results with Simplify, but it is possible that we switch to other theorem-provers such as PVS [11] or Eves [12] if we find serious limitations in Simplify.

4.3 Limitations of Consistency Checking

As mentioned above, we are currently working on simple class structures and are exploring all aspects associated with determining inconsistencies. This section provides some inside into the difficulties we anticipate and partial solutions we might have.

- One important issue is the way feature calls – "dot notations"- are treated in pre and postconditions (e.g. account.balance > 400). The difficulty arises since the feature being referred to belongs to a different object (i.e. is not an internal attribute or query). As a result, whenever a feature is encountered – whether internally or called within another class – it is replaced by the name of the class and the feature name. For example, imagine class A has feature *f1* and class B has an attribute *a* of type A and somewhere in class

B *a.f1* is used in a pre- or postcondition. In this case, we will replace *a.f1* by *A.f1*. A similar substitution is performed in A. As a result, whenever *f1* is used it appears as *A.f1*. This solution only works with one runtime object per class. We are investigating other solutions.

- Contracts involving feature arguments pose a major hurdle for our tool. For example, consider the routine *set_age(i: INTEGER)* with the precondition *(i >= 0),* which enforces that the supplied argument is greater equal 0. When this routine is called the value of *i* will not be specified and our tool will not be able to check the specified precondition. One solution is to assume preconditions that involve arguments (assume they are true) and ultimately display to the user which values can be used for the arguments in order to satisfy the preconditions.

5 Conclusion and Future Work

We presented a BON CASE Tool plug-in for Eclipse that allows the user to construct class diagrams and collaboration diagrams. The tool further supports seamless forward and reverse engineering and maintains consistent code and class diagrams. Finally, formal method support allows developers to ensure that separately constructed static and dynamic diagrams are consistent. We have outlined the path we are planning to take for this consistency checking and the difficulties we anticipate.

Our future work will consist of completing BDT and improving its features. We plan to make consistency checking as automatic as possible and easy whenever user interaction is required. Further we would like to investigate the integration of BDT with JDT. EMF [9] seems to be a very promising and powerful feature of Eclipse that could make this task easier.

About the Authors

Ali Taleghani is a graduate student at York University. His main interests are the use of formal methods in OO and MDD development.

Jonathan Ostroff is an associate professor of computer science at York University. He is interested in software engineering and the use of formal methods for designing real-time and object-oriented systems.

Acknowledgements

The authors would like to thank IBM and the Eclipse Innovation Grant committee for providing the funding that mostly enabled this project.

References

[1] Rational Software et. al., Unified Modelling Language (UML) version 1.3; http://www.rational.com/uml, June 1999.

[2] J. Warmer, A. Kleppe, OCL: the constraint language of the UML, Journal of Object-Oriented Programming 12 (2) (1999) 10-13.

[3] A. Formica, H. Frank, Consistency of the static and dynamic components of object-oriented specifications, Data and Knowledge Engineering 40 (2002) 195-215.

[4] R. Page, J. S. Ostroff, P. Brooke, Checking the Consistency of Class and Collaboration Diagrams using PVS, Proc. Rigorous Object-Oriented Methods 4, British Computer Society (2002).

[5] K. Walden, J.-M. Nerson, Seamless Object-Oriented Software Architecture, Prentice Hall (1995).

[6] D. Makalsky, Eiffel Development Tool, http://www.sourceforge.com/edt (2003).

[7] IBM, Java Development Tool, http://www.eclipse.org/jdt (2001).

[8] R. Page, L. KaminskayaXS, J.S. Ostroff, J. Lancaric, BON-CASE: An Extensible CASE Tool for Formal Specification and Reasoning, Journal of Object Technology, vol. 1 no. 3, Special issue: TOOLS USA 2002 proceedings, 77-96, http://www.jot.fm/issues/issue_2002_08/article5.

[9] Catherine Griffin, Eclipse Modeling Framework, IBM, http://www.eclipse.org/articles/index.html (2002).

[10] Compaq, SIMPLIFY, http://research.compaq.com/SRC/esc/Simplify.html (1999).

[11] S. Owre, N. Shankar, J. Rushby, D. Stringer-Calvert, PVS System Guide 2.4, CSL, SRI International (2001).

[12] Ora Canada, Eves, http://www.ora.on.ca/eves.html.

[13] Object Technology International, Inc., Eclipse Platform Technical Overview, http://www.eclipse.org/whitepapers/eclipse-overview.pdf, 2001.