CONTRACTUAL CONSISTENCY BETWEEN BON STATIC AND DYNAMIC DIAGRAMS

By

Ali Taleghani

A Thesis Submitted to the Faculty of

Pure and Applied Science

In Partial Fulfilment of the Requirements For the Degree of

MASTER OF SCIENCE

AT

YORK UNIVERSITY

TORONTO, ONTARIO

AUGUST 2004

© Copyright by Ali Taleghani, 2004

Copyright Page (ii - not typed)

Certificate Page (iii - not typed)

Abstract

Model-Driven Development (MDD) is a new software development technique in which the primary software artifact is a model which is a collection of views. A model is more abstract than program code in traditional languages as it is closer to the problem domain and not tied to the underlying implementation platform. The main benefit of MDD is that models are executable and automatically translated to code. Current MDD tools (e.g. Rational Rose RealTime and BridgePoint) use the UML notions of structured classes and state machines for constructing models. However, many software developers prefer to also provide a dynamic object communication view of the product (e.g. using sequence diagrams). However, the problem then arises that the dynamic view might be inconsistent with the static class view. Current MDD tools do not provide support for executable dynamic diagrams or the consistency problems that may arise.

In this thesis, we develop a theoretical approach for checking the consistency between a BON static and dynamic diagram. Specifically, we concentrate on *contractual consistency* of a system, whose behaviour is described through the use of contracts in the static diagram. Contractual consistency is checked via symbolic execution of the dynamic diagram using a theorem prover to deal with the contracts that occur in the static diagram. We develop a formal theory and definition of contractual consistency and provide a prototype tool BDT (BON Development Tool) for doing the checks automatically. To our knowledge, this is the first tool to actually check multi-view contractual consistency for a partial model involving contracts only, without the need to implement features.

BDT is a prototype and would need substantial work to make it industrial strength. Nevertheless, our multi-view approach indicates that current single-view MDD tools can be substantially improved by offering developers the use of dynamic diagrams and the benefits of contracting.

Acknowledgments

I would like to take this opportunity to thank all those who made this thesis possible. Firstly, I would like to thank my supervisor, Dr. Jonathan Ostroff, for his continued support throughout the work on this thesis. His dedication, helpfulness and expertise made this thesis possible and taught me many valuable lessons for a future in research.

Special thanks go to Dr. Vassilios Tzerpos for serving as member of my thesis committee and for serving on my Oral Examining Committee.

I would also like to thank my fellow students Yan Gao, David Makalsky and Oleksandr Fuks for supporting me when necessary and providing valuable feedback.

In addition, I would like to acknowledge IBM's contribution to this work. This research was mainly supported by an Eclipse Technology Grant. Further, special thanks to Marin Litoiu and Arthur Ryman for providing valuable support throughout my research.

Finally, I would like to thank many family for their support and understanding.

Contents

Chapter 1

Introduction

Today, computers and computer programs infiltrate our lives as never before. We use software when checking E-mail, driving a car or simply warming up food in a microwave. Over a very short period of time, software has become a very important social and economic factor in our society. This increase in both the amount and scope of usage has created increased pressure on software vendors to produce high-quality reliable software. Creating high quality software, however, is a very demanding and often costly task. Many different development techniques have been suggested to automate many tasks and therefore lift some of the burden from the developer.

One technique that has evolved recently is that of Model-Driven Development (MDD) [MCF03]. In MDD, models are the primary artifact of development and are kept and maintained throughout the development process.

It is argued that using models for development raises the abstraction level considerably and as a result, simplifies the task of software development for programmers [AK03]. In MDD, models are executable and the developer works at the model level rather than at the code level [Sel03].

Modelling in Model-Driven Development is achieved through the use of a modelling language such as the Unified Modeling Language (UML) [BRJ99, OMG03]. Currently, UML is the most widely used modelling language in the software engineering industry. In fact, it has out-grown this field and is often used in other engineering disciplines as well. It can be argued that this increase in both size and scope has made the language difficult to understand and manage [Dou03]. A discussion of UML is, however, beyond the scope of this work.

UML 2.0 offers nine different views that can be used by the developer to demonstrate various aspects about a software system. In particular, state machines, class diagrams and collaboration diagrams can be used to model an object-oriented system in UML. Using modelling and various views to model a software system - in the case of this work an object-oriented system - has several advantages [Sel03]:

- *Abstraction:* A model removes detail that is irrelevant for a given viewpoint and thus increases understandability. The developer has the ability to show and hide details throughout the development.

- *Understandability:* Abstracting information away is not enough. Models try to present whatever information is left in a way that appeals to our intuition. Some models use graphical notation, others use textual notation.

- *Accuracy:* A model can be an accurate representation of the modelled systems' features of interest. It is important, therefore, that the model is unambiguous.

- *Predictiveness:* Models can be used to predict the modelled systems interesting, but non-obvious properties.

- *Inexpensiveness:* Construction of models is in general cheaper than that of the modelled system and code.

To achieve these advantages, however, several conditions must be satisfied by MDD. Many development techniques such as I-CASE and AD/Cycle have been proposed in the past [Amb03], but they have failed to become industry-wide standards either because they were too complex or they were not market-ready when released. In order for MDD not to follow their dead-end path, the following requirements must be met [Sel03, SK03]:

- *Model consistency:* Working with multiple, interrelated, views in a model requires significant effort to ensure their overall consistency. If we have

an abstract view A of a system and we construct A by using different abstract views $A_1 \ldots A_n$, each corresponding to the concern it models, then it is possible to transform each A_i into the implementation C_i [KR03]. The main problem with this approach is that inconsistencies between A_i's could exist that have to be found before the code is generated. If these inconsistencies are not found, then the model, and as a result the generated code, will be inconsistent.

- *Model executability:* In order to obtain valuable insight into the system and allow the developer to work at the model level only, model execution has to be achieved. It is also crucial for incomplete models to be executable and therefore testable.

- *Model-level observability:* It is important that model-level error reporting and debugging functionality accompanies automatic code generation. Programmers that are faced with fixing code they do not understand could easily break it and further become discouraged from relying on the model in the future.

- *Efficient code generation:* The code that is automatically generated must be at least as efficient as hand-crafted code otherwise developers will not rely on models and MDD.

- *Scalability:* Model-Driven Development is aimed at large scale software

development. It is therefore important to support scalability and enable large development teams to work on a product. Compilation and build-times cannot be significantly increased if MDD is to be introduced into the development cycle.

In this thesis we will concentrate on the requirement of *model consistency* and as will be evident, we will introduce a theoretical approach that uses *symbolic model execution* to check for *contractual consistency*.

1.1 Consistency of Views in a Model

As mentioned above, the consistency between separately constructed views of a software system is one of the requirements of MDD for the construction of error free software. In this section, we will define consistency in more detail and explain the area that this work will concentrate on. Our discussion will be based on models of object-oriented systems [Mey97].

Figure 1.1 shows a possible consistency hierarchy that could be used to define model consistency more precisely. We can divide model consistency into *single-view* and *multi-view* consistency.

1.1.1 Single-View Consistency

Single-view consistency refers to the constraints that must hold within a single view. A single-view constraint for a class diagram for example, could be to

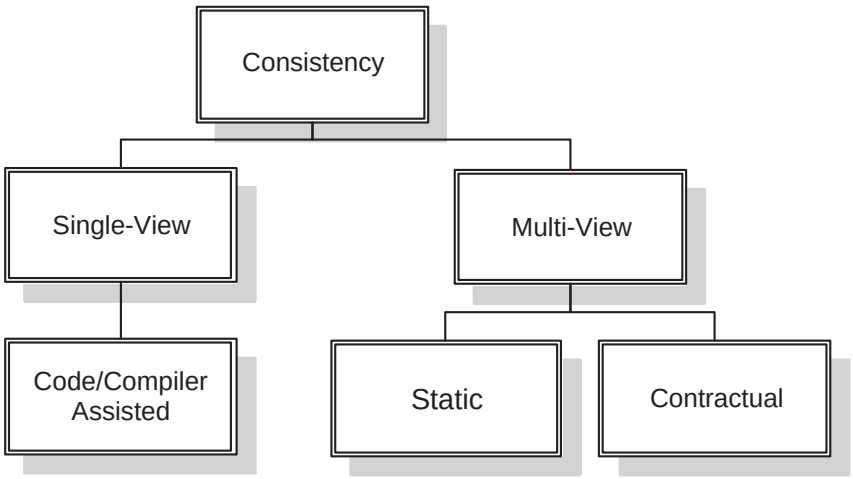

Figure 1.1: Possible division of consistency in single-view and multi-view consistency.

disallow classes with the same name. Single-view consistency is supported in most modelling tools. One form of single-view consistency is *code/compiler assisted* consistency (other forms of single-view consistencies exist as well, but are not discussed here). In this kind of consistency, a UML *structured class* [Dou03] is used to model the static and dynamic behavior of a class. The static structure is modelled in the form of a class diagram and the dynamic behavior in the form of a state machine (both are part of a structured class). In addition, the UML Action Language is used to specify detailed behaviour in the state chart.

We consider this model as a single view since the same information can be described as part of one view (e.g. see the expanded BON contract view in Chapter 2). Single-view consistency of this model is checked by compiling and

executing the code specified as transitions in the state chart. Currently, there are several tools that support this type of consistency. Two of them will be discussed later in Section 3.1.1 and Appendix B.

1.1.2 Multi-View Consistency

Multi-view consistency refers to the satisfaction of constraints between two or more different views of a modelled system. In general, views of a system have overlapping information and we have to ensure that this information is not contradictory in order to preserve consistency.

Multi-view consistency can be divided up into constraints that can be checked without a (symbolic) execution of models and those for which execution is necessary. These two cases are as follows:

- *Static Consistency:* This form of consistency can be checked statically without execution of models. Depending on the views that are being checked several static conditions can be checked. An example for this type of consistency for a class diagram and collaboration diagram would be that objects appearing in the collaboration diagram must have corresponding classes in the class diagram.

- *Contractual Consistency:* Contractual consistency is a form of consistency that requires (symbolic) execution of models and is the main focus

of this thesis. The concept is mentioned in [POB02], but not defined precisely enough for the purposes of a tool. In the next section, we explain the idea behind contractual consistency and present a simple example. Contractual consistency is defined more precisely in Chapter 3.

1.2 Contractual Consistency - An Overview

The purpose of this thesis is to develop a method for checking the consistency of (static) class diagrams and (dynamic) object collaboration diagrams. We will consider a class and collaboration diagram consistent if they are contractually and statically consistent.

The behavior of a system is modelled through the use of contracts. These contracts specify operations via pre- and postconditions and class invariants. In this approach, a class diagram with routine contracts and a collaboration diagram are used as the static and dynamic view respectively.

Informally, when checking for *contractual consistency* between a class diagram and a collaboration diagram we have to ensure that all contracts are satisfied each time a message from the collaboration diagram executes. If this condition applies to all messages then we will consider the two diagrams *contractually consistent*. We will define this concept more precisely in Chapter 4.

The following example illustrates the idea behind contractual consistency. Our example uses the modelling language BON [WN95] rather than UML,

since we believe that BON is a simpler modelling language to understand, but at the same time offers most of the features of UML that are needed for our discussion. BON will be discussed in more detail in Chapter 2.

Figure 1.2 shows the static class diagram of two classes. Class ACCOUNT is shown in its expanded form to show all its features. Class PERSON is shown in its compact form since its features are of no interest at the moment. Class PERSON has a client-supplier relationship with ACCOUNT which is indicated by the double-arrow.

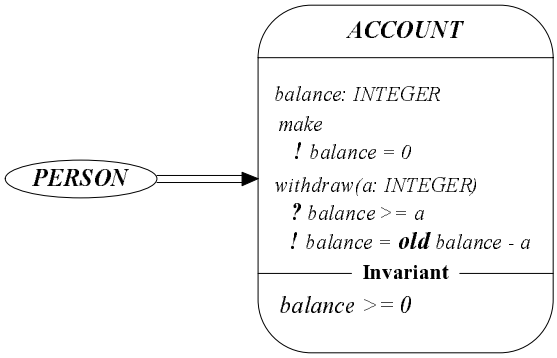

Figure 1.2: Static diagram of example demonstrating contractual consistency. Two classes are shown and their client supplier relationship. Class ACCOUNT has three features: *balance*, *make* and *withdraw*.

Class ACCOUNT has three features: *balance*, *make* and *withdraw*. Feature *balance* represents the balance of an account, *make* creates a new account and *withdraw* withdraws from the account. Routines *make* and *withdraw* have pre and postconditions. The feature *make* has only one postcondition that asserts that *balance* is set to zero after the execution of *make*. Routine *withdraw*'s

precondition asserts that *balance* is greater or equal to the amount *a* that is being withdrawn and its postcondition specifies that after the execution of *withdraw* *balance* is decremented by *a*. The expression ***old*** *balance* refers to the state of *balance* before the execution of *withdraw*.

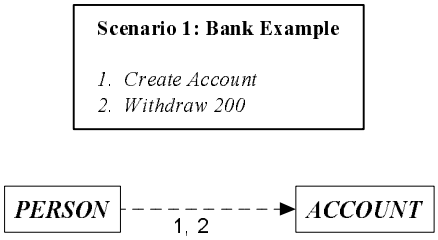

Figure 1.3: Dynamic diagram of our example. Two objects and two messages are shown. The description of the messages is given in the scenario box.

Figure 1.3 shows the dynamic view of our model. In this view, we can see two run-time objects each corresponding to the classes in Figure 1.2. The object of type PERSON sends two messages to the object of type ACCOUNT and those messages are explained in the scenario box. The first message calls *make* and the second calls *withdraw* with argument $200 on the object of type ACCOUNT. As can be seen, no implementation is provided and we are interested to see whether these two views are *contractually consistent*.

As mentioned above, to check for contractual consistency, we have to check that the contracts of all routines associated with the corresponding messages are satisfied before and after the message is executed (preconditions before and postconditions after the message execution). In the example above, the routine

make does not have a precondition, but its postcondition asserts that *balance* is set to zero after its execution. As a result, the execution of *make* can proceed without any problems and we will assume that *balance* is set to zero afterwards. The precondition of *withdraw*, however, asserts that $balance \geq 200$. But after the execution of *make* we have $balance = 0$ and as a result, the precondition of *withdraw* is not satisfied. We therefore conclude that the two views are *contractually inconsistent*. Though small, this example demonstrates the idea behind *contractual consistency*. We will define it more precisely later in this work.

This thesis is organized as follows: Chapter 2 introduces the modelling language BON and discusses the features relevant to this thesis. Chapter 3 discusses work done previously on multi-view consistency and presents one MDD tool. Chapter 4 presents the major theoretical work on *contractual consistency*. It defines the concept clearly and presents a more detailed example. Chapter 5 presents our prototype BON Development Tool (BDT) that implements the ideas presented in this thesis. An early description of this tool was introduced in [TO03]. This shows the possibility of this type of consistency checking in industrial strength tools. Chapter 6 concludes this work by discussing the significance of our work and the contribution to the software engineering field.

Chapter 2

Business Object Notation (BON)

In order to present our method for checking contractual consistency, we had to choose a modelling language. We had the choice between UML and Business Object Notation (BON) [WN95]. Both languages support static class diagrams and dynamic collaboration diagrams - the two views we are interested in. In addition, both allow for the integration of contracts within routines and classes.

UML is the most widely used software modelling language. It is, in fact, the *de facto* modelling standard. It supports nine different views of a software system including structural, behavioural and deployment views [OMG03]. The Object Constraint Language (OCL) [Gro99] could then be used to specify the class contracts. It seems that UML would have offered all features necessary

for explaining our approach, but we chose BON over UML because of the following two reasons:

- There is no standard uncluttered way to represent OCL contracts in UML class diagrams. They might reside in notes or in other text descriptions. Thus one cannot understand the contract view in one glance. By contrast, BON has a simple, uncluttered and standard way for representing contracts in the class diagrams themselves. [PO99] offers an extensive comparison between BON and UML. Figure 2.1 and 2.2 were taken from [PO99] and clearly show the difference between representing contracts in UML and BON.

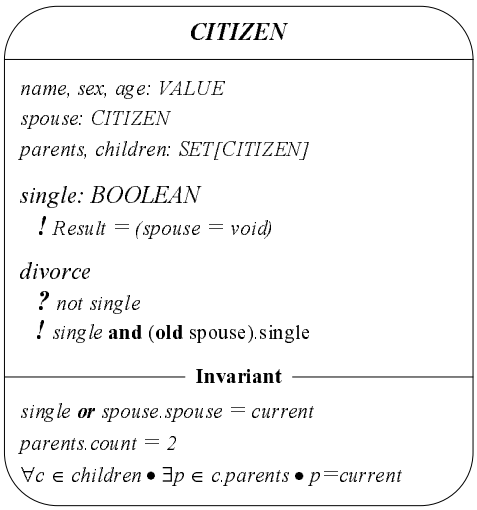

Figure 2.1: A static diagram in BON. Class CITIZEN has six attributes and two routines. The routine contracts and class invariant are elegantly incorporated into the modelling language.

- None of the current UML tools supports automatic translation of OCL

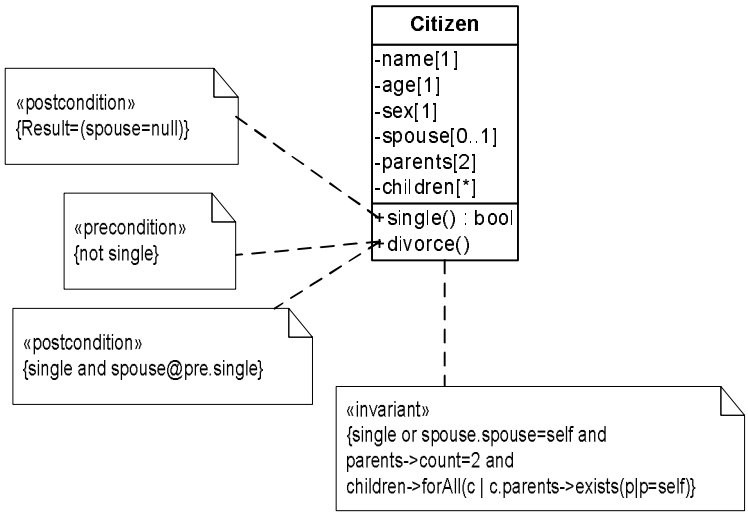

Figure 2.2: A class diagram in UML representing the same class as in Figure2.1. Contracts are shown in OCL, but cause the diagram to become cluttered.

to code. For example, contracts in Java could be described with JML [GTLJ00]. However, there is an impedance mismatch between OCL and these other specification languages (e.g., see the mapping from OCL to JML provided in [Ali04]). Translating BON class diagrams with their contracts to Eiffel [Mey92] code is seamless.

It is important to note that the method presented in this work can easily be applied to models modelled in UML and OCL without major modification. BON was chosen as the modelling language in order to simplify the addition of contracts into static diagrams.

BON is a set of concepts for modelling object oriented software. It supports two notations - a graphical and textual notation - and has a set of rules and

guidelines for producing these models. The concepts and notations in BON are designed to encourage a reusability approach by emphasizing *seamlessness*, *reversibility* and *software contracting*. Further, the software system is described through modelling its static and dynamic behavior. Static descriptions document the structure of a system and show what the components are and how they are related to each other. Dynamic descriptions, by contrast, document how the system will behave over time. The ideas of *software contracting*, *static diagrams* and *dynamic diagrams* are the ones most relevant to this work and will be discussed below. The interested reader can refer to [WN95] for a more detailed discussion of all aspects of BON.

2.1 Software Contracting

Perhaps the most important feature of BON is its software contracting ability and support of Design by Contract (DbC) [Mey97]. Software that has been designed for reuse needs to be of extra high quality because of the accumulated damage it can cause when used in thousands of components. If one part fails, it is possible for the whole system to fail. It is therefore important to find new ways to guarantee software correctness. The theory of software contracting tries to bring elements from the research fields of abstract data types and formal specification into standard use in software engineering. The idea is that assertions can define the *semantics* of each class. These assertions are in

the form of *pre* and *postconditions* for routines and *class invariants* for overall class consistency. These assertions form the basis for a contract between a class, the supplier, and all classes using its features, the clients.

Routines in BON can include *preconditions* and *postconditions*. The precondition states a predicate that must be true at the time the feature is called by the client. It is the client's responsibility to ensure that the precondition is fulfilled before the call is made. The postcondition states a predicate that must be true when the feature has been executed and the supplier object returns control to the client. Given that the precondition is true on feature entry, it is the supplier's responsibility to ensure that the postcondition is true before returning control to the client. Further, postconditions are double state formulae, i.e., they contain information regarding the state of attributes before and after the execution of a routine. The keyword **old** is used to refer to state of attributes before the execution of a routine.

Class invariants represent conditions that must hold for the entire class before and after each routine execution. In effect, they are an integral part of every pre and postcondition. Class invariants are single state formulae and do not have to be satisfied before the constructor of a class is called. A more detailed explanation on invariants is given in Section 4.6.

Based on these contracts, a consistent error handling mechanism is possible. Assertions can be monitored at run-time and contract violations can

be made to cause system exceptions. It is argued that software contracting represents a significant step towards the production of correct software and should be included in any object-oriented analysis and design method aimed at building reliable, high-quality software systems [WN95]. Our approach for defining *contractual consistency* depends on contracts in the static diagram as an alternative to state machines to specify behavior.

2.2 Static Diagrams

Static diagrams in BON show the classes that make up the software system, their interfaces and how they are related to each other. A static diagram concentrates on the *what* part and downplays the *how*. We are interested in the components of the system and the operations they can perform, but not how they interact. BON static diagrams offer a description of the system, which consists of fully typed class interfaces and formal specification of software contracts.

2.2.1 Class

The fundamental construct in a BON static diagram is a class. Classes in BON can be represented in two forms: A compact and an expanded representation. Figure 2.3 shows examples of these two forms. A compact class is represented through an ellipse, which contains the class name in the center. In addition

to the class name, the state of the class can be shown, e.g. deferred, effective etc. The compact representation is aimed at situations when the interface is less important and interactions are the main concern.

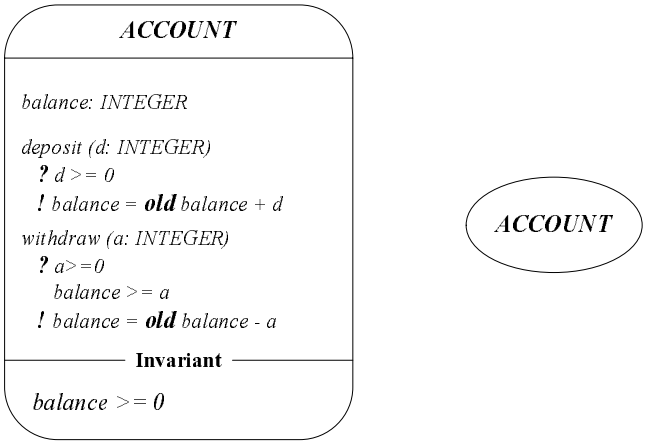

Figure 2.3: Examples of compact and expanded classes in a BON static diagram.

A BON expanded class displays all the information a class contains. As shown in Figure 2.3, it contains the information from a compact class, but can also include the following:

- An *indexing clause*, which acts as documentation for the class;
- An *inherit clause* listing all super classes;
- Public, private and selectively exported *features* including their contracts;
- An *invariant clause*, which lists all class invariants.

2.2.2 Features

Class features are fully typed in BON and the signature is specified for each feature. Class features consist of *commands*, *functions* and *attributes*. A *query* can be a function or an attribute. Commands are routines that can change the state of an object, but do not return a value. Queries on the other hand do not change the state, but return a value (public attributes are also considered queries, but do not have pre and post conditions). On top of specifying the signature of the feature, pre and postconditions can be specified for commands and queries. Figure 2.3 shows an example of this. In graphical BON, "**?**" is the graphical notation for preconditions and "**!**" the notation for postconditions. The meanings of pre and postconditions were explained in the previous section.

2.2.3 Relationships

There are three static relations in BON. Two are client-supplier relationships and one is an inheritance relationship. Figure 2.4 shows the three different relationships and their graphical representation. A class may inherit from one class (single inheritance), from several classes (multiple inheritance), or several times from the same class (repeated 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. Inheritance may translate differently depending on the object-oriented language used, so the definition is

kept very general in BON. An inheritance relation is represented by a single arrow pointing from the child to its parent, called an inheritance link.

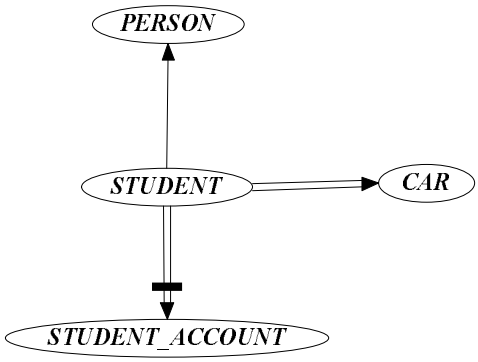

Figure 2.4: Illustration of three static relations in BON. STUDENT has inheritance relationship with PERSON, association with CAR and aggregation with STUDENT_ACCOUNT

An association between a client class and a supplier class means that (at system execution time) some instances of the client class may be attached to one or more instances of the supplier class. A particular instance of the supplier class may take part in many such attachments, thus permitting supplier objects to be shared by different clients.

An aggregation relation between a client class and a supplier class means that each client instance may be attached to one or more supplier instances which represent "integral parts" of the client instance. The parts may in turn have client relations to other classes, which may be aggregations (subparts) or associations. Aggregation is an important semantic concept, useful when thinking about the properties of objects in a system - which is why it has a

special notation in BON - but not so easy to define exactly. Aggregations are often implemented as simple associations.

More detailed information regarding the static model including clusters, BON assertion language and relationships can be obtained from [WN95].

2.3 Dynamic Diagrams

Feature calls, or message passing between objects, are what constitute a system execution and consequently, this is what a BON dynamic diagram expresses. [WN95] proposes a new notation for dynamic behavior of a system, called dynamic diagrams. Dynamic diagrams in BON are similar to collaboration diagrams in UML. BON dynamic diagrams strive for a higher level of abstraction that can be naturally expressed with fewer parameters. In this section, we will give an overview of BON dynamic diagrams and refer the user to [WN95] for a more detailed discussion of these.

2.3.1 Objects

A dynamic diagram consists of a set of communicating objects passing messages to each other. An object is represented by its type - that is its class name, and an optional object qualifier. Objects are represented graphically by rectangles to make them stand out from the class ellipses. Figure 2.5 shows an example of a BON object in a dynamic diagram.

Figure 2.5: A single object (left) and a set of objects (right) are shown. PERSON is a single object with the qualifier *Peter*. ACCOUNT is a set of objects. The number of objects within a set can vary.

The name of the corresponding class is centered inside the rectangle in upper case. A single rectangle in a diagram always refers to an individual object, so two single rectangles with the same name will refer to two individual objects of the same type. A qualifying identifier may be written below the class name, enclosed in parentheses, to distinguish several objects of the same type in a dynamic diagram. An object rectangle may be double, in which case it refers to a set of objects of the corresponding type. Passing a message to a double object rectangle means that a message is passed to all objects in the set. Figure 2.5 shows an example of the notation for a set of objects.

2.3.2 Messages

A dynamic diagram also shows the messages passed between objects. A message sent from one object to another is shown as a dashed line extending form the calling object to the receiving object. Messages are numbered which serves a double purpose: First, they represent time in the scenario and second, they correspond to entries in the scenario box where each message's role might be described using free text. Figure 2.6 shows examples of messages and a

scenario box. There is one scenario box per dynamic diagram.

In [WN95], it is understood that a message in a dynamic diagram corresponds to *some* routine in the static diagram. It is not necessary to specify the precise routine in the initial design phase. A textual description in the scenario box suffices. But, this is the critical point at which the static and dynamic views overlap (i.e. the correspondence between messages in the dynamic diagram with feature calls in the static diagram). In an actual BON consistency tool, the mapping between messages and features will need to be more precise. This means that we will need to support (a) informal diagrams where the mapping is unspecified, but also (b) the formal ability to map messages to actual features (which must be completed before the actual consistency check can be invoked). This mapping will play a major role in our definition of contractual consistency.

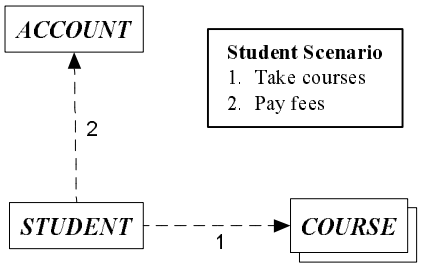

Figure 2.6: Three run-time objects and scenario box are shown. STUDENT sends two messages which are explained in the scenario box.

A message relation is always potential; that is, we cannot tell from the diagram whether the call will actually occur in a specific execution. A message

call might not succeed at the time of execution if the preconditions of an associated routine are not satisfied.

2.3.3 Sub Messages

Another important aspect of messages in a dynamic diagram are sub-messages [Gao04]. A message m_1 sent between objects o_1 and o_2 represents the routine call r_1 on o_2. Once the routine call is completed, control is returned to o_1 again. According to this definition a scenario such as that in Figure 2.7 is not permitted because of the following: After the execution of m_1 control is returned to $OBJECT\_1$ and as a result, m_2 cannot be executed anymore. In order for this diagram to be executable, m_2 must be called within the body of m_1 and control is returned to $OBJECT\_1$ only after the execution of m_2 completes. As a result, m_2 becomes a sub-message of m_1.

This scenario is not discussed in [WN95] and [Gao04] suggests using a new notation for sub-messages. Figure 2.8 demonstrates this notation for sub-messages. The notation indicates that $m_{1.1}$ is called within the body of m_1 and as a result, control is not returned to $OBJECT\_1$ until $m_{1.1}$ has completed its execution.

Figure 2.7: An illegal message sequence in BON.

Figure 2.8: Using multi-dot notation to represent sub-messages.

In summary, a diagram such as the one in Figure 2.7 is permitted in BON, but will not be permitted in diagrams that are later checked for contractual consistency. In order to specify sub-messages for such messages a notation such as the one in Figure 2.8 must be used. As will be seen later, this case does not have to be considered in our approach to *contractual consistency* since no implementation is provided. The developer might, however, wish to draw sub-messages in the dynamic diagram. As a result, it is important that a solution for sub-messages within contractual consistency is found.

Chapter 3

Literature Survey

Chapter 1 gave an overview of the problem of consistency and our distinction between single-view and multi-view consistency. In general, the problem of consistency arises when models are used to describe a software system and we have to ensure that the information in these models is consistent. For multi-view consistency, multiple views of a system can have overlapping information that can cause the two views to become inconsistent. It is important to discover these inconsistencies early and ensure that they do not propagate into the program implementation.

The problem of view consistency is one that has been known and discussed extensively. In most cases, UML is used as the modelling language for the discussion. Discussions are often divided into those trying to solve single-view

consistency problems [ARR00, SC02] and those working on multi-view consistency [Gli00, HHM01, Kri00, EN95, Egy01]. While there are several single-view consistency tools available [Rat02, Tec04], not much work has been done on multi-view consistency tools. Apart from the work in this thesis, the only other such tool that we are aware of is the recent M.Sc. thesis of Gao [Gao04]. Gao describes and implements a multi-view consistency tool that detects inconsistencies that we have categorized as static inconsistencies (see Figure 1.1). Gao's tool also treats contractual consistency by allowing developers to generate a test driver. The test driver can be compiled by any Eiffel compiler and executed. If no contract violations occur then there is at least one execution of the dynamic diagram that satisfies the static diagram. This approach requires implementation detail for the contracts of each feature. The BDT tool developed in this thesis complements Gao's tool as BDT focuses on contractual consistency. Both tools use the definitions of multi-view consistency developed in [POB02] as the starting point.

In this chapter, we will introduce the Model-Driven Architecture (MDA) [MSUW03] framework and one MDA modelling tool that promises model execution and model consistency, but in fact only checks single-view consistency. A second tool will be discussed in Appendix B. Further, we will discuss the most relevant works in the area of multi-view consistency. Finally, we will explain and discuss [POB02], which is the work most closely related to this

thesis.

3.1 Model-Driven Architecture and MDA Tools

Model-Driven Development presents a set of ideas that could deliver improvements in productivity and decrease the cost of software development. Model-Driven Architecture (MDA) [MSUW03], an initiative by the Object Management Group [OMG04], tries to offer a conceptual framework for defining a set of standards in support of the concepts of MDD. MDA's main initiative is to provide a framework for software development that uses models to describe the system to be built [MSUW03]. The goal is to have MDA tools that let the developer specify models at various abstraction levels and keep them synchronized [Uhl03]. These models also tend to grow increasingly independent of the target platform, making future adjustments to new target platforms easier [MSUW03].

Key to achieving platform independence within MDA is the notion of two models: *platform-independent models* (PIM) and *platform-specific models* (PSM) [McN03]. As the name suggests, PIMs are platform independent and as a result, do not require any change as the underlying technology and deployment platform change. It is argued that the benefit of PIMs is that being technology independent, they can best preserve the intellectual and financial investment. PSMs can be generated through refinement of PIMs, and

one given PIM can be mapped to several PSMs. The final stage of the development is to generate the code from the PSM so the system can be deployed. If at some point in time the application has to be deployed on several platforms or migrated from one platform to another as a result of technology changes, then the required code can be generated from the PIM. It is argued that this method is faster and cheaper than migrating the deployed code [McN03].

In terms of modelling language, MDA has been supporting UML. Supporters of this standard argue that UML is a solid foundation on which MDA is built and that UML has been instrumental in the transition from code-oriented to model-oriented software production techniques [BB01]. Within UML there are two ways to specify the behavior of the system. This issue has become a dividing point in the MDA community. Two approaches have been suggested to specify system behaviour [McN03]:

- The dynamic behavior of a system can be specified as a set of constraints that must be true about the system before and after execution of messages (operations). These constraints can be in the form of contracts - pre and postconditions - for operations and invariants for classes. In UML, these constraints can be specified using the Object Constraint Language (OCL) [Gro99]. It is often argued that the expressive power of this form of dynamic specification is limited and not all aspects of a system can be defined this way. Secondly, the automatic generation of complete code

from pre and postcondition is not possible. User intervention is necessary and the programmer has to supply additional implementation for operation bodies.

- An alternate approach is to use UML state machine diagrams with Action Language (AL) of UML [OMG03]. This language extends UML with compatible mechanisms for specifying action semantics in software-platform independent manner. AL specifies how the model will react in response to a particular stimulus, what changes are made within the model and how they are applied. One limitation of Action Language is that execution semantics of UML concepts are not precisely defined. As a result, actions and their executions are defined in the documentation of UML, but not within the meta-model. There is therefore the risk of two software engines interpreting the same model in two different manners.

In the following subsection, we will describe one MDA tool that uses state machines to specify behavior of the system. As mentioned, this tool allows construction of collaboration diagrams, but they are not incorporated into model consistency and model execution. A second tool is described in Appendix B. As will be seen, this method for specifying behaviour is in contrast to our approach which uses contracts only. In this thesis, we argue that contracts are valuable enough to allow the user symbolic model execution, model consistency and partial code generation as seen in Eiffel [Mey92].

3.1.1 BridgePoint

BridgePoint [Tec04] is an MDA tool developed by Project Technologies. BridgePoint's development environment is based on executable and translatable UML (xtUML) [MB02]. It is argued that xtUML Platform Independent Models (PIMs) [MB02] completely and concisely describe what the system does and are fully testable and executable. BridgePoint relies on class diagrams (Data), statecharts (Control) and Action Language (Processing) to define a system and its behavior. According to BridgePoint, this kind of specification allows for early PIM testing and defect elimination and as a result, system quality is increased. In addition, xtUML PIMs are automatically translated, by customizable model compilers compromised of translation rules and patterns, to generate complete target code. Changes to the application defined by the PIM or to the software architecture defined in the model compiler are automatically reflected in the system's generated code.

The model verifier in BridgePoint is expected to provide early PIM execution, debug and consistency capabilities before code translation. It is claimed that entire domains or individual threads of control can quickly be tested and debugged. Single-view consistency is achieved through compilation and execution of the Action Language. Any problems in the code can be seen as inconsistencies that are brought to the attention of the user through a model debugger.

As mentioned before, BridgePoint allows the user to construct collaboration and sequence diagrams, but does not check their consistency and therefore, they are not incorporated in model execution and code generation. As a result, developers cannot take full advantage of such diagrams.

3.2 Multi-View Consistency

As mentioned, multi-view consistency has been discussed before, but no tools have been developed to check contractual consistency directly. [Gli00] introduces a new approach for checking the consistency between UML class diagrams and scenario models. In this approach, there are three main possibilities for inconsistencies to arise. These inconsistencies are based on the following overlap between the two views:

- Stimuli in a scenario require a corresponding operation or state transition in the class model.
- A response in a scenario that needs more data than the stimuli of this scenario provide requires stored data.
- Every operation in the class model which is not referenced by another operation in the class model is typically used by a scenario in order to process a stimulus or to produce a response.

[Gli00] tries to resolve inconsistencies by reducing the amount of overlap between the two views. We believe, however, that a reduction in amount of overlap is not the correct approach. A certain amount of overlap is needed to assure that both views describe the same system. Further, the inconsistencies that this approach catches are static inconsistencies that are further described and implemented in [Gao04]. The approach described in [Gli00] does not involve the behavior of diagrams or the execution of a model.

[Kri00] discusses an approach that is most related to our work and that in [POB02]. This work tries to translate each view in UML into a constraint described in OCL [Gro99] and using the theorem prover PVS [OSR01] to check certain constraints. In particular, consistency between a class diagram and collaboration diagram is achieved by trying to find a trace that satisfies all constraints in the static diagram. Each state in the trace has to satisfy all relevant state predicates in the system. This approach is very similar to that in [POB02] and the authors argue that their approach is superior since it deals with UML and all the views this modelling language supports, while that in [POB02] applies to BON, which only supports two views of a system. [Kri00], however, does not present any tool that implements this approach and shows that this method is implementable.

Another approach that is suggested is to transform two views into one common view and then compare them [Egy01]. In this approach, when trying

to find inconsistencies between a UML state machine and sequence diagram, a sequence diagram is converted to a state machine and then the two models are compared. A method introduced in [WS00] could be used to convert sequence diagrams to state machines. [Egy01], however, does not explain in detail how similar models can be compared and which constraints should be used for their comparison.

As can be seen, many different approaches for multi-view consistency exist, but few tackle the difficult task of contractual consistency and symbolic model execution. In most cases, the multi-view consistency check is only a static consistency check or the approach described does not have an accompanying tool to show it is implementable. The next section discusses [POB02], which is most closely related to our work in this thesis.

3.3 Related Work on Contractual Consistency

The problem of multi-view consistency is not unique to UML or BON and any modelling language that supports multiple views has the potential to introduce inconsistencies into the system. In this thesis, we are interested in checking the consistency between dynamic and static diagrams of an object-oriented system. Specifically, the *contractual consistency* between a BON class diagram and dynamic diagram is discussed. In this section we will discuss the work of [POB02] which is the first work on contractual consistency between a dynamic

and static diagram.

[POB02] introduces a four step approach to checking consistency between a BON static and dynamic diagram. The four steps are based on conditions that can be checked statically (symbolic execution is not necessary) and those that require execution of messages. The four steps are as follows:

1. Each object appearing in the collaboration diagram must have a corresponding class in the class diagram. This condition ensures that objects for which no class is defined cannot be used in consistency checking.

2. Each message in the collaboration diagram has a routine from the static diagram associated with it. If no routine is associated with a message then it cannot be determined how a message call would change the system state.

3. Each routine associated with a message must be from the target object of that message and must be exported to the source object. This requirement ensures that source objects call routines in the appropriate classes and have permissions to access them.

4. Routines that are called must be enabled, i.e., the precondition of a routine associated with a message must be satisfied before the message can be executed. A precondition can only be satisfied if the sequence of previous calls to routines left a system state satisfying the precondition.

[POB02] argues that if all four requirements from above are met then a BON static and dynamic diagram can be considered consistent. We will refer to the first three requirements as static multi-view consistency. They have been further discussed and implemented in [Gao04]. These requirements can be checked statically without the symbolic execution of any messages. [Gao04] also checks indirectly for contractual consistency. The user is given an infrastructure to develop tests that check the implementation against the contracts. One major drawback of this approach is that code must be provided to check for contractual consistency.

Requirement four is the more challenging step and requires (symbolic) execution of the model. It is this step that we refer to as checking for *contractual consistency* and will define it more precisely in the next chapter.

[POB02] offers an initial attempt at defining contractual consistency. The definition is as follows: Assume that dd represents the dynamic diagram, $dd.calls$ all the message calls in dd and $dd.calls.item(i)$ message m_i in dd. Further, $dd.calls.item(i).pre$ is the precondition of the routine associated with m_i. Then, in order to start checking requirement 4, a user specified initial state $init$ (specified as a predicate) must imply the precondition of the first element in the sequence of calls in the collaboration diagram:

$$init \Rightarrow dd.calls.item(1).pre$$

For any subsequent message call we have to ensure that the system state left

behind satisfies the precondition of the routine associated with a message. Formally, this was expressed in [POB02] as:

$$\forall i : 2 \leq i \leq dd.calls.length \bullet dd.spec(i-1) \Rightarrow dd.calls.item(i).pre \quad (3.1)$$

where $dd.spec(i-1) \stackrel{def}{\Rightarrow} (dd.calls.item(1).spec; ...; dd.calls.item(i-1).spec)$ and *spec* is defined as **old** *precondition* $\Rightarrow$ *postcondition*. (3.1) states that the sequential composition of all specifications (*spec*) of messages executed must imply the precondition of the current message. The sequential composition is defined as:

$$P; Q = \exists s' \bullet P[s := s'] \wedge Q[\mathbf{old}\, s := s']$$

While this definition offers an early attempt to define *contractual consistency*, it stops short in defining it fully for the following reasons:

- The notion of class invariants are not explicitly included into this theory. Class invariants are an integral part of every pre and postcondition and must be checked to ensure that they are not violated.

- (3.1) does not incorporate *init* into the formula. Once m_1 is checked, *init* must be incorporated into the system state and propagated to the next message.

- A second check is needed in addition to (3.1). For example, suppose that

$$dd.spec(i-1) = (x > 1) \ \wedge \ (x < 1)$$

and that we have some arbitrarily chosen precondition $r.pre$. Then, (3.1) results in:

$$((x > 1) \wedge (x < 1)) \Rightarrow r.pre$$

Since the antecedent is equivalent to false, (3.1) is trivially *true* and message m_i is enabled. However, there is no state that satisfies the antecedent $dd.spec(i-1)$. In BON terms, there would be a postcondition contract violation after execution of one of the prior messages. We therefore need an additional check to make sure that $dd.spec(i-1)$ is not a contradiction.

In this thesis, we will clearly define the concepts mentioned above and further introduce a prototype tool that was developed to demonstrate that our approach to *contractual consistency* is implementable.

Chapter 4

Contractual Consistency

The purpose of this thesis is to precisely define *contractual consistency* between a BON static diagram and a dynamic diagram and introduce a theoretical approach for checking this kind of consistency. In this chapter we will present the theoretical work necessary to define this concept and present the BON Development Tool (BDT) in the next chapter. We will show that BDT incorporates most of the concepts discussed in this chapter. It will also be evident that checking for *contractual consistency* is closely related to *symbolic model execution.*

Informally, when checking for *contractual consistency* between a static and dynamic diagram in BON, we check whether the precondition of a routine associated with a message in the dynamic diagram is satisfied before the message is executed. In addition, the postcondition of the routine should not evaluate

to false. If these conditions apply to all messages in the dynamic diagram, then we consider the static and dynamic diagram *contractually consistent*.

The procedure for checking for contractual consistency will be reduced to checking the validity of a predicate. If we had access to an ideal oracle that could return *valid* precisely when the predicate is a theorem, and *invalid* otherwise, our procedure would be sound and complete. However, in the real world, we must pass the predicate to a theorem prover. The theorem prover is sound but not complete. Hence, our tool will be incomplete. We will discuss the sources for unsoundness and incompleteness in Chapter 5

Despite the incompleteness and unsoundness of the tool, it is still useful. First, many theorems can be proved automatically in a short period of time, thus providing immediate feedback that consistency holds. When an invalid answer is returned, enough feedback may at the same time be supplied to establish that the predicate is not a theorem.

An important aspect of Model Driven Development and the use of models, is that models should be executable or testable even if they are not complete. Our approach (and the BDT tool) will allow for symbolic execution of partial models. The ability to deal with partial models is significant because it allows for early analysis of the model.

4.1 The BON Model

We will check *contractual consistency* between a BON static diagram and dynamic diagram. Since the focus of this thesis is on contractual conistency, we assume that static consistency has already been checked. This can be done, for example, by using a tool such as that in [Gao04]. Thus, the static diagram must satisfy the following conditions:

- Classes are defined by their data (attributes) and operations (routines);
- Routines are specified by their specifications, i.e., preconditions and post-conditions;
- Preconditions are single-state formulas and postconditions double-state formulas. Double-state formulas contain information regarding the state before and after the execution of a routine. The keyword **old** is used to refer to states of attributes before the execution of a routine;
- Classes contain no invariants or implementation code. Invariants are discussed as a special case.

The following conditions must be satisfied by the dynamic diagram:

- An object in the dynamic diagram has a corresponding class in the static diagram.

- Each message in the dynamic diagram is associated with a routine in the static diagram.
- The client object must have access to the routine it is calling in the supplier object.

A BON model of an object-oriented system that consists of a static diagram and dynamic diagram and satisfies the conditions listed above will be the input to our system. We will refer to this model as $model(SD, DD)$ where SD and DD represent the static and dynamic diagram respectively.

4.2 Example

In this section, we will introduce an example that acts as sample input to our system and that we are going to use throughout this chapter to explain our approach to contractual consistency. In this example we will model a bank that provides customers with checking and saving accounts. Each customer has one of each and can make deposits and withdrawals. In addition, she can transfer money from her checking account to her savings account. In the following two subsections, we will introduce the static and dynamic structure of this example.

4.2.1 Static Structure

Figure 4.1 shows the BON static diagram of the bank example including the relationships between classes. As can be seen, there are three classes in this system: PERSON, CHECKING and SAVING. The static structure shows that class PERSON has association relationships with both classes, CHECKING and SAVING. Class CHECKING has an association relationship with class SAVING. Using plain English, a person has both, a checking account and a savings account and the checking account is associated with the savings account.

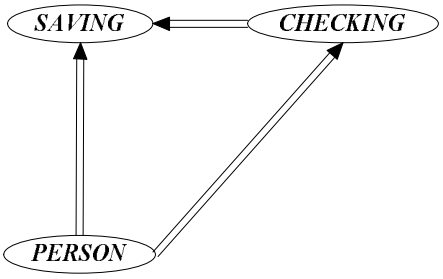

Figure 4.1: Class structure of SAVING, CHECKING and PERSON.

Figure 4.2, 4.3 and 4.4 show the classes PERSON, CHECKING and SAVING in more detail through their expanded forms. The class PERSON has two attributes *checking* and *saving* of type CHECKING and SAVING respectively. Class SAVING has an attribute *balance* of type INTEGER. Further, this class has one routine *withdraw(w: INTEGER)*. The precondition of *withdraw* states that when calling this routine, *balance* must be greater or equal

to w. The postcondition ensures that *balance* is decreased by w after completing execution. In addition, the modifies clause (Δ) is specified for this class. The modifies clause identifies the attributes that this routine can modify. The meaning of the modifies clause and its importance in this discussion is explained in more detail in Section 4.3.2.

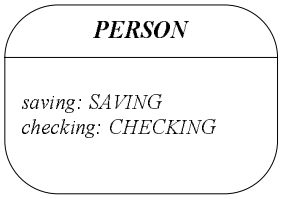

Figure 4.2: Expanded class view for PERSON. Two attributes can be seen.

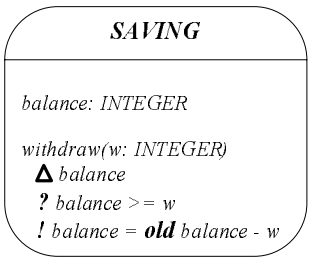

Figure 4.3: Expanded class view for SAVING. One attribute and one command can be seen.

Figure 4.4 shows the detailed view of class CHECKING. This class has three attributes: *balance* of type INTEGER, *saving* of type SAVING and *active* of type BOOLEAN. It has four commands that change the state of the object. These commands are *activate_account*, *deposit(y: INTEGER)*, *withdraw(i: INTEGER)* and *transfer(x: INTEGER)*.

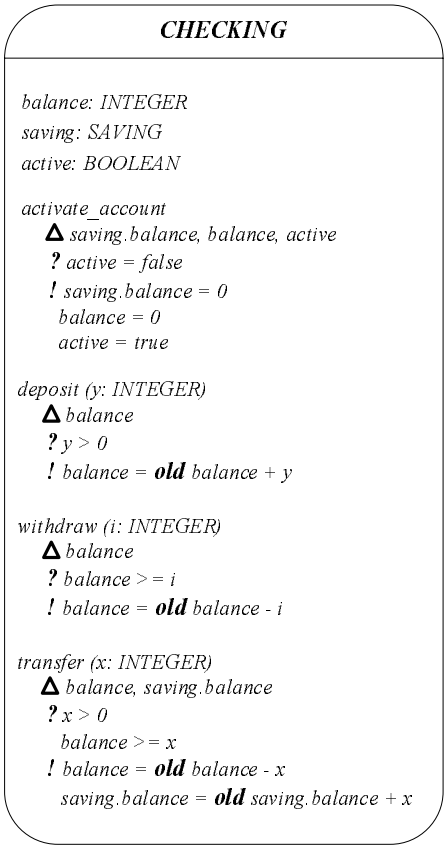

Figure 4.4: Expanded class view for CHECKING. Three attributes and four commands can be seen.

The routines and their contracts are as follows:

- *activate_account:* Activates the current instance of CHECKING. Based on the contracts, *active* must be $false$ before *activatate_account* is called and after the call, it is set to *true* and *balance* in *saving* and in the current object are set to 0.

- *deposit(y:INTEGER):* Deposits can be made to *checking* as long as the amount deposited y is greater than 0 and after deposit *balance* is increased by y.

- *withdraw(i:INTEGER):* A withdrawal succeeds if *balance* is greater than the amount i being withdrawn. After a withdrawal *balance* is decreased by i.

- *transfer(x:INTEGER):* An amount x is transferred from the current instance of CHECKING to *saving*. The precondition of this routine ensures that x is greater than 0 and that it is not larger than *balance*. The postcondition states that *balance* is decreased by x and that *balance* in *saving* (*saving.balance*) is increased by x.

This is all the information that is contained within the static diagram. No implementation is provided.

A postcondition of a routine may specify direct changes as well as indirect changes to the attributes of the model. Consider the routine *activate_account*

in CHECKING. An example of a direct change for this routine is the postcondition $balance = 0$ as *balance* is an attribute of CHECKING. An example of an indirect change is the routine's postcondition $saving.balance = 0$.

Of course, when it comes to implementation, the *activate_account* routine may not change attributes of other classes such as *saving.balance*, because, in BON, a routine has read access but not write access to attributes of other classes. This is an important part of reliable information hiding. However, at the specification level, such indirect changes may be asserted, with the understanding that at implementation time, a suitable routine of the other class will be called to do the change. But, at the specification level, this additional routine need not be declared. This is a powerful idea, because it means that we can deal with partial models.

However, at the specification level, we need a precise account of which variables change and which stay the same as will be seen later in this chapter in equation (4.1). We could try to infer the changes by analyzing the postcondition, but this is a risky business. Instead, we require that the designer specifies this clearly and precisely up front in the modifies clause. The modifies clause will thus list both direct and indirect changes. For example, the modifies clause of *activate_account* is the list $saving.balance, balance, active$. In a commercial application, however, the developer should only have to specify the direct changes and the indirect changes should be inferred automatically

by the tool.

4.2.2 Dynamic Structure

Figure 4.5 shows the dynamic diagram of our bank example. There are three runtime objects in the system: *checking* of type CHECKING, *saving* of type SAVING and $p1$ of type PERSON. In this example, four messages are sent between these objects as described in the scenario box in Figure 4.5.

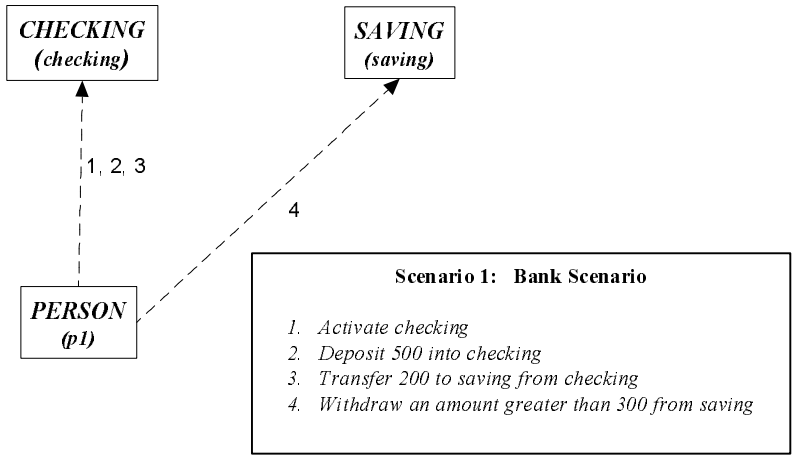

Figure 4.5: Dynamic diagram for Bank scenario. There are three objects and four messages. The scenario box describes the messages in more detail.

Table 4.1 lists the mapping between the messages in the dynamic diagram and routines in the static diagram:

Given this $model(SD, DD)$, we would like to know whether it is *contractually consistent*. In order to simplify the task of writing object and feature names, we will use the abbreviations in Table 4.3 and 4.2 for the remainder

Message	Routine	$args_i$
m_1	*checking.activate_account*	$args_1 = true$
m_2	*checking.deposit(y)*	$args_2 = (y = 500)$
m_3	*checking.transfer(x)*	$args_3 = (x = 200)$
m_4	*saving.withdraw(i)*	$args_4 = (i > 300)$

Table 4.1: Table showing mapping between messages in DD and associated routines in SD. arg_i is the constraint on the arguments of routine r_i.

Feature Name	Abbreviation
balance	*b*
saving	*s*
active	*a*
checking	*c*

Table 4.2: Abbreviations used for feature names.

of this chapter. It is important to note that BDT works with full feature and object names and communicates with the user through those names.

4.3 System State Constraint and Prover

In this section we will define the notion of a *System State Constraint (SSC)* and introduce the function *Prover* that we will use throughout our discussion.

Object Name	Abbreviation
saving	*sav*
checking	*ch*
*p*1	*p*1

Table 4.3: Abbreviations used for object names.

As will be seen, this function will later translate into the theorem prover we use in BDT.

4.3.1 System State Constraint

In the discussion of consistency, the notion of a *system state* is of importance. The system state is defined by all the entities and their values in all run-time objects at a certain moment during execution. For example, suppose the state space corresponding to our objects are the attributes a_1, a_2, a_3 where a_1, a_2 are integers and a_3 is a string, then the system state SS at an instance of time might look as follows:

$$SS \stackrel{def}{=} a_1 = 2 \wedge a_2 = 3 \wedge a_3 = Void$$

However, we are not dealing with implementations in which the state is fully defined. All we have are contracts (single state, and double-state predicates in the variables). Thus, instead of SS, what we might have at an instance of time is the System State Constraint SSC:

$$SSC \stackrel{def}{=} a_2 > a1 \wedge a_3 = Void$$

where $SS \Rightarrow SSC$.

Thus, a system state constraint SSC for a $model(SD, DD)$ is just a predicate whose free variables are the attributes occurring in the contracts of routines in the classes in SD as well as the formal arguments of these routines.

This predicate is a constraint on the state at a certain instance of time. An unqualified attribute in a contract is treated as *Current.a* where *Current* is the object in the DD sending the message.

The variables in the banking system include: $sav.b$, $ch.b$, $ch.s$, $ch.a$, $p1.s$ and $p1.c$, although not all of them are used in the contracts. There are syntactic variable names equivalent to the ones already mentioned. In the bank example, the multi-dot expressions $p1.sv.b$ and $p1.ch.sv.b$ all refer to $sv.b$ and should be in the state space with a marker to say that they are all equivalent. The Eiffel [Mey92] compiler is able to make checks for this, or alternatively we could use a version of the specified depth algorithm in [Gao04]. For simplicity, we do not consider such multi-dot syntactic variable names in this discussion, but the discussion could be generalized to deal with the multi-dot case as well.

Semantically, the execution of a message m_i results in the execution of an associated routine r_i that takes the system from one system state constraint to the next:

$$SSC_{i-1} \xrightarrow{r_i} SSC_i$$

where $0 \leq i \leq n$ where n is the number of messages in the DD. The initial state constraint is SSC_0, which represents the system state constraint before the execution of any messages. This state is either the predicate $true$ (meaning that the attributes may have any value) or is user specified, i.e. the user has the freedom to specify specific values for attributes.

In the bank example, the user might specify the following system state constraint for the initial state:

$$SSC_0 \ = \ (sav.b > 500 \ \wedge \ ch.b = sav.b \ \wedge \ ch.a = false)$$

After the execution of a message m_1 and associated routine r_1 the new system state constraint SSC_1 has to be calculated. The next section describes this step.

4.3.2 Post SSC

After a message m_i is executed with a constraint SSC_{i-1}, a new system state constraint SSC_i must be determined. This new constraint must depend on SSC_{i-1} that existed before m_i started execution and on the contracts of r_i, which is associated with m_i. The contracts state the new constraints that apply once m_i completes execution.

Routine r_i which is associated with message m_i has a precondition $r_i.pre$, a postcondition $r_i.post$ and a modifies set $r_i.mod$. For example, consider the routine *withdraw* of class CHECKING in Fig. 4.4:

```
withdraw(i :INTEGER)
    modifies
        balance
    require
        i < balance
    ensure
        balance = old balance − i
```

for which $withdraw.mod = \{checking.balance\}$ (or in the abbreviated version $withdraw.mod = \{ch.b\}$. Thus, only $ch.b$ may change when this routine is executed, but all other attributes and variables remain the same.

Let $r_i.mod = \{v_1, \cdots, v_m\}$. Then

$$\begin{aligned} SSC_i \;=\; & \exists v'_1, \cdots v'_m \bullet \\ & (SSC_{i-1}[v_1 := v'_1, \cdots, v_m := v'_m] \\ & \wedge\; r_i.post[\mathbf{old}\, v_1 := v'_1, \cdots, \mathbf{old}\, v_m := v'_m]) \end{aligned} \tag{4.1}$$

where the notation $P[x_1 := x'_1, x_2 := x'_2]$ means the predicate similar to the predicate P, except that there is the simultaneous replacement of every free occurrence of x_1 by x'_1 and x_2 by x'_2.

In (4.1), we must propagate the values of the variables that are unchanged while appending the new constraints from the postcondition on the variables that have changed. The postcondition is a double state formula which may refer to the new value of a variable (v) in the poststate as well as to its old value in the prestate ($\mathbf{old}\, v$). But, in the prestate SSC_{i-1}, $\mathbf{old}\, v$ is referred to by v. Thus, (4.1) renames the variables that change (both those in the prestate as well as $\mathbf{old}\, v$) to the same new fresh variable v'. Thus, (4.1) is the formal description of the following:

1. Constraints in variables that are not modified are propagated from SSC_{i-1} to SSC_i unchanged;

2. The postcondition of r_i is used to append the constraints on the variables

that have changed;

3. The renaming via simultaneous replacement avoids conflict between different names in the prestate.

The existential operator projects out the old state and we are left with a single state formula or constraint describing the poststate.

(4.1) represents the system state constraint after the execution of m_i. Since the existential quantifier projects out all new primed variables, no information regarding past constraints is kept. In a commercial tool, however, we might be interested in keeping this crucial information. It would help us to give feedback to the user regarding the transition of attributes as messages execute. As a result, we can re-write the above formula in a stronger form:

$$\begin{aligned} SSC_i \;\; = \;\; & (SSC_{i-1}[v_1 := v_1', \cdots, v_m := v_m'] \\ & \wedge\; r_i.post[\mathbf{old}\, v_1 := v_1', \cdots, \mathbf{old}\, v_m := v_m']) \end{aligned} \tag{4.2}$$

(4.2) is stronger than (4.1), but retains information regarding past states of the variables. Our tool BDT implements (4.2) in order to provide more valuable feedback to the user.

We can now use (4.1) to determine the post state for our bank example. Assume we started with SSC_0 as described above and executed m_1. As a result, SSC_1 has to be determined.

We have the following SSC_0:

$$SSC_0 \;=\; (sav.b > 500 \;\wedge\; ch.b = sav.b \;\wedge\; ch.a = false)$$

The postcondition of r_1 (i.e. routine *activate_account*) is:

$$r_1.post = (sav.b = 0 \ \wedge \ ch.b = 0 \ \wedge \ ch.a = true)$$

The modifies clause of r_1 is as follows:

$$r_1.mod = \{sav.b, \ ch.b, \ ch.a\}$$

As a result, we know that the attributes in $r_1.mod$ are modified and have to be renamed in the construction of SSC_1. Using equation 4.1 we obtain the following:

$$\begin{aligned} SSC_1 \ = \ & \exists \, sav.b', ch.b', ch.a' \bullet \\ & SSC_0[sav.b := sav.b', \ ch.b := ch.b', \ ch.a := ch.a'] \\ & \wedge \, r_1.post[\mathbf{old}\, sav.b := sav.b', \ \mathbf{old}\, ch.b := ch.b', \ \mathbf{old}\, ch.a := ch.a'] \end{aligned}$$

Evaluating this expression, we obtain the following SSC_1:

$$SSC_1 = sav.b = 0 \wedge ch.b = 0 \wedge ch.a = true$$

SSC_1 represents the system state constraint after the execution of m_1 with associated routine r_1.

4.3.3 Prover

For our discussion in this chapter, we will define a function *Prover* that acts as an oracle. Assume p is a predicate, then *Prover* is defined as follows:

$$Prover(p) = \begin{cases} T & \text{if } \models p \\ C & \text{if } p \text{ is contingent} \\ F & \text{if } p \text{ is a contradiction} \end{cases} \tag{4.3}$$

The function *Prover* returns *true* if p is valid, C if p is contingent and $false$ if p contains contradictory terms and evaluates to $false$. We will use this oracle in the remaining discussion of consistency checking and as will be evident, we will use a theorem prover in our tool to replace this oracle.

4.4 Symbolic Execution Step

Given a message in the dynamic diagram of $model(SD, DD)$, we will define a *symbolic execution step* as the semantics of a message execution. If a symbolic execution of a message is *successful*, then we consider the message to be consistent with the static diagram SD. Repeating this step for all messages in the dynamic diagram checks for *contractual consistency* of the $model(SD, DD)$.

Suppose the current state constraint is SSC_{i-1}, and assume that we now want to invoke a message $o_k \xrightarrow{m_i} o_{k+1}$ where o_k (o_{k+1}) is the source (respectively target) object in DD for message m_i. From the model mapping (e.g. Table 4.1), we obtain a corresponding routine r_i in the class associated with the target object o_{i+1}. Using (4.1), we may now compute SSC_i from r_i and SSC_{i-1}. We define *step* as follows:

$$step(SSC_{i-1},\ r_i,\ SSC_i) \text{ iff} \qquad \begin{array}{l} Prover((SSC_{i-1} \wedge args_i) \Rightarrow r_i.pre) = T \wedge \\ Prover(SSC_i) \neq F \end{array} \tag{4.4}$$

which defines under what conditions the symbolic execution of the original message $o_k \xrightarrow{m_i} o_{k+1}$ in the DD is successful. $args_i$ is the constraint on the

formal arguments of routine r_i (see Table 4.1). Note that we constructed (4.4) from information contained in both DD and SD (e.g. the contracts come from the SD).

(4.4) asserts that in order for an execution step to be successful, two conditions have to be met. First, the system state constraint before the execution of message m_i must satisfy the precondition of the routine r_i. Second, the post state SSC_i must be free of contradictory expressions and not evaluate to $false$. If both these requirements are met, then we consider the symbolic execution of m_i as successful. If any of the two conditions are not met, then a *contract violation* has occurred.

Continuing with our bank example, we have to complete a step that we omitted above. We have to check whether the execution of m_1 results in a successful step using (4.4).

We have the following values:

$$SSC_0 \ = \ (sav.b > 500 \ \wedge \ ch.b = sav.b \ \wedge \ ch.a = false)$$

$$r_1.pre = (ch.a = false)$$

$$SSC_1 = (sav.b = 0 \wedge ch.b = 0 \wedge ch.a = true)$$

As a result, the condition we have to check is the following:

$$step(SSC_0, r_1, SSC_1) = \quad Prover((SSC_0 \wedge args_1) \Rightarrow r_1.pre) \wedge Prover(SSC_1)$$

We know that $args_1 = true$ and using the values from above, we obtain:

$$Prover(SSC_0 \Rightarrow r_1.pre) = T$$

$$Prover(SSC_1) \neq F$$

We therefore conclude that m_1 can be symbolically executed without any contract violations.

4.5 Contractual Consistency of model(SD, DD)

In the section above we defined the meaning of a single *symbolic execution step*. A step is the symbolic execution of a single message in the dynamic diagram and a successful execution step was defined by (4.4).

We are now ready to define contractual consistency for $model(SD, DD)$ from our definitions above. Informally, a static and dynamic diagram are contractually consistent if each message in the dynamic diagram can be successfully symbolically executed. Formally, we define *contractual consistency (CC)* as follows:

$$\text{CC}(model(SD, DD)) \stackrel{def}{=} \forall i \mid 1 \leq i \leq n \bullet step\ (SSC_{i-1},\ r_i,\ SSC_i) \qquad (4.5)$$

where n is the number of messages in the DD. The above formula is the definition of contractual consistency for $model(SD, DD)$. As can be seen, $step\ (SSC_{i-1},\ r_i,\ SSC_i)$ must hold for all messages in the dynamic diagram in order for $model(SD, DD)$ to be contractually consistent. A contract violation at any time during the symbolic execution specifies that the two views are

contractually inconsistent.

In order to check for contractual consistency for our bank example, we have to symbolically execute the remaining three messages and if all of them can be successfully executed then we can claim that our $model(SD, DD)$ is contractually consistent.

In order to check whether message m_2 can be successfully executed, we have to check $step(SSC_1, r_2, SSC_2)$, which translates into the following two conditions (r_2 corresponds to the routine *deposit(y)* in Table 4.1):

(1) - $Prover((SSC_1 \wedge args_2) \Rightarrow r_2.pre) = T$

(2) - $Prover(SSC_2) \neq F$

We have the following values:

$SSC_1 = (sav.b = 0 \wedge ch.b = 0 \wedge ch.a = true)$

$r_2.pre = (y > 0)$

Incorporating the constraint on the argument of r_2 ($args_2$) into SSC_1, we obtain the following:

$SSC_1 \wedge args_2 = (sav.b = 0 \wedge ch.b = 0 \wedge ch.a = true \wedge y = 500)$

Using the above values for (1), Prover returns T. To check (2) we obtain SSC_2 after using the modifies clause of r_2, (4.1) and simplifying the expression using arithmetic:

$SSC_2 = (sav.b = 0 \wedge ch.a = true \wedge ch.b = 500)$

Prover returns *not* F for (2) as well. As a result, we can conclude that m_2 can

be successfully symbolically executed.

In m_3 we are transferring 200 from *ch* to *sav*. The correspond routine for this message is *transfer(x)*. We will not describe this step in detail as it is similar to the ones described above. This message can also be successfully executed and we obtain the following system state constraint after the symbolic execution of m_3:

$$SSC_3 = (sav.b = 200 \wedge ch.a = true \wedge ch.b = 300)$$

The last message to be symbolically executed is m_4. This message tries to withdraw an amount that is greater than 300 from *sav* (i.e. $args_4 = (i > 300)$). We have to therefore check the following two conditions as described in (4.4):

(1) $Prover((SSC_3 \wedge args_4) \Rightarrow r_4.pre) = T$

(2) $Prover(SSC_4) \neq F$

We have the following value for SSC_3 with the additional information regarding $args_4$:

$$SSC_3 \wedge args_4 = (sav.b = 200 \wedge ch.a = true \wedge ch.b = 300 \wedge i > 300)$$

The value for $r_4.pre$ is

$$r_4.pre = sav.b \geq i$$

Checking condition (1), we can see that *Prover* would not return T. The attribute $sav.b$ has a value of 200 at the time of the execution which is not greater or equal to i, which is greater than 300. This failure signals a *contract violation* and as a result, we conclude that $model(SD, DD)$ is *contractually*

inconsistent.

4.6 Special Case - Incorporation of Class Invariants

From page 36 of [Mey97] we have the following definition for a correct invariant for a class:

An assertion I is a correct class invariant for a class C if and only if it meets the following two conditions:

E1: Given a creation procedure *make* of C and attributes that have their default values, the execution of *make* with arguments that satisfy its precondition, yields a state satisfying I.

E2: Every exported routine of the class, when applied to arguments and a state satisfying both I and the routine's precondition, yields a state satisfying I.

Property E2 indicates that we may consider the invariant as being implicitly added to both the precondition and postcondition of every exported routine. It therefore has to be checked before and after each routine because of the indirect invariant effect (page 402 of [Mey97]). So in principle, we could enrich the pre and postcondition with the invariant and do without it.

Such a transformation is not desirable because of two reasons: First, it

would complicate the routine text. Second, an invariant transcends individual routines and applies to the class as a whole. As a result, the invariant will apply to a class written later as an extension to the current class (i.e. classes that inherit from the current class).

Incorporating invariants into the theory already developed is simple. Given a routine r, the precondition $r.pre$ was defined as the conjunction of all the require clauses of the routine. Likewise the postcondition was defined as the conjunction of all the ensure clauses.

To accommodate invariants, it suffices to redefine $r.pre$ as the conjunction of all the require clauses as well as the class invariant. Likewise the postcondition $r.post$ is redefined as the conjunction of all ensure clauses as well as the class invariant.

(4.4) can now be applied as is to check a successful step which also checks for the invariant. In addition, the invariant should be conjoined to the user defined initial constraint, if the routine is not a creation routine.

When this definition is implemented in a tool, the user has to be given the opportunity to specify whether or not SSC_1 represents the very start of an execution. Currently, BDT does not include implementation for invariants, but pre- and postconditions only. The addition of invariants can be achieved with little complexity.

Chapter 5

BON Development Tool

In this chapter, we will introduce and discuss the BON Development Tool (BDT). BDT was developed as a prototype to demonstrate the implementability of our approach to contractual consistency. BDT is a software modelling environment allowing for the construction of static and dynamic diagrams in BON. In addition, BDT incorporates most of the concepts of consistency checking that were discussed in the previous chapters. BDT was developed as an extension (plug-in) for Eclipse [IBM04] in order to simplify the development of a graphical application and at the same time make BDT available in an environment that is widely used. The graphical framework used for the implementation of BDT was GEF [IBM03]. Both, Eclipse and GEF are described in more detail below.

BDT consists of three parts: a static diagramming module, a dynamic

diagramming module and a consistency checking module. In this chapter, we will explain important concepts of the static diagramming module and consistency module. Since the structure of the dynamic diagramming module is similar to that of the static diagramming tool, its discussion will be left to the Appendix A.

When describing classes used to implement BDT, we will use the modelling language BON to describe our implementation. In BON, it is recommended to write class names in capital letters and separate words by a "_" (e.g. STUDENT_ACCOUNT). Such notation, however, is difficult to read when it appears outside of diagrams. As a result, we will use BON notation for classes in diagrams, but regular Java naming convention for explaining those diagrams.

5.1 Eclipse and GEF Framework

5.1.1 Eclipse

Eclipse is an extensible platform for tool integration and a wide range of tools have been built on that platform. In its roots, Eclipse is an IDE for software development which allows for plug-ins to be plugged into it to increase its functions. Eclipse is an open-source project that has created a community around itself. This community is very diverse and consists of users of Eclipse-based products, writers of tool extensions and researchers exploring new ways

to use Eclipse [IBM04].

Eclipse provides a platform that allows diverse tools to inter-operate, often in ways that tool writers initially did not imagine. The Eclipse Platform consists of numerous plug-ins itself and many more are available for free and commercially. Eclipse is a collection of places-to-plug-things-in (extension points) and things-plugged-in (extensions or plug-ins). The power of Eclipse lies in the possibility of many plug-ins working together and providing functionality that was not expected from the original plug-in developers.

BDT is one such plug-in that provides modelling, consistency checking and model execution functionality to Eclipse. Further, BDT is expected to work with Eiffel Development Tool (EDT) [Mak03] to provide seamless forward and reverse-engineering from Eiffel to BON and vice-versa. More information regarding Eclipse can be obtained from various online sources and [IBM04].

5.1.2 Graphical Editing Framework

The Graphical Editing Framework (GEF) [IBM03] allows developers to create a rich graphical editor from an existing application model. GEF uses the SWT-based drawing plug-in Draw2d to create a graphical environment within Eclipse. The developer can take advantage of the many common operations provided in GEF and/or extend them for the specific domain. GEF employs an MVC (model-view-controller) architecture, which enables simple changes to

be applied to the model from the view. GEF is completely application neutral and provides the groundwork to build almost any application, including but not limited to: flow builders, GUI builders, UML diagram editors (such as work-flow and class modelling diagrams), and even WYSIWYG (What You See Is What You Get) text editors like those to write HTML code.

GEF is divided into two major plug-ins, each with their own subset of features: GEF, and Draw2d. The Draw2d plug-in provides a stand-alone rendering and layout package for an SWT Canvas. It includes common shapes and layouts that can be assembled to render almost anything. The GEF plug-in uses Draw2d for rendering, but adds a rich MVC editing framework on top of it. This framework helps ensure that most graphical applications look and behave in a similar way. The plug-ins are written in Java with no native code and thus may be ported to any platform supported by Eclipse.

GEF is used extensively throughout BDT. We use this plug-in for all drawing and display capabilities of our application. The GEF classes and functions that are used are explained in the subsections below. For more information regarding GEF refer to [IBM03].

5.2 Static Diagramming Module

As explained above, the BDT application consists of three parts. The first part is the static diagram drawing module. It was developed to support the

construction of BON static diagrams and specifically allow the addition of contracts in the diagram that can be used later for consistency checking. Figure 5.1 shows a screen shot of the static diagram module. In subsections below, we will concentrate on the Model-View-Controller (MVC) we used, and we will explain the various aspects of this approach.

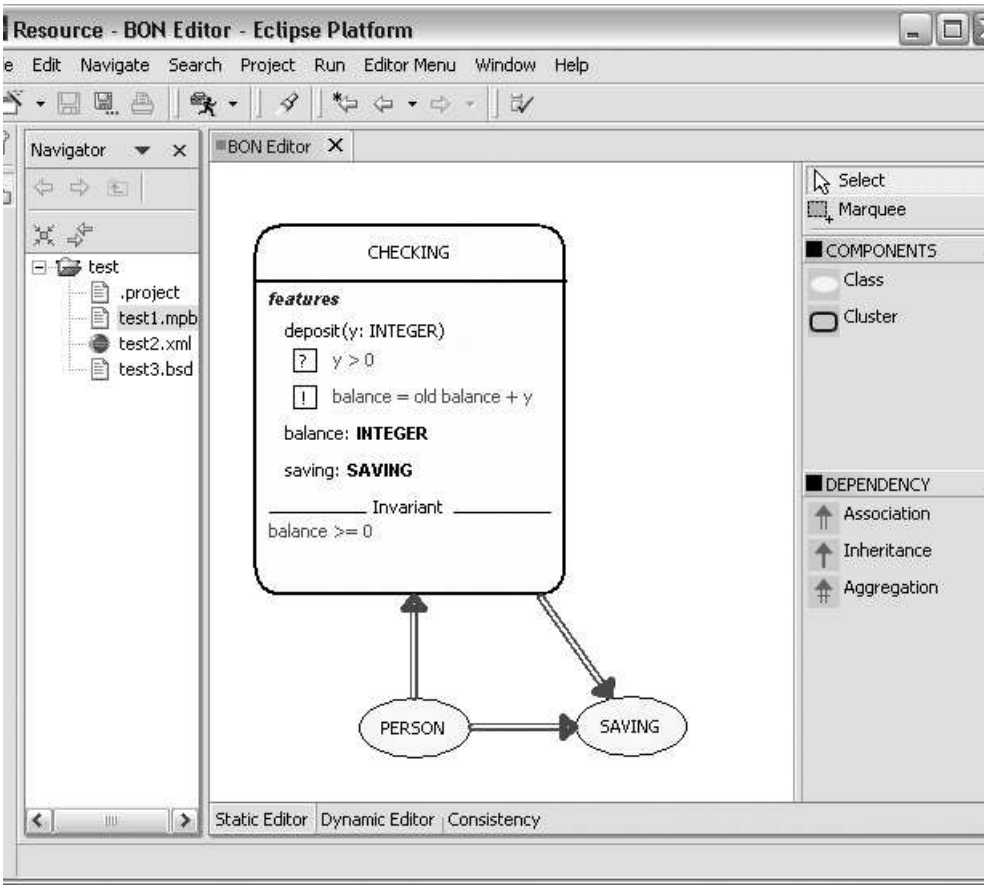

Figure 5.1: A screen shot from the static diagramming tool of BDT.

5.2.1 Model

The model used to represent a static diagram has a simple structure. A BON static diagram consists of elements such as class ellipses and containers such as clusters. Further, it can contain dependencies between classes and features and routines within classes.

Figure 5.2 shows the class structure of five classes within the model. The class *BonTopDiagram* acts as the top diagram where BON elements are added and therefore behaves like a container. It contains features for adding and removing children which are BON elements. This class does not contain any implementation as various containers inheriting from *BonTopDiagram* might implement features for adding and removing children differently. Two classes inheriting from *BonTopDiagram* are *StaticDiagramModel* and *ClusterModel*. The class *StaticDiagramModel* acts as the implemented top diagram in the system where all the items in a static diagram are added to. It implements the features from its parent class and includes additional features for reporting changes to itself to the classes responsible for maintaining the view. Since a cluster also contains other elements within itself (such as classes), *ClusterModel* also inherits from *BonTopDiagram*.

BonElement is a deferred class representing classes and clusters in the system. A *BonElement* has a name, keeps track of incoming and outgoing dependency arrows and can be added to an instance of *BonTopDiagram* as

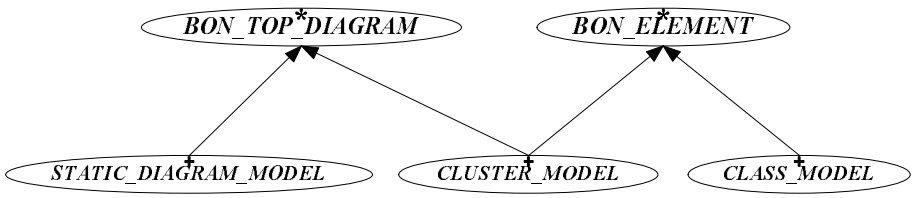

Figure 5.2: Class relationship between *BonTopDiagram*, *BonElement*, *StaticDiagramModel, ClusterModel* and *ClassModel*.

a child. Figure 5.2 shows that *ClassModel* and *ClusterModel* inherit from *BonElement*. As a result, instances of *ClassModel* can be added to instances of *ClusterModel* and *StaticDiagramModel* since both of these classes act as containers. This class structure also ensures that instances of *ClusterModel* can be added to instances of the same class since a *ClusterModel* is both a container and an element.

Another component in BON static diagrams are dependencies. These include inheritance, association and aggregation dependencies. Figure 5.3 shows the class structure around these items. *DependencyModel* represents a dependency between two instances of *BonElement*. It contains features for distinguishing the source and target *BonElement* and for attaching and disconnecting connections. The three inheriting classes *InheritanceModel*, *AssociationModel* and *AggregationModel* are used to distinguish between the various possible dependencies and implement features specific to each type of dependency.

Class features are another component in a static diagram. These components belong to specific classes. Figure 5.4 shows the structure of the classes

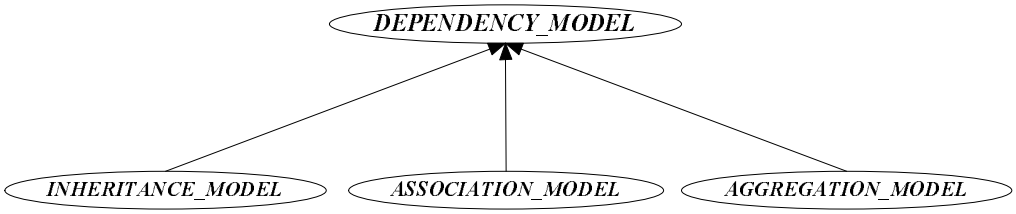

Figure 5.3: Class structure of dependencies in a static diagram.

involved. *FeatureModel* defines a general structure for all features. It includes features such as the name and status (implemented, deferred) of a feature. *FunctionModel* defines the structure for all features that are also routines, such as commands and queries. This class includes routines for adding and returning contracts and feature-arguments. *AttributeModel* inherits from *FeatureModel* only, whereas *FunctionModel* and *CommandModel* inherit from both, *FeatureModel* and *RoutineModel*.

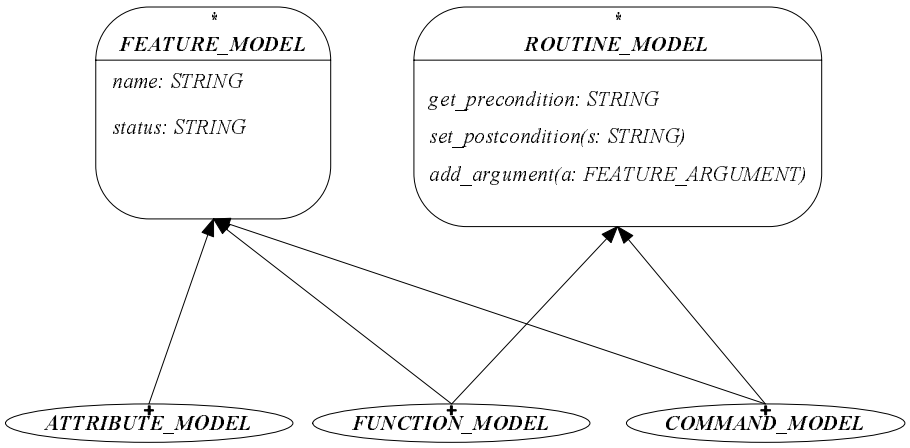

Figure 5.4: Class structure for features in static diagrams.

The classes described above are the most important ones in the model of

the static module. Many more exist, but are not relevant to this discussion. Figure 5.5 shows a more comprehensive class structure for the static module. As can be seen, *ClassModel* contains a list of *DependencyModel*. This list represents all dependencies that have their source in that class. Further, it has a list of *GroupFeatureModel*. This class represents a grouping of features within a BON class diagram. An instance of *ClassModel* has various instances of *GroupFeatureModel* and these instances in turn contain instances of *FeatureModel* and *RoutineModel*. Figure 5.5 also contains the class *BooleanExpression*. This class represents a Boolean expression in a contract of a routine which in this case can include pre and postconditions. This class will be explained in more detail below when discussing the module for consistency checking.

5.2.2 View

The view in the static module consists of figures from the draw2D package within the GEF framework. All figures are customized to meet the expectations of our application. Figure 5.6 shows the class structure for figures representing classes and clusters.

ClassFigure and *ClusterFigure* both inherit from *Label*, which represents the name of the class or cluster respectively. This inheritance makes the editing of a name by double clicking on the name easier to implement. Both classes

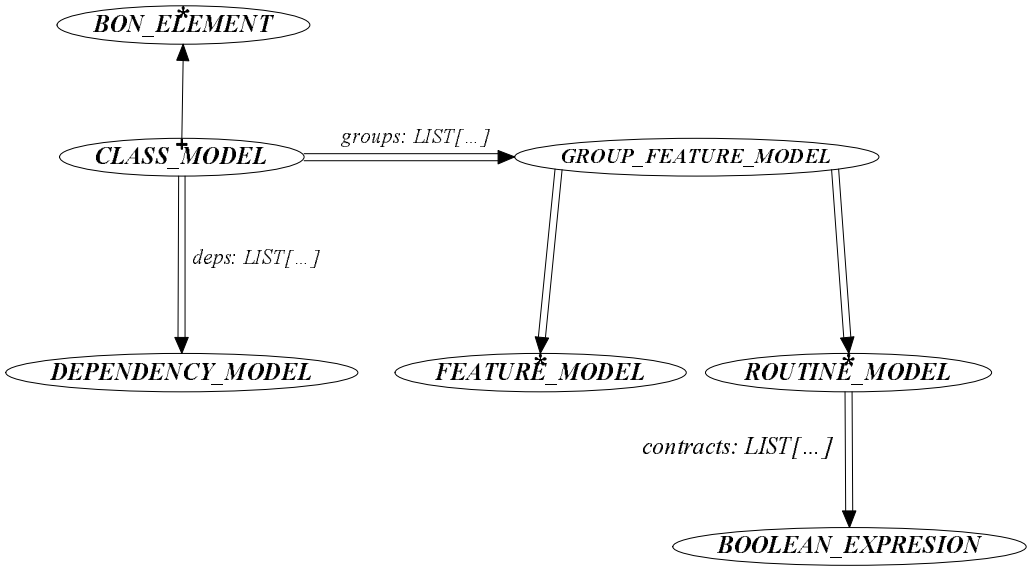

Figure 5.5: Partial class structure for the static model. Two new classes *BooleanExpression* and *GroupFeatureModel* are shown.

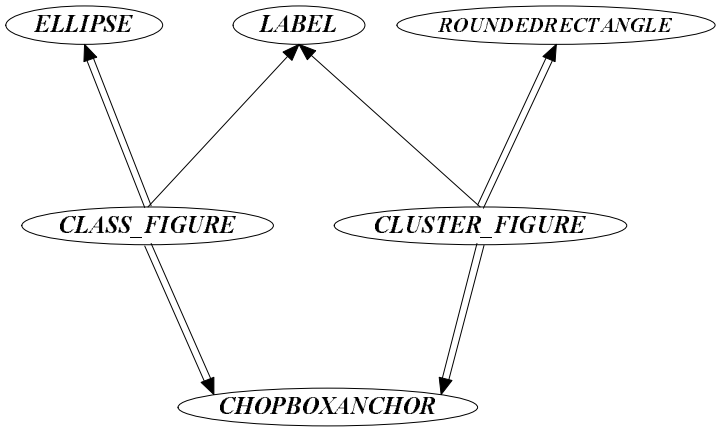

Figure 5.6: Class structure around *ClassFigure* and *ClusterFigure*. Both inherit from *Label* and use *ChopBoxAnchor*. Their shapes are defined by *Ellipse* and *RoundedRectangle*.

have a *ChopBoxAnchor*, which acts as the anchor to which all dependency lines connect. Finally, *ClassFigure's* shape is defined by an *Ellipse* and that of *ClusterFigure* by a *RoundedRectangle*.

Figure 5.7 shows the class structure for visually representing dependencies such as aggregation, association and inheritance. The class *PolyLineConnection* is the draw2D class for representing lines in GEF. Each of the inheriting classes redefines the *paint* routine to have the appropriate shape for the dependency. Further, the feature *setTargetDecoration* of *PolyLineConnection* is used to define the various arrow heads of the three inheriting classes.

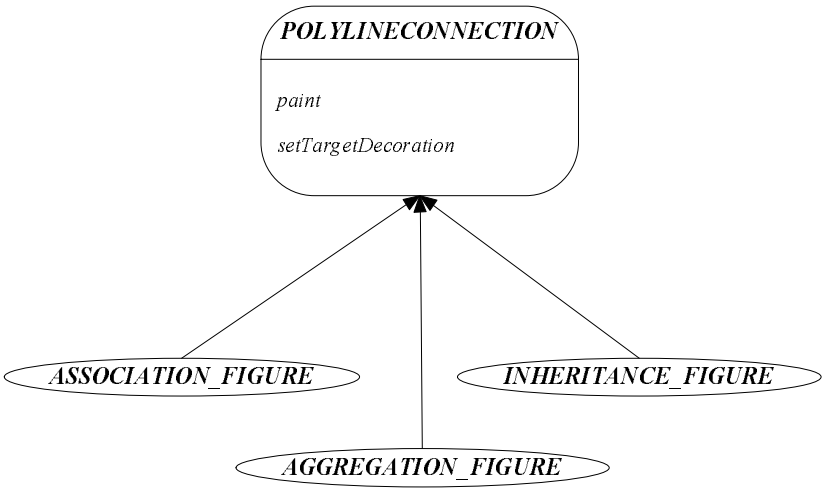

Figure 5.7: Class structure of figures representing dependency lines.

Figure 5.8 captures the class structure necessary to visualize feature groups and features. All four classes *FeatureGroupFigure*, *QueryFigure*, *CommandFigure* and *AttributeFigure* inherit from *RectangleFigure* which represents the

bounds for the feature and feature groups. When drawing classes, the rectangles are not visible and only act as helpers to calculate bounds and dimensions. All four classes also use *Label* to visualize the text in these elements.

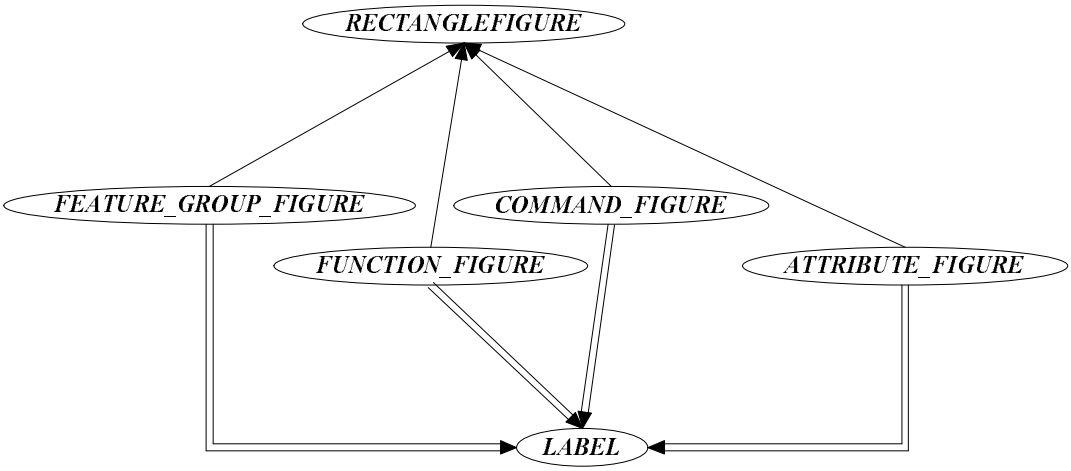

Figure 5.8: Class structure of figures representing feature group and features.

5.2.3 Controller

In GEF, the controller in the MVC paradigm is represented by an *EditPart*. An *EditPart's* responsibility is to ensure that model and view are consistent and to reflect any changes from the view to the model and vice versa. An *EditPart* represents a single conceptual object with which the user can directly or indirectly interact. In general, an *EditPart* will directly represent something in the model. The *EditPart* itself is not visible to the user, but will present itself through its view, as described above. The most fundamental role of an *EditPart* and the one that is usually implemented first, is to populate the

viewer with visuals, and to maintain those visuals as the model changes. The second and more interesting role of an *EditPart* is to perform graphical editing. Graphical editing is defined here as:

- **Manipulating the Model** - *EditParts* must manipulate the model by creating Commands in response to Requests.
- **Display Feedback** - *EditParts* should show feedback during complex interactions with the user such as drag and drop.
- **Delegation** - Either of the previous two jobs can be delegated to additional *EditParts*.

Figure 5.9 shows classes involved in controlling the view and model of classes and clusters. *AbstractGraphicalEditPart* and *NodeEditPart* are classes from the GEF framework that act as the controller in MVC. An instance of *AbstractGraphicalEditPart* represents an EditPart whose view is a figure as described above. A *NodeEditPart* is a specialized *EditPart* that supports both target and source connections (these will be described below). *BonContainerEditPart* and *BonNodeEditPart* are BDT implementations and correspond to *BonTopDiagram* and *BonElement* in the model. They are fully implemented as *ClusterEditPart* and *ClassEditPart*.

As mentioned above, *EditParts* correspond to the controller in our application that ensures that model and view are always consistent. Figure 5.10

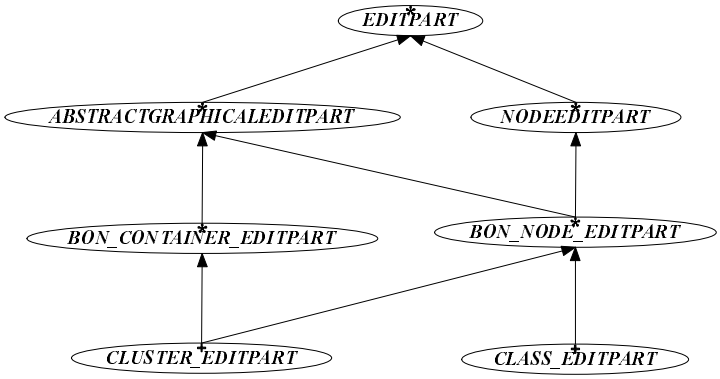

Figure 5.9: Class structure for *EditParts* controlling the view and model of classes and clusters in the static diagramming module.

shows the relationship between *ClassEditPart* and *ClusterEditPart* with their corresponding models and views. Each *EditPart* has features for setting and getting its model and view. An instance of *ClusterEditPart* has also an instance of *ClusterModel* as its model and an instance of *ClusterFigure* as its figure. *ClusterEditPart* ensures that these two are consistent and that changes in one of them are reflected in the other. The same applies to *ClassEditPart*, *ClassFigure* and *ClassModel*. The main features that are inherited from EditPart and implemented to achieve this consistency are *getModel*, *createFigure* and *updateFigure*.

As described above, the top model for the static diagram is *StaticDiagramModel*. Figure 5.11 shows the classes that represent the canvas on which the elements of a static diagram are drawn. *BonDiagramEditPart* is the controller which registers any additions to the canvas. The figure representing the canvas

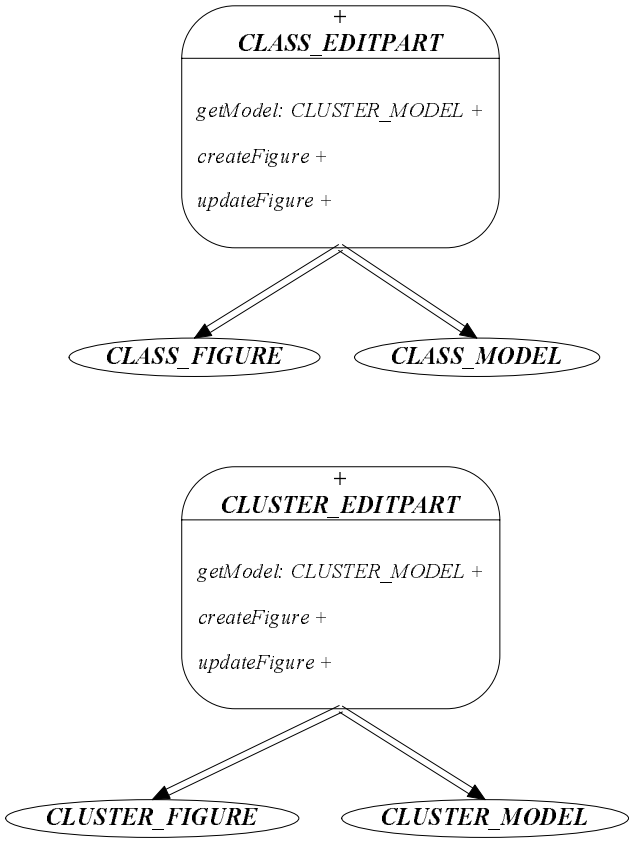

Figure 5.10: Relationship between model, view and controller. EditParts are the controller and each have a model and a figure.

is an instance of *FreeFormLayout*. Any changes to the view are reflected to the model, *StaticDiagramModel*. The class *BonDiagramEditPart* inherits from *BonContainerEditPart* as it behaves like a container where elements can be dropped to and removed from.

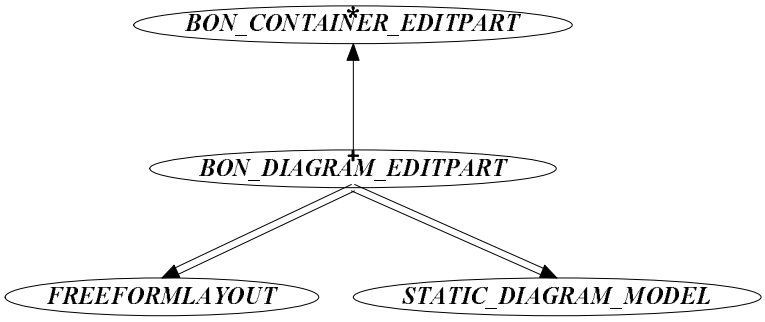

Figure 5.11: vis responsible for maintaining consistency between the view *FreeFormLayout* and the model *StaticDiagramModel*. It inherits from *BonContainerEditPart* which was described above.

The last controller we are going to discuss in this section is that of dependencies. There are three different models and views for dependencies and as a result we have three *EditParts*. *AggregationEditPart*, *AssociationEditPart* and *InheritanceEditPart* inherit from *BonConnectionEditPart* which is seen in Figure 5.12. *BonConnectionEditPart* inherits from *AbstractConnectionEditPart* and implements the abstract routines for this class. An instance of *BonConnectionEditPart* has an instant of *DependencyModel* which acts as the model for a dependency. Each of the three connection *EditParts* have their own figure, which can also be seen in Figure 5.12.

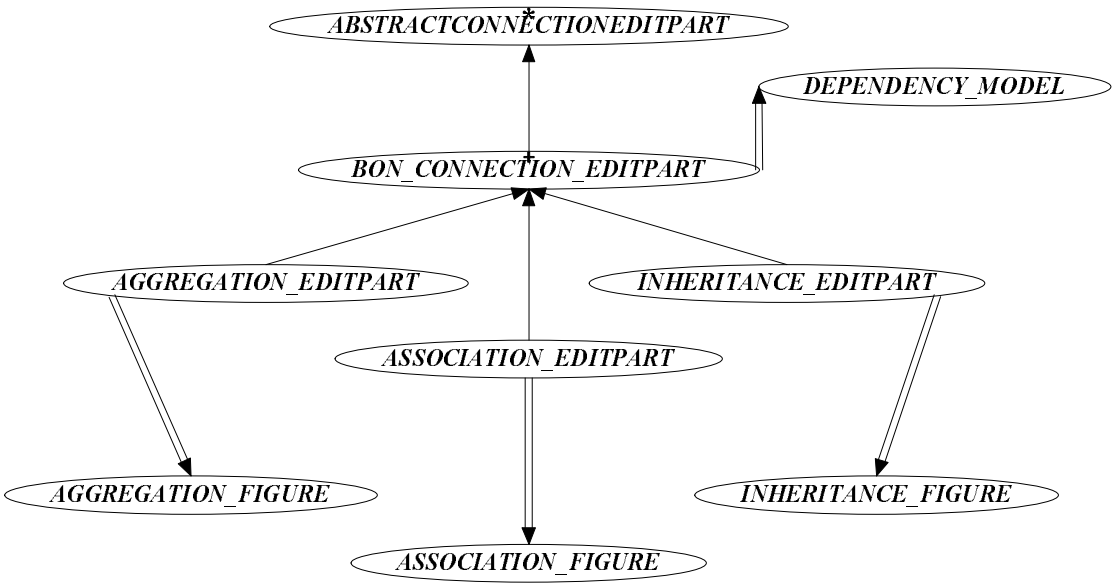

Figure 5.12: Class structure of *EditParts* involved in controlling the view and model of dependencies. *AssociationEditPart*, *AssociationEditPart* and *InheritanceEditPart* inherit from *BonConnectionEditPart* which in turn inherits from *AbstractConnectionEditPart*.

In this section, we did not mention *EditParts* involved in controlling models and figures that represent class features. This class structure is very similar to that of *ClassEditPart* and *ClusterEditPart* and it was therefore omitted. The interested reader can inspect the code to find out more about those *EditParts.*

5.2.4 BON Static Editor

In the sections above, we described the model, view and controller. These three, however, have to be started at some point and in this section we will discuss the Editor, the class that is responsible to start the static diagram application and set all initial values. Figure 5.13 shows the structure of classes that are involved in maintaining an editor and initializing its values.

The class *EditorPart* is the base abstract implementation of all Editors. *GraphicalEditor* extends *EditorPart* and represents an editor for graphical use. Each *GraphicalEditor* has an instance of *GraphicalViewer* which is responsible for showing the contents of the editor. Figure 5.13 also shows that each *GraphicalViewer* has an instance of EditPartFactory. Whenever an *EditPart* in that viewer needs to create another *EditPart*, it can use the Viewer's factory. In our case, EditPartFactory is set up in a way to create the right *EditParts* based on the model that is added to the editor. This structure follows the Factory design pattern.

Once an instance of *BonStaticEditor* is created, it sets its own content

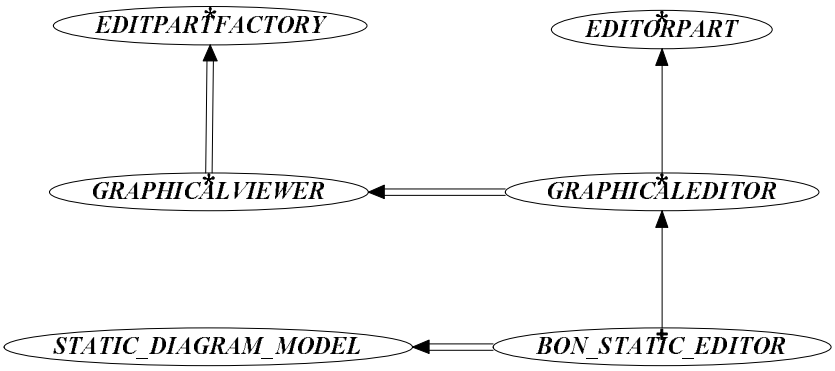

Figure 5.13: Class structure of EditParts involved in controlling the view and model of dependencies. *AggregationEditPart*, *AssocationEditPart* and *InheritanceEditPart* inherit from *BonConnectionEditPart* which in turn inherits from *AbstractConnectionEditPart*.

by creating a new instance of *StaticDiagramModel*. This model is passed to the *GraphicalViewer* and then to the *EditPartFactory*. It is the factory's responsibility to create the correct *EditPart* for this model. In our case, the factory would create an instance of *BonDiagramEditPart*. Once this *EditPart* is created, it creates an instance of *FreeFormLayout* to act as its figure. After that, BON_DIAGRAM_EDITPART waits for user input.

5.3 Consistency Module

The module for checking contractual consistency consists of a small number of classes. *ConsistencyComposite*, *ConsistencyCheck* and *SSC* are the three main classes in this module. In addition, the class *BooleanExpression* from the static diagram module is of importance to this module. Figure 5.14 shows the

classes and their interactions in the consistency module. *ConsistencyComposite* is the class that holds the visual elements such as buttons and text boxes for user interaction. *ConsistencyCheck* is the main class responsible for coordinating consistency checking and has references to *StaticDiagramModel* and *DynamicDiagramModel*, which represent SD and DD in Chapter 4 respectively. An instance of the class *SSC* represents the system state constraint and consists of a list of predicates.

In this section, we will discuss how contracts and the system state constraint are represented, how BDT interacts with the theorem prover and how the system state constraint is modified. In addition, we will outline the sources of incompleteness in our tool and any features from the theory that were not implemented in BDT.

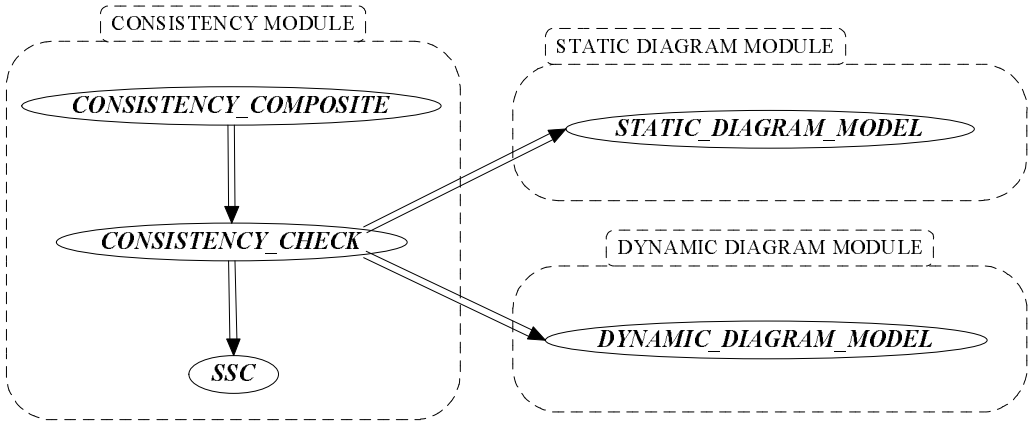

Figure 5.14: *ConsistencyCheck* has references to *StaticDiagramModel* and *DynamicDiagramModel*. It also has one instance of *SSC*.

5.3.1 Boolean Expression

Checking for contractual consistency is dependent on contracts of classes. As a result, it was important to have a well-defined structure for contracts in order to simplify their manipulation. BDT currently supports a subset of legal contracts as defined by [WN95]. The following forms of contracts are supported:

- The primitive types supported in a contract are INTEGER, REAL, BOOLEAN, STRING and ARRAY[G]
- Composite types can be composed from supported primitive types (e.g. SAVING). Composites of composites are supported as well.
- Contracts involving equality and inequality of the supported types, e.g., $a = 5$ and $a \leq b$.
- Query calls that return a supported type, e.g. $a = get\_balance$.
- Contracts involving one level of multi-dot expressions, e.g. $a = sav.balance$. The current application does not deal with multi-dot expressions and it needs improvement in this area for the future.
- Four basic operations (+, –, *, /) for INTEGER and REAL; negation, conjunction, disjunction and implication for BOOLEAN, e.g. $a = sav.balance + x$ and $a = x\ implies\ b$.

- Three operations are supported for arrays. These are *insert(i: INTEGER, x: G)*, *count* and *item(y: INTEGER)*. *item(y)* returns the item at position y in the array; *insert(y)* inserts an item at position y and *count* returns the number of items in the array. Support for other features can easily be added, but we chose to limit our implementation to these three array operations to avoid unnecessary complexity.

- Array expressions that hold one of the supported primitive or composite types, e.g. $a.item(5) > 15$.

Class *BooleanExpression* represents a predicate that asserts a condition on *one* attribute. As a result, a precondition of a routine can consist of several instances of *BooleanExpression*. Figure 5.15 shows the most important features of *BooleanExpression*. This class is deferred and consists of unimplemented features *getVariables*, *getPrefix*, *renameVariables*, *replace* and a few others that are not relevant for the current discussion. The function of each routine is as follows:

- *getVariables:* Skims through the expression and returns a list of all variables in the Boolean expression removing all key words, constants and feature names.

- *getPrefix:* Returns this Boolean expression in prefix format. This routine is used to communicate with the theorem prover, which accepts

predicates in prefix form only.

- *renameVariables (s: String):* Renames all variables *var* in the expression to *s.var*. This feature is used to rename all unqualified variables *var* to *s.var* where *s* is the object name.

- *replace (s, newS):* Replaces all occurrences of *s* in the current instance where it appears as **old** *s* with the new variable *newS*. This feature is used to remove all occurrences of old variables and is used for evaluating post system state constraints.

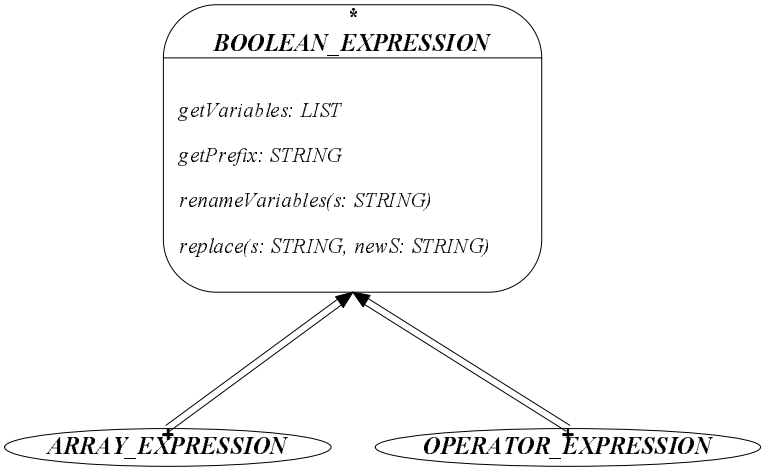

Figure 5.15: *BooleanExpression* is the class representing Boolean expressions. It is a deferred class and is implemented by *ArrayExpression* and *OperatorExpression*.

There are two classes that implement *BooleanExpression*: class *ArrayExpression* and class *OperatorExpression*. *ArrayExpression* is used to represent Boolean expressions that involve arrays whereas *OperatorExpression* is used for

all other forms of supported Boolean expressions. These expressions cannot contain array expressions. In our current implementation, operator expressions consist of feature calls and expressions involving equality and inequality operators.

5.3.2 Mapping between Messages and Routines

As was discussed in Chapter 4, a mapping between messages in the dynamic diagram and routines in the static diagram has to be specified by the user (see Table 4.1). This mapping represents the overlap of information between the two views and this information is used to check for consistency, or in our case, contractual consistency.

In BDT, the mapping occurs in the dynamic diagram by choosing a routine for each message. The user can specify a routine through the messages *Properties* interface. In this interface, the user will be presented with a list of legal routines that can be associated with the chosen message. The legality of routines depends on the routines that are exported to the source object from the target object. Figure 5.16 shows a screen-shot of this interface from BDT.

If contractual consistency is being checked between a static diagram and dynamic diagram for which not all mappings have been specified, an error message will be displayed to the user and she will be prompted to specify the necessary mappings.

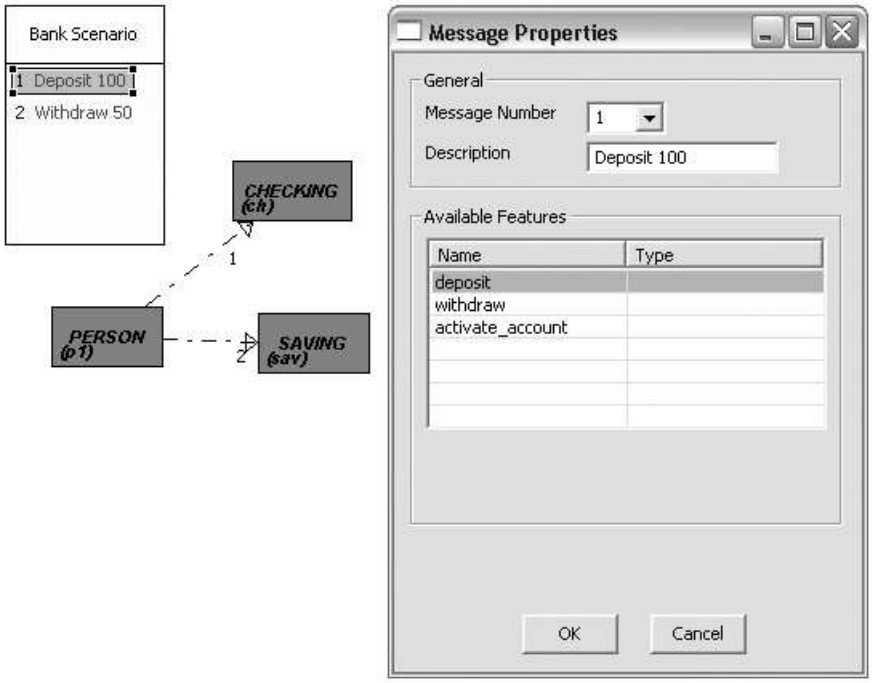

Figure 5.16: The interface used to specify a mapping between messages and routines.

5.3.3 Theorem Prover - Simplify

In section 4.3.3 we described the function *Prover* that we used as an oracle for the development of our theoretical approach to contractual consistency checking. The definition of *Prover* was given by (4.3). In order to implement BDT, we have to replace this oracle by a theorem prover that provides us with similar abilities.

There are several theorem provers available, both commercially and open-source. *PVS* [OSR01] is one of the most famous and widely used theorem provers. [POB02] suggests using *PVS* for the approach presented in that paper. [Kri00] also suggests the use of *PVS* for checking the consistency between UML diagrams. We have decided to use the theorem prover *Simplify* [DNS03] which is the prover used in the Extended Static Checker ESC/Java [LNS00] for the following reasons:

- *Simplify* is available for all development platforms including Windows and Linux. *PVS*, for example, is not available for Windows.

- *Simplify* tries to find simple proofs *rapidly* if they exist. This provides an advantage for software developers who are not interested in long waiting times when testing models.

- *Simplify* simplifies the use of the multi-dot notation that is extensively used in object-oriented programming. A variable *sav.balance* can easily

be supplied to *Simplify* as $|sav.balance|$. This is a major simplification compared to other theorem provers.

- An important property of the theorem-prover is that it is refutation-based: to prove a formula P, it tries to satisfy the negation of P. If the prover finds a set of variable assignments that satisfy the negation of P, those variable assignments are returned as feedback. This feature is very useful as it can be used to give valuable feedback about contract violations. Consider the predicate $(x = -1) \Rightarrow (x > 0)$. The following are sample outputs from *Simplify*:

```
(IMPLIES (EQ x -1)(> x 0))
Counterexample:
context:
     (AND
          (EQ x -1)
     )
1: Invalid.
```

The first line represents the predicate that was sent to *Simplify* to be proved. *Invalid* indicates that the predicate could not be proved. The section *context* specifies the counter example that could be used to satisfy the negation of the predicate. In this example, if x was set to -1 then the above predicate would evaluate to $false$. Note that *Simplify* uses the prefix notation. It is important to note that *Invalid* does not necessarily mean that the predicate is not a theorem. In cases when quantifications are used, an output of *Invalid* with a counter example could still mean

that the predicate is a theorem. This is one source of incompleteness in *Simplify*.

The next example also evaluates to $Invalid$, but *Simplify* has not found a counter example to return.

```
(AND (EQ x 1) (NEQ x 1))
Counterexample:

3: Invalid.
```

Simplify reports $Invalid$. In this case, the tested predicate $x = 1 \wedge x \neq 1$ really is not a theorem — it is in fact a contradiction. However, $Invalid$ just means that the theorem prover could not definitively prove the theorem. But it does not definitively tell us that we are dealing with a contradiction. *Simplify* returns *Invalid* without a counter example if it did not exhaust all search paths, but also could not come up with an assignment that satisfies the negation of the supplied predicate. Note, however, that there is a way to clarify the situation using the theorem prover. If we submit $\neg(x = 1 \wedge x \neq 1)$ we get back $Valid$. Thus we now have definitive evidence that $x = 1 \wedge x \neq 1$ is a contradiction. With regard to the oracle of the previous chapter, we can thus assert that $Prover(x = 1 \wedge x \neq 1) = F$.

The next example evaluates to *Valid*.

```
(IMPLIES (EQ x 10)(> x 0))
1. Valid
```

The authors of *Simplify* claim that an output of *Valid* guarantees that the supplied predicate is a theorem.

The implications of the above three cases in regards to the response from our tool are discussed in the next section.

Incompleteness of BDT

We use *Simplify* in BDT to check for a successful symbolic execution step as defined by (4.4). *Simplify* is only an approximation to our oracle *Prover* described in Section 4.3.3. This theorem prover was "engineered to be automatic rather than complete" [FLL+02] and as a result this theorem prover is incomplete. In fact, due to Gödel's completeness Theorem, all theorem provers are incomplete and a prefect oracle cannot be constructed. Due to this restriction, BDT is also incomplete as it can provide spurious warnings and label a model as contractually inconsistent, but in fact the model could be contractually consistent.

Simplify has two possible outputs given a predicate P:

- *Valid*;
- *Invalid* (often with a context).

Given a predicate P as the input to *Simplify*, the output *Valid* signals that P is definitely a theorem (assuming that the theorem prover was implemented

	Valid	Invalid
$Simplify(X)$	$Prover(X) = T$. The precondition is enabled under the current state constraint.	*Incomplete*. Context usually provides enough feedback to help the designer decide whether the contractual design is in error or whether the theorem can be proved and the precondition is enabled.
$Simplify(\neg Y)$	$Prover(Y) = F$. SSC_{i+1} is a contradiction; thus there is no poststate with which to continue the symbolic execution.	*Incomplete*. Context usually provides enough feedback to help the designer decide whether there is or is not a consistent poststate.

Table 5.1: Table explaining the mapping between the output from *Simplify* and *Prover*. $X = SSC_i \rightarrow r_i.pre$ and $Y = SSC_{i+1}$. Note that we use $\neg Y$ for the second condition.

correctly). An output of *Invalid* signals incompleteness as it cannot be determined whether P is indeed not a theorem. In these cases, *Simplify* tries to find a counter example that would make P evaluate to $false$. There are, however, cases in which the theorem prover does not return a counter example as it did not find one (even though one does exist). Table 5.1 shows how *Simplify* is used in order to evaluate (4.4).

If $Simplify(SSC_i \rightarrow r_i.pre)$ returns *Valid* then the precondition of $r_i.pre$ is satisfied and the user is given the appropriate feedback. However, if the theorem prover returns $Invalid$ then it is the user's responsibility to decide based on the context returned whether a contract violation will occur. Similarly, if $Simplify(\neg SSC_{i+1})$ returns $Valid$ then the system state constraint SSC_{i+1}

is a contradiction and the user is notified. An output of *Invalid* can be used with the context to determine whether the system state constraint is in fact a contradiction.

Interaction with Simplify

BDT interacts with Simplify through input and output files. The formula to be proved is written into an input file and then Simplify is invoked on that file. After the execution, the result is read from an output file.

The execution of Simplify is achieved through running the following command in Java:

```
Runtime.getRuntime().exec(``cmd.exe /E:1900 /C
    simplify.exe < file.in > file.out'');
```

where *simplify.exe* is the executable file for the theorem prover in a Windows environment[1], *file.in* contains the predicate to be proved and *file.out* will contain the output from *Simplify*. As described, once this command is executed, BDT reads *file.out* to extract useful feedback that can be shown to the user.

Modification of Response from Simplify

The extraction of relevant information as feedback is dependent on the programmer. No theory was developed during this thesis that dictates which

[1]Other files must be used for other environments. Ideally a commercial release of BDT should come with *Simplify* files for all supported platforms.

result should be communicated back to the user. As (4.2) shows, variables can be renamed several times during the execution of messages. As a result, the output from *Simplify* can be very extensive and overburden the developer.

We use (4.2) in our implementation in order to preserve information regarding past states of variables. This information can be used to provide the developer with targeted feedback regarding the transition of every variable. In the current implementation of BDT, however, we do not use this information and project out information regarding past states when presenting the user with feedback. As a result, the user only sees the *current* state of attributes at the time a contract violation occurred. It is possible to define more elaborate rules for displaying feedback to the user. This is left as a future improvement.

The following is an output from BDT after checking for successful symbolic execution of some message $Message1$. The output is as follows:

```
Precondition (saving.active = false) from Message1
was not satisfied.
The following terms caused this error:
(NEQ saving.active false)
(EQ balance 0)
```

The first line indicates the exact term that caused the precondition to fail. In this case, it was $(saving.active = false)$. Further, the output identifies the message whose associated routine failed, i.e. $Message1$. After that, useful terms that can help the developer in finding the source of the error are listed. The terms presented as feedback are a subset of the terms that made $(SSC_i \rightarrow$

$r_i.pre$) evaluate to $false$. In this case, the developer knows that at the time of the contract violation ($saving.active \neq false$) and that ($balance = 0$). This information can now be used to find the source of the violation. At this stage, it is the user's responsibility to recognize the most relevant expressions and change the model accordingly.

5.3.4 Representation and Modification of System State Constraint

In BDT, we use the class *SSC* to represent the current system state. *SSC* has three main features that are used to update the system state constraint as shown in Figure 5.17. The routines and their roles are explained below:

- *getVariables* is used to get all the variables in the state constraint. This feature is used to rename variables after a message has been successfully executed and a new system state constraint has to be constructed.

- *addExpression (BooleanExpression exp)* adds a new Boolean expression to the system state constraint. This feature is mainly used during the beginning of the simulation when the initial, user-specified constraint is added.

- *seqCompose (LIST newExp)* is the main feature of this class. This routine implements (4.2). *seqCompose* takes as argument *newExp* which is

a list of Boolean expressions. *newExp* represents the postcondition of a routine whose precondition has been satisfied by the current system state constraint. As outlined in (4.2), *seqCompose* must rename attributes in both, *newExp* and the current SSC. For each variable v that is being modified, this routine has to find a fresh variable v'. In this routine, we use the Java construct

```
UID uid = new UID();
```

to find a unique identifier in the system. This value is then attached to the variable v to give us a fresh variable v'. This is repeated for each variable that is being modified.

In our discussion in Chapter 4 we presented the idea of a *modifies* clause that specifies all variables that a routine can change. This clauses is used in (4.2) to obtain the variables that are being modified and therefore have to be renamed. In BDT, the user can specify the modifies clause in the static diagram together with the contracts of a class.

5.3.5 Process of Checking Contractual Consistency

Given a static and dynamic diagram in BON, BDT checks for contractual consistency between the two views as follows:

1. BDT checks that all messages in the dynamic diagram have a mapping

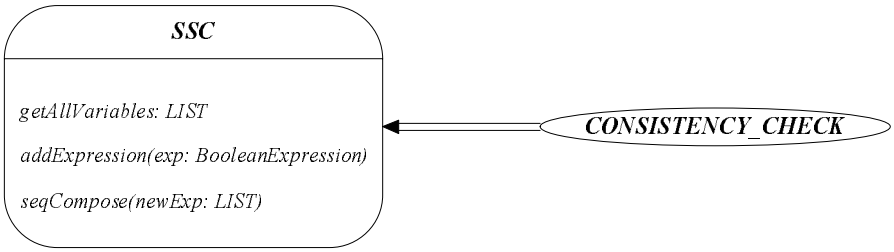

Figure 5.17: SSC represents the system state when executing messages. *ConsistencyCheck* has a reference to SSC.

to routines in the static diagram. If this is not the case, an error message is displayed.

2. The user is given the opportunity to specify an initial constraint. This constraint becomes SSC_1, otherwise it is $true$.

3. If the routine r_i associated with message m_i contains formal arguments, then the user is asked for a constraint involving the arguments. This constraint is added to SSC_i.

4. For each message m_i, BDT checks whether SSC_i implies the precondition of r_i which is associated with m_i. For this step, the condition to be checked is converted to prefix format and checked by using *Simplify*. If the theorem prover returns $Valid$, then BDT continues to the next step, otherwise an error message with an edited response from *Simplify* is returned to the user to signal a possible contract violation.

5. BDT uses the routines in the class SSC to construct the new system

state constraint according to (4.2).

6. Using the new state constraint, *Simplify* is used again to check the second condition of (4.4). Table 5.1 is used to evaluate the output from the theorem prover.

7. If both conditions of (4.4) are satisfied, BDT notifies the user that the message was successfully symbolically executed.

8. If all messages are successfully executed, the two views are considered contractually consistent.

5.3.6 Limitations of the Tool

Our main objective for BDT was to develop a prototype tool that illustrates the implementability, in principle, of the theoretical work on contractual consistency. Our objective was not to develop a full-strength tool that catches all errors and provides all features of a modelling tool. As a result, the following features were not implemented in BDT:

- Class invariants were not implemented in BDT as they do not add additional complexity to the work. Section 4.6 shows what is needed to add invariants.

- Sub-messages as described in Section 2.3.3 are not supported in BDT. Further research is necessary to develop a theory for them first before

they are added to the tool.

- As mentioned above, only a subset of legal contracts is supported. As a result, quantifications and object equality (inequality) are not fully supported. Simple expressions involving reference equality work as expected. For example, in BDT a contract might state that $(x = Void \wedge x = y) \Rightarrow (y = Void)$. BDT submits this to *Simplify* as

  ```
  (IMPLIES (AND (EQ x |Void|) (EQ x y)) (EQ x y))
  ```

  Simplify will reduce this to *true* as expected. However, we would need further axioms to be added to the theory to fully support reference equality. For example, suppose we have `x,y:SAVING`, then we need to add an axiom that says $(x = y) \Rightarrow (x.balance = y.balance)$. These kinds of extensions are not that difficult to do, but they are not currently implemented in BDT.

- *Simplify* supports quantifications and as a result, they could be added to the tool. The work with contracts involving objects needs additional research.

- Currently, BDT only supports single-dot notations, e.g. *sav.balance*. Support for multi-dot notation can be added, but it would require a more robust parser.

- Genericity is not supported.

- Inheritance of contracts is not supported.

Most of the features from above can be easily implemented, but as mentioned, this is a prototype tool for the purpose of demonstrating our ideas only. The next stage in the development of BDT would involve implementing these omitted features and adding more error checking to BDT.

Chapter 6

Discussion

While multi-view consistency of software architectural languages (e.g. UML and BON) is discussed in the literature, no consistency tools have emerged until very recently. Yet, inconsistent views could have major ramifications for software quality and reliability.

Gao [Gao04] recently developed the first such multi-view consistency tool. The main focus of the tool was to check for static consistency. To check for contractual consistency, class features have to be implemented, and tests are run simulating the dynamic diagrams. In this thesis, we develop a tool (complementary to that of Gao) that can deal with multi-view contractual consistency directly, without the need to develop implementations. The main contributions of this work are as follows:

- Our BDT tool allows developers to describe and edit BON static diagrams with contracts that describe the class behaviour, as well as dynamic diagrams to describe object communication sequences.

- To our knowledge, BDT is the first tool to automatically check contractual consistency between message sequences in the dynamic diagrams against their contracts in the static diagrams, without the need to supply implementation detail.

- Consistency checking is done by symbolic execution of the messages using their contracts with the help of a theorem prover. Thus, BDT provides executability and thus the ability to test partial models early on in the design process. Feedback is provided as to the precise location of contract violations. This allows the designer to change the design accordingly.

- Current MDD tools are limited to the use of statecharts for describing behaviours. BDT allows the designer to work with contracts which describe behaviour at a higher abstraction level than statecharts.

- Current MDD tools support the informal use of dynamic diagrams, but these diagrams are not formally factored into the final generated model. But dynamic diagrams (message sequence diagrams and collaboration diagrams) are widely used in the development of business systems. BDT directly incorporates dynamic diagrams into the design process and in

fact ensures that they are consistent with the rest of the design. This does not mean that the use of statecharts is discouraged, rather that our approach is an advantage orthogonal to the use of statecharts.

- The use of *Simplify* as our theorem prover has made our tool fast, which returns proofs and counter examples quickly.

In summary, we have developed a theoretical approach for defining and checking contractual consistency of a multi-view model. In addition, we presented a tool that implements most of the ideas in this thesis. There will be three major areas for future work.

First, our BDT tool requires improvements and additions to make it an industrial-strength tool. Section 5.3.6 outlined the areas in which BDT needs improvements. The most important ones include support of invariants and quantifications in contracts. Quantifications would allow us to specify more advanced behaviour about routines such as properties of Lists and Arrays. In addition, inheritance and genericity need to be included in BDT. We believe that once these features are implemented, BDT will provide a unique modelling environment that does not currently exist.

Second, BDT can be incorporated with the Eiffel Development Tool (EDT) [Mak03]. EDT is an integrated development environment (IDE) for software production using Eiffel. EDT has also been developed as a plug-in for Eclipse and as a result can be integrated with BDT. The integration of both tools

would support forward and reverse engineering. Therefore, the user could develop models, check their consistency and generate partial program code.

Finally, BDT/EDT can be expanded to provide program verification similar to ESC/Java [LNS00]. The Eiffel Refinement Calculus (ERC) [PO04] could be used to ensure that program code satisfies the specifications of a class. Using this environment, the user can start by developing models of the system using contracts only and checking their consistency. Then, partial program code can be generated and routine bodies completed. Finally, the correctness of the program code can be checked against the specifications.

Appendix A

Dynamic Diagramming Module

BDT allows the user to construct dynamic diagrams in BON. Dynamic diagrams can be used to illustrate the runtime/dynamic behavior of an object-oriented system. Figure A.1 shows a screen shot of the dynamic diagramming module. The sections below describe the implementation of this module and highlight the most important design decisions.

A.1 Model

This section outlines the model behind a dynamic diagram that a user constructs. Figure A.2 displays the structure of classes that represent objects, object groups and the dynamic diagram within the model. The class *DynamicDiagramContainer* represents a container to which instances of *DynamicElement* can be added. An instance of *DynamicElement* represents an element

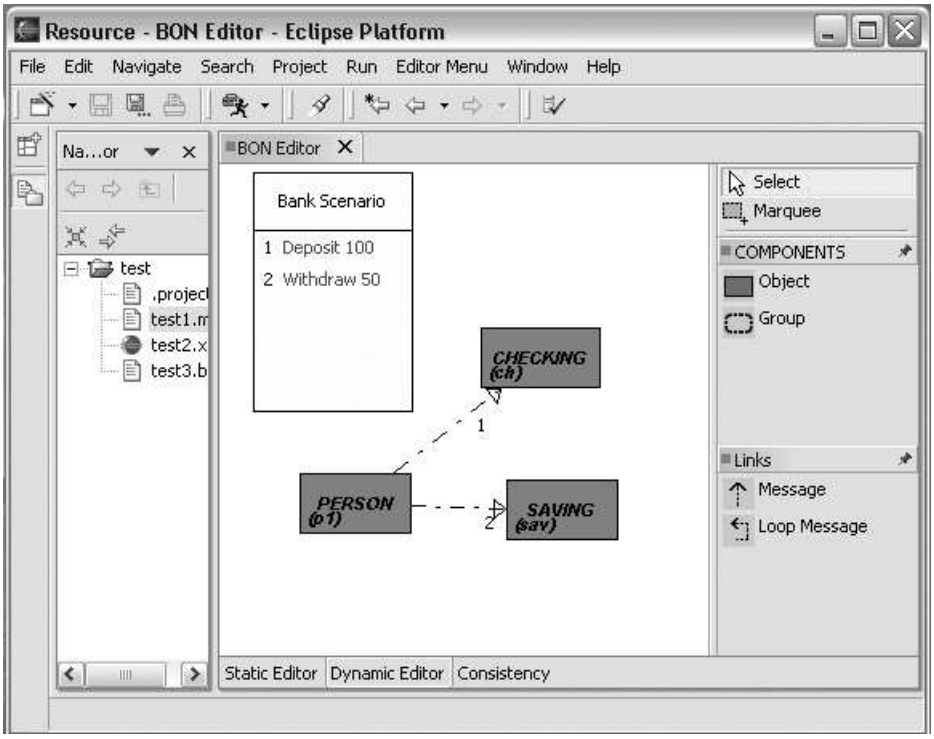

Figure A.1: A screen shot from the dynamic diagramming tool of BDT.

in the diagram and is implemented by *ObjectModel* and *ObjectGroupModel*. Class *DynamicDiagramModel* implements the class *DynamicContainer* and represents the entire dynamic diagram to which elements are added.

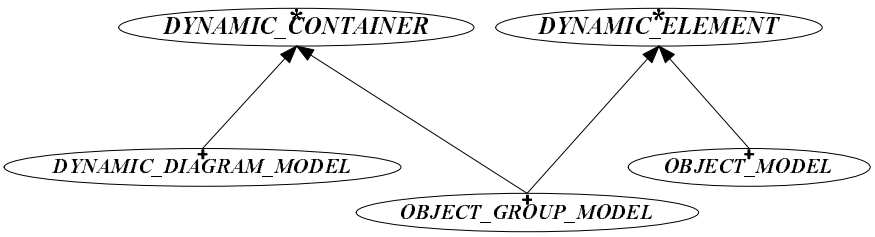

Figure A.2: The model for a dynamic diagram, an object and a group of objects. *DynamicContainer* represents the model of a container. *DynamicElement* represents an element such as an object or group of objects.

There are two classes representing messages within the model. First, the class *MessageModel* represents messages between two different objects, i.e., the source and target objects are different. Second, class *SelfMessageModel* is for those messages that are "loop-messages". These messages have the same source and target object. A self message has also a different graphical representation.

A.2 View

The view in the dynamic diagram is based on figures in the draw2D package of GEF. Objects are graphically represented by *ObjectFigure* and object-groups by *ObjectGroupFigure*. As Figure A.4 shows, both of these classes extend

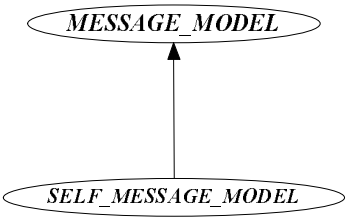

Figure A.3: *MessageModel* is the class representing the model of a message in the dynamic diagram. A *SelfMessageModel* is a special case of a *MessageModel* where source and target object are the same.

Label and use *ChopBoxAnchor* to represent the anchors to which message arrow-lines connect. Classes *ObjectFigure* and *ObjectGroupFigure* additionally use the classes *RectangleFigure* and *RoundedRectangleFigure* respectively for representing the outlines for their shapes.

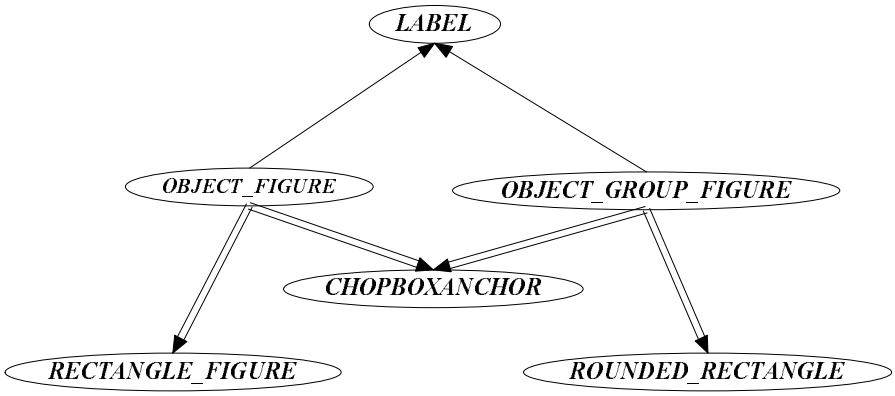

Figure A.4: *ObjectFigure* and *ObjectGroupFigure* are the figures in the view when representing an object and object group respectively. Both inherit from *Label* from draw2D and have an anchor.

The structure of classes representing messages graphically is similar to that of the model of messages. Class *MessageFigure* extends *PolyLineConnection*,

which represents a line in draw2D. Class *SelfMessageFigure* extends *MessageFigure* to implement a different representation of the message line. Both, *MessageFigure* and *SelfMessageFigure* use instances of *PolyLineDecoration* to get their shape and arrow head. Figure A.5 shows these concepts.

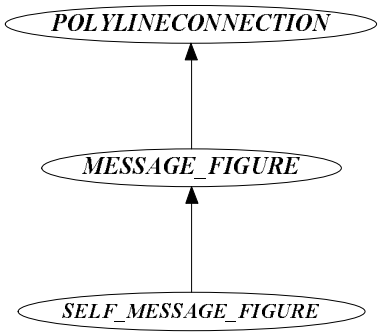

Figure A.5: *MessageFigure* and *SelfMessageFigure* are the figures that represent messages in a dynamic diagram. Both of these classes inherit from *PolyLineConnection*.

A.3 Control

The class structures representing controls in the dynamic diagram tool are very similar to those in the static diagram. Figure A.6 depicts the class structure of EditParts controlling the view and model of object-groups and objects. Class *ObjectGroupEditPart* inherits from *NodeEditPart* and *DynamicContainerEditPart*. The *EditPart* for containers has features to act as a container. *NodeEditPart* represents a node in the diagram to which lines (messages) can be connected.

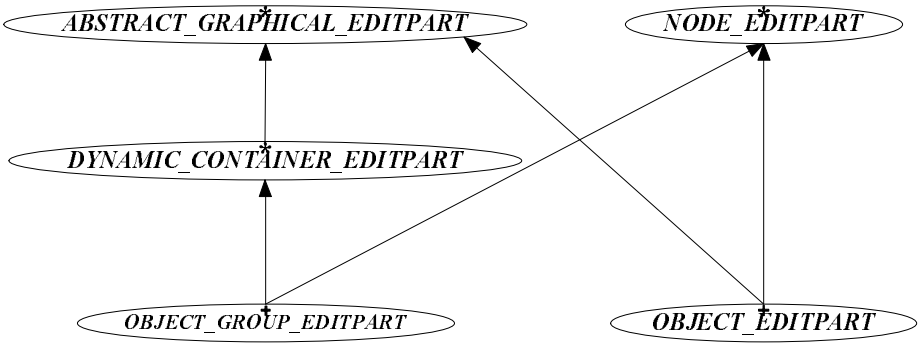

Figure A.6: *ObjectEditPart* is the control for object figures and model. *ObjectGroupEditPart* does the same for object group models and figures. Both inherit from *NodeEditPart* and *AbstractGraphicalEditPart*. *DynamicContainerEditPart* works similar to *BonContainerEditPart* for static diagrams. It represents a container.

Class *ObjectEditPart* inherits from both *NodeEditPart* and *AbstractGraphicalEditPart*. Figure A.7 shows that this class has references to *ObjectFigure* and *ObjectModel*, the view and model it keeps consistent. Figure A.8 shows the same for *ObjectGroupEditpart*. This class has references to *ObjectGroupFigure* and *ObjectGroupModel*.

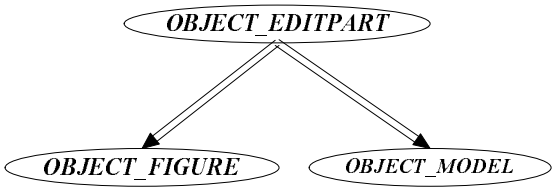

Figure A.7: The *EditPart* in this figure is responsible to keep view and model consistent. The view is represented by *ObjectFigure* and the model by *ObjectModel*.

The *EditPart* for the overall dynamic diagram has a similar structure as

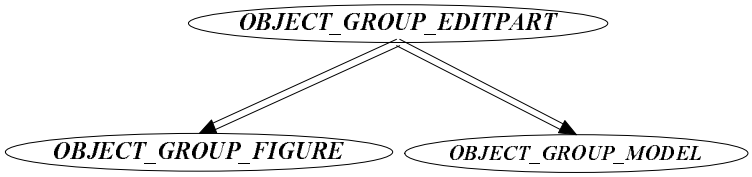

Figure A.8: *ObjectGroupEditPart* has a reference to the model, *ObjectGroupModel* and view, *ObjectGroupFigure*. It tries to keep both consistent.

its counterpart in the static diagram. *DynamicDiagramEditPart* extends *DynamicContainerEditPart* and has also two references to *FreeFormLayout* as its figure and *DynamicDiagramModel* as the model. *FreeFormLayout* is a figure from the draw2D package and acts as the canvas on which elements are drawn. This structure is shown in Figure A.9

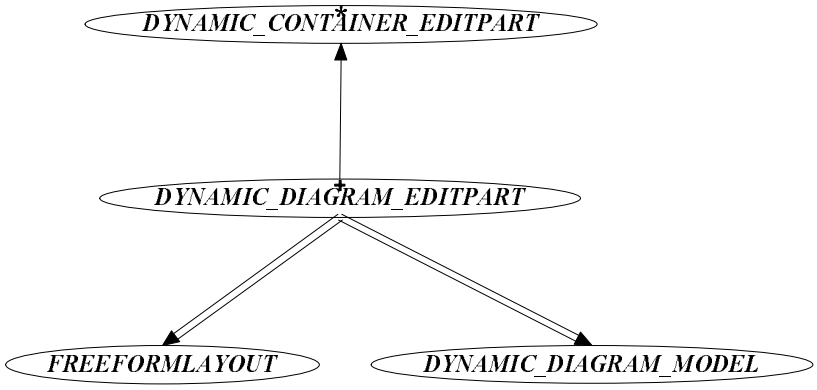

Figure A.9: *DynamicDiagramEditPart* extends *DynamicContainerEditPart* as it behaves like a container. It has references to the model and the view which it controls.

Even though there are two types of messages (regular and self-message) there is only one *EditPart* for controlling the view and model of messages. It has references to both *MessageModel* and *MessageFigure* to control the model

and view. The structure is shown in Figure A.10

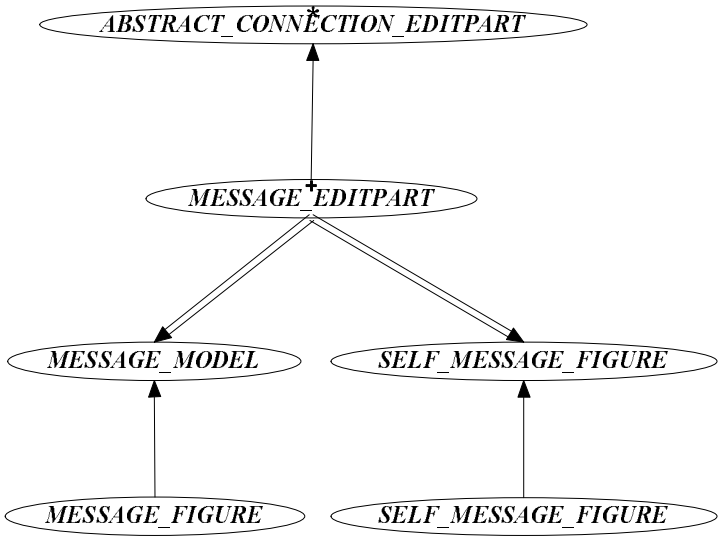

Figure A.10: *MessageEditPart* is the control for messages. It has a reference to *MessageModel* and *MessageFigure*. There is only one *EditPart* for both, messages and self-messages. The class deals with both cases.

A.4 BON Dynamic Editor

The editor for dynamic diagrams is set up similarly as that for static diagrams. *BonDynamicEditor* is responsible for creating a new instance of *DynamicDiagramModel* and passing it to *DynamicGraphicalFactory*. This factory is responsible for creating the appropriate *EditPart* of the model. In this case, the factory would create an instance of *DynamicDiagramModel*. Figure A.11 shows the class structure for classes involved in setting up and maintaining the editor. *BonDynamicEditor* implements routines for activating and deactivating

the editor, responding to actions and displaying graphical objects.

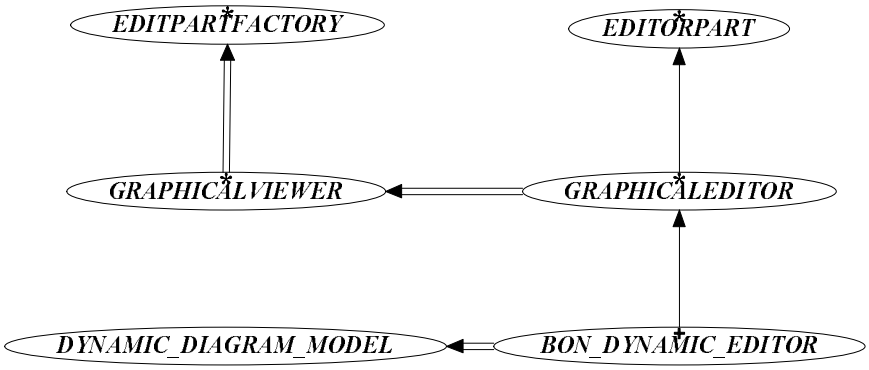

Figure A.11: *BonDynamicEditor* is the editor responsible for starting the dynamic diagramming application and has mechanisms for receiving events and processing actions. It has a reference to *DynamicDiagramModel* which is instantiated when the application starts.

Appendix B

Rational Rose RealTime

Rational Rose RealTime (RRRT) [Rat02] is a software development environment that is tailored to meet the special needs of real-time system development. It is, however, argued that it can be used for other software development domains as well. RRRT can be used to create models of the software system based on UML, generate implementation code and run the application. RRRT can be used through all phases of the software development lifecycle: from requirements analysis through design, implementation, testing and final deployment. All the development is performed through one single interface for model-based development.

Since Rational Rose RealTime is in support of the Model-Driven Architecture [MSUW03] framework it allows development at a higher level of abstraction. Developers specify behavior in statecharts and static structure in class

diagrams. In contrast to BridgePoint, the developer does not have to use the UML Action Language to specify behavior in statecharts. RRRT gives the designer the option to use from a number of programming languages, including Java and C++, to be used as the Action Language. This features provides the developer with the advantage of not having to learn a new language.

In this chapter, we will discuss the use of Rational Rose RealTime. The discussion will introduce the new construct *capsule* in UML, then give an example of code generation and finally discuss model execution.

B.1 Classes vs. Capsules

In order to achieve model executability and support for real-time systems, Rational Rose RealTime uses the extensibility features of the UML to define some new constructs. One of these new constructs is a *capsule*. A capsule is the fundamental element of Rational Rose RealTime. Capsules have much the same behavior as classes and have operations and attributes. They can also, as do classes, participate in dependency, generalization and association relationships. They have, however, some additional functionality and differ in some areas from classes. Here is a list of these features and differences:

- Capsules can communicate with other capsules only through the use of public ports, i.e. capsules do not have public operations, only public ports.

- A capsule's structure is defined by its attributes (like a class). These attributes, however, are completely private and no outside object can directly access them.

- Capsules can only communicate through messages which are sent and received through ports.

- When an operation is invoked in a class the action taken depends on the implementation of the operation. When a capsule receives a signal event, however, the behavior is controlled by its statechart. When a capsule receives a message from another capsule a signal event is generated and a response must be produced. The optional statechart associated with a capsule represents its behavior. The statechart is the only element that can access the protected attributes of the capsule.

As can be seen, capsules are very similar to classes, but have some additional features, the most important being statecharts. The user has the ability to develop systems without the use of capsules at all. This kind of development, however, would eliminate the ability to execute (partial) models. In order to achieve model executability with a model consisting of classes only, the full implementation is necessary. If model executability at a higher level and at earlier stages in the development is desired then capsules have to be used.

B.2 Code Generation

As discussed above, two of the main objectives of MDA and Rational Rose RealTime are to achieve model executability and automatic code generation for the developer. In order to achieve automatic code generation, capsule and statechart specifications are necessary. Program code is generated based on the semantics of capsules which is specified through a statechart. Statecharts describe the states that an object goes through in its life cycle and the transitions that can be taken from each state.

According to Rational Rose RealTime, round trip engineering is not necessary and forward engineering is enough for the developer to achieve the best performance and efficiency. As a result, the code generated by Rational Rose RealTime can be very complicated to inspect. Below is an example of a Hello World example. This example consists of one class and one method and is written in Java. The expected class and method look as follows:

```
Class HelloCapsule{
    public static void main (String[] args){
        System.out.println(``Hello World'');
    }
}
```

The above class is the code a programmer would produce if given the task to write a Java class that prints "Hello World" outside of RRRT. The same class can be developed using Rational Rose RealTime. In this case, to achieve full code generation a *capsule* must be defined and its statechart specified.

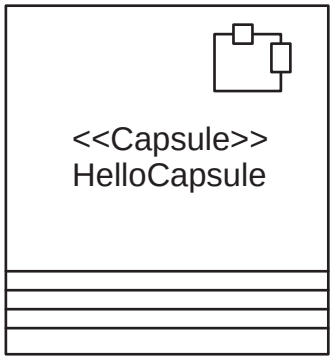

Figure B.1: HelloCapsule Capsule in Rational Rose RealTime.

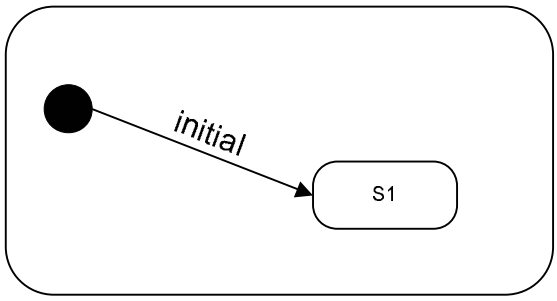

Figure B.2: State machine for HelloCapsule in Rational Rose RealTime.

The capsule and statechart are shown in Figure B.1 and B.2 respectively. The action belonging to the transition *initial* in this diagram is as follows:

```
java.lang.System.out.
    println(``Hello World from Capsule'');
```

After completing the model described above and performing code generation, RRRT produces a Java file with fourteen methods and 151 lines of code. Through quick inspection, the following easy-to-understand method can be located:

```
protected void rtTransition0_Initial() {
    java.lang.System.out.println
        (``Hello World from Capsule'');
}
```

This method represents the code for the initial transition. The following two methods are also among the fourteen added:

```
protected void rtPreparePorts() {
    super.rtPreparePorts();
    NewPort1 = new NewProtocol1.Base
    (   this, 1,
        (1 << com.rational.rosert.Capsule.
            PortDescriptor.IdShift )
        + com.rational.rosert.Capsule.
            PortDescriptor.NotificationDisabled
        + com.rational.rosert.Capsule.
            PortDescriptor.RegisterNotPermitted
        + com.rational.rosert.Capsule.
            PortDescriptor.Wired
        + com.rational.rosert.Capsule.
            PortDescriptor.VisibilityPublic
    );
}
```

```
protected com.rational.rosert.BindingEnd rtFollowInV (int
relay,int index){
    switch(relay){
    case 0:
        if(index < 1){
            return new com.rational.rosert.
                BindingEnd(this.NewPort1, index);
            break;
        }
        default:
            break;
        }
    return super.rtFollowInV(relay, index);
}
```

These two methods, however, are more difficult to understand and their purpose is not fully clear. The above example shows clearly that the code generated automatically can be very difficult to understand, even for very small programs. Whether this poses any clear difficulties for the programmer is debatable. MDA and RRRT argue that the programmer should only work at the model level and not at the code level. This approach, however, requires very strong debugging, execution and consistency capabilities at the model level. RRRT currently has debugging and execution capabilities, but lacks methods to check multi-view consistency. The next section, discusses this in more detail.

B.3 Model Executability

Model executability is one of the key objectives of MDA and as a result, that of Rational Rose RealTime. As with code generation, key to model executability

are the specification of capsules and statecharts. Statecharts are the only components in Rational Rose RealTime that take part in model executability.

The developer specifies the various capsules in the system and defines their statecharts and the communication between these statecharts. Once completed, the developer can build and subsequently "run" the model using model compilers. As with BridgePoint, the main disadvantage with this approach is the lack of activity diagrams such as collaboration diagrams in the process of model execution (and code generation). Many designers, especially those developing business systems, prefer the use of collaboration diagrams over statecharts. RRRT currently, allows the construction of such diagrams, but does not check their consistency with the capsules or incorporates them in model execution.

Bibliography

[AK03] Colin Atkinson and Thomas Kuhne. Model Driven Development: A Metamodeling Foundation. *IEEE Software*, 20:36–41, 2003.

[Ali04] Ali Hamie. Translating the Object Constraint Language into the Java Modelling Language. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1531 – 1535, Nicosia, Cyprus, 2004.

[Amb03] Scott Ambler. Agile Model Driven Development Is Good Enough. *IEEE Software*, 20:71–72, 2003.

[ARR00] Pascal Andre, Annya Romanczuk, and Jean-Claude Royer. Checking the Consistency of UML Class Diagrams Using Larch Prover. In *Draft Proceeding of Third Workshop on Rigorous Object Oriented Methods*, York, UK, 2000. ROOM'2000.

[BB01] Erwan Breton and Jean Bezivin. Towards an Understanding of Model Executability. In *FOIS*, pages 70–80, Maine, USA, 2001.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The UML Reference Guide*. Addison-Wesley, 1999.

[DNS03] David Detlefs, Greg Nelson, and James Saxe. Simplify: A Theorem Prover for Program Checking. Technical Report HPL-2003-148, HP Labs, 2003.

[Dou03] Bruce Douglass. *Real-Time UML*. Addison Wesley, third edition edition, 2003.

[Egy01] Alexander Egyed. Scalable Consistency Checking between Diagrams - the ViewIntegra Approach. In *Proceeding of the 16th IEEE International Conference on Automated Software Engineering(ASE)*, San Diego, USA, November 2001.

[EN95] Steve Easterbrook and Bashar Nuseibeh. Managing Inconsistencies in an Evolving Specification. In *Second IEEE International Symposium on Requirements Engineering*, 1995.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, May 2002.

[Gao04] Yan Gao. Multi-View Consitency Checking of BON Software Description Diagrams. Master's thesis, York University, July 2004.

[Gli00] Martin Glinz. A Lightweight Approach to Consistency of Scenarios and Class Models. In *Proceedings of the Fourth International Conference on Requirements Engineering*, Schaumburg, Illinois, 2000.

[Gro99] Object Management Group. Object Constraint Language Specification. OMG Unified Modelling Language Specification 1.3, 1999.

[GTLJ00] Erik Poll Clyde Ruby Gary T. Leavens, K. Rustan M. Leino and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, ftp://ftp.cs.iastate.edu/pub/techreports/TR00-15/, 2000. ACM.

[HHM01] Bogumila Hnatkowaska, Zbigniew Huzar, and Jan Magott. Consistency Checking in UML Models. In *Proceedings of the Conference "Information System Modelling"*, pages 35–40, Ostrava, 2001.

[IBM03] IBM. Graphical Editing Framework. http://www.eclipse.org/gef, 2003.

[IBM04] IBM. Eclipse. http://www.eclipse.org, 2004.

[KR03] Vinay Kulkarni and Sreedhar Reddy. Seperation of Concerns in Model-Driven Development. *IEEE Software*, 20:64–69, 2003.

[Kri00] Padmanabhan Krishnan. Consistency Checks for UML. In *Proceedings of the 7th Asia-Pacific Software Engineering Conference*, pages 162–169. IEEE, 2000.

[LNS00] Rustan Leino, Greg Nelson, and James Saxe. ESC/Java User's Manual. Technical Report Technical Note 2002-002, Compaq Systems Research Center, 2000.

[Mak03] David Makalsky. Eiffel Development Tool. http://sourceforge.net/projects/edt, 2003.

[MB02] Stephen Mellor and Marc Balcer. *Executable UML*. Addison-Wesley, 2002.

[MCF03] Stephen Mellor, Anthony Clark, and Takao Futagami. Model-Driven Development. *IEEE Software*, 20:14–18, 2003.

[McN03] Ashley McNeile. MDA: Vision with the Hole? http://www.metamaxim.com/download/documents/MDAv1.pdf, 2003.

[Mey92] Bertrand Meyer. *Eiffel - The Language*. Prentice-Hall, 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1997.

[MSUW03] Stephen Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. Model-Driven Architecture. http://www.omg.org/mda, 2003.

[OMG03] OMG. Unified Modelling Language Specification: Version 2.0. http://www.omg.org, 2003.

[OMG04] OMG. Object Management Group. http://www.omg.org, 2004.

[OSR01] S. Owre, N. Shankar, and J. Rushby. PVS System Guide 2.4. CSL, SRI International, 2001.

[PO99] Richard Paige and Jonathan Ostroff. A Comparison of Business Object Notation and the Unified Modeling Language. In Richard Paige and Jonathan Ostroff, editors, *International Conference on the Unified Modeling Language (UML'99)*, pages 67–82. Springer-Verlag, 1999.

[PO04] Richard Paige and Jonathan Ostroff. ERC: an Object-Oriented Refinement Calculus for Eiffel. *Formal Aspects of Computing*, 16:51–79, April 2004.

[POB02] Richard Paige, Jonathan Ostroff, and Phillip Brooke. Checking the Consistency of Collaboration and Class Diagrams using PVS.

In *Fourth Workshop on Rigorous Object-Oriented Methods*, London, England, 2002.

[Rat02] IBM Rational. Rational Rose RealTime. http://www.ibm.com/rational, 2002.

[SC02] Jean Louis Sourrouille and Guy Caplat. Checking UML Model Consistency. In *Fifth International Conference on the Unified Modeling Language*, 2002.

[Sel03] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20:19–25, 2003.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20:5–12, 2003.

[Tec04] Project Techonology. BridgePoint Development Suite. http://www.projtech.com, 2004.

[TO03] Ali Taleghani and Jonathan Ostroff. The BON Development Tool. In *Proc. Eclipse Technology eXchange eTX/OOPSLA*, Anaheim, CA, 2003.

[Uhl03] Axel Uhl. Model Driven Architecture is Ready for Prime Time. *IEEE Software*, 20:70–71, 2003.

[WN95] Kim Walden and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture*. Prentice-Hall, 1995.

[WS00] Jon Whittle and Johann Schumann. Generating Statechart Designs From Scenarios. In *International Conference on Software Engineering*, 2000.