MULTI-VIEW CONSISTENCY CHECKING OF BON SOFTWARE DESCRIPTION DIAGRAMS

by

Yan Gao

A Thesis Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfilment of the Requirements
For the Degree of

MASTER OF SCIENCE

In the Graduate College

YORK UNIVERSITY

2 0 0 4

Copyright Page (ii — not typed)

Certificate Page (iii — not typed)

Abstract

Multi-View Consistency is an important aspect of current Model Driven Development (MDD) methods for software construction. A model may consist of many views. We need some assurance that these views are consistent. Yet, none of the current MDD tools provide any justification that the generated code is consistent. The current generation of MDD tools also do not support code generation based on dynamic diagrams and contracts, so that consistency of static and dynamic diagrams is an unexplored territory.

In this thesis we describe a first (to our knowledge) prototype multi-view consistency checking tool. We first formalize the notion *consistency(v1,v2)* of two views *v1* (a static view with contracts) and *v2* (a dynamic view) of a model, based on prior work by Paige and Ostroff in [1;14]. The definition of consistency (which is more comprehensive than the earlier work) is divided into syntactic and contractual consistency. We then develop algorithms to check syntactic consistency, and incorporate these algorithms in a new tool called the BON Consistency Checking Tool (BCCT).

The tool can be used to construct graphical models, features, contracts and detailed body code and automatically run the syntactic consistency tests. The model can be automatically translated to executable Eiffel code, and a testdriver is used to check contractual consistency. The tool can be used to interactively and repeatedly construct models, automatically test for consistency and refactor the models as required. This leads to a design method that we call Consistency Driven Development (CDD).

Acknowledgements

I'd like to take this opportunity to thank everyone who made this project possible.

Firstly, I would like to express my deep gratitude to my supervisor, Dr. Jonathan Ostroff, who is so knowledgeable and generous. His continuous encouragement, motivation and patience have kept my research on the right track and made this thesis possible.

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

I particularly wish to thank my team members: Ali Taleghani, Dave Makalsky, and Oleksandr Fuks. I also thank my many classmates. My special thanks go to Ali Taleghani for his help on my thesis writing and Dave Makalsky for his good questions.

I would also like to thank many people in our department, including students, support staff and faculty, for always being helpful over the years. I thank my friends at York University for their kind help.

Thanks go to my family for their understanding, support and patience.

Table of Contents

List of Algorithm

List of Figures

Chapter 1 Introduction

Imagine that you are building your dream home. You hire an architect to draw up the blueprints (Figure 1-1). In the elevation view you specify a window facing west with a beautiful sea view. In the plan view the architect omits the window. These two views are now *inconsistent* with each other. If the builder uses the plan view to construct the floors, walls, windows and doors, then the window may actually be omitted in the actual construction. Fixing it at a later date may also prove much costlier than getting it right the first time.

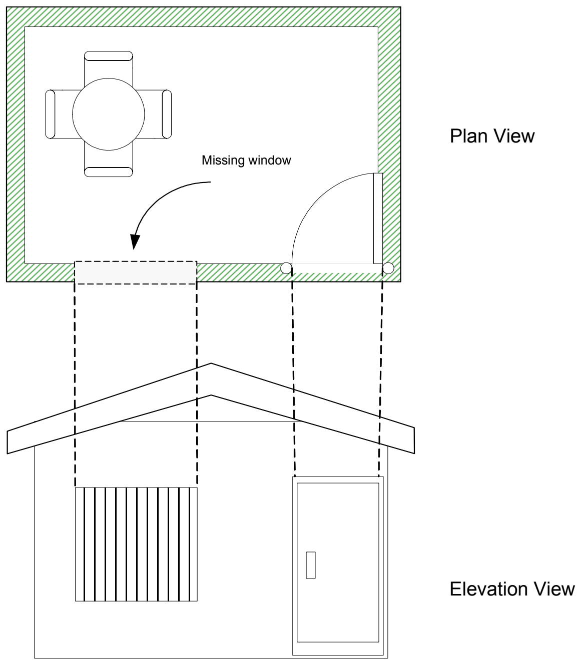


Figure 1-1 Plan and Elevation views of a house

Modern software products involve some of the most complex artefacts in existence. Software engineers have suggested that constructing *models* of software artefacts is as necessary as constructing models of buildings before they are built and aeroplanes before they are constructed and flown. Models help us understand and analyze complex problems and potential solutions through *abstraction*. By removing or hiding detail that is irrelevant in a given viewpoint, we may understand the essence of the problem more easily.

Models must also be understandable, accurate, predictive and inexpensive [37]. The last property ("inexpensive") is obvious – the model must be significantly cheaper to construct and analyze than the modelled system itself. But here is the rub. The cost of building a bridge is immense (steel girders, concrete etc.). A paper blueprint of the bridge is cheap. By contrast however, software is by definition "soft". There is no real cost difference between code and models of code – they are both just bits in a file on the hard disk. This has suggested to some software professionals to stress code over models and documentation [48].

But, many software developers do not want to give up on the many benefits that come from models. This is especially the case given that only one third of commercial software development products complete and are successful [49]. The remaining products either fail altogether, or are late and over budget. Modelling will clearly aid us in developing better quality code. But, if we must develop the artefact twice, once as a model and again as code that implements the model, then the process is not inexpensive.

A solution to this conundrum has been suggested – it is called Model Driven Development (MDD). MDD's defining characteristic is that software development's primary focus and products are *models* rather than *code* [37, 50]. A key idea is that executable code (the final software product) is automatically generated from their corresponding models. To obtain the full benefit of MDD we must have the following:

- Complete code must be automatically generated from models, as opposed to just skeletons and fragments;
- Changes are always made to the model, and not to the generated code, otherwise the code and model may become inconsistent;
- Models must at the very least be executable. David Harel compares models that cannot be executed to cars that do not have engines. There must be some way to test or check the behaviour of the model.

Thus *abstraction* and *automation* are the two key elements of MDD.

There are now emerging industrial standards to support MDD [37]. The Object Management Group is a consortium of software vendors, users, governments and academia that has recently announced its Model Driven Architecture (MDA) initiative that will support MDD. A key part of the standard is an enhancement to the Unified Modelling Language (UML). The MDA/MDD approach works as follows:

- The designer develops a platform independent model (PIM) in UML to represent the desired business functionality;
- An MDA-compliant tool applies standard mappings to generate a Platform Specific Model (PSM) e.g. based on J2EE, .NET or XML/SOAP;
- The MDA tool generates all (or most of) the implementation code from the PSM for deployment.

Not all MDD tools follow the strict MDA route. For example, the IBM Real-Time Rational Rose tool follows a two-stage process. In the first stage, a composite PIM/PSM model is constructed and tested, e.g. Java might be used to provide behavioural detail in a UML statechart. In the second stage, the model is automatically translated to code.

UML is a software description language that allows the designer to specify, visualize, and document models of software systems. However, there are problems associated with the use of UML.

The UML standard (v1.3) states: "Every complex system is best approached through a small set of nearly independent views of a model; no single view is sufficient. These diagrams provide multiple perspectives of the system under analysis or development. The underlying model integrates these perspectives so that a *self-consistent* system can be analyzed and built. These diagrams, along with supporting documentation, are the primary artefacts that a modeller sees, although UML and its support tools will provide for a number of derivative views."

As explained in [11], the phrase "self-consistent" in the above quote is problematic. UML allows for multiple views of the system. This alone is not problematic; indeed, experienced practitioners know that no single view of the system will suffice. What is problematic is the claim that these views will be consistent with each other. No such guarantee exists, and very little guidance has been provided with UML for how one would achieve such consistency.

Model inconsistency may arise for various reasons, e.g. due to misunderstandings of requirements, mistakes in constructing designs, and syntactical or semantic errors in writing the models themselves. It is desirable to be able to detect model inconsistency at an early stage, so that the problems will not be propagated to code or customer deliverables such as documentation [14].

1.1 Model Consistency

A UML *model* for a software artefact may involve many *views*, e.g. use case diagrams, class diagrams, statecharts, collaboration diagrams and deployment diagrams. The whole notion of view consistency does not have an accepted formal definition in the literature. Rather, different researchers have developed their own notions, and this makes the literature on the topic hard to summarize.

We may categorize the subject matter of consistency into *single-view* consistency, and *multi-view* consistency.

1.1.1 Single-view consistency

Single-view consistency describes the constraints that must exist in a single view to make that view legal. For example, not every UML class diagram is legal, e.g.

- If class A inherits from class B, then class B cannot inherit from class A;
- If class A has an attribute named *d*, then no other attribute may have the name *d*.

A modelling language such as UML therefore consists of two parts: a *notation* which is used to describe models, and a *metamodel* which expresses the well-formedness constraints that all legitimate models written in the notation must obey. Without a precise metamodel it is difficult to explain the notation and build tools that support the notation.

As an example, consider a brief overview of the UML metamodel as described in the UML 2.0 Infrastructure Specification [52]. The Specification states that a model typically contains model elements. These are created by instantiating model elements from a metamodel, i.e., metamodel elements. The typical role of a metamodel is to define the semantics for how model elements in a model get instantiated.

The UML Specification illustrates the notion of a metamodel with Figure 1-2. The metaclasses *Association* and *Class* are both defined as part of the UML metamodel. These are instantiated in a user model in such a way that classes *Person* and *Car* are both instances of the metaclass *Class*, and the association *Person.car* between the classes is an instance of the metaclass *Association*.

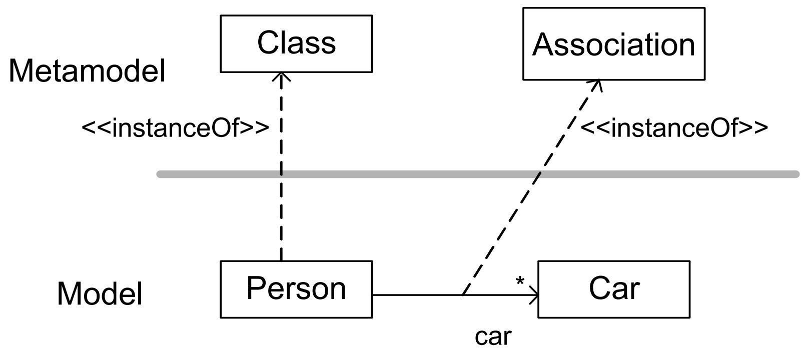


Figure 1-2 UML models and metamodels

We have therefore used UML to provide a metamodel of itself. While this approach lacks some of the rigour of a formal specification method, it offers the advantage of being more intuitive and pragmatic for tool implementers and practioners. Constraints can now be written on the metamodel to specify legitimate elements.

Some researchers [18,19,20] have used the UML logic called OCL to write metamodel constraints. These constraints are applied to single-view consistency. Various calculi have also been developed for providing metamodels with an appropriate semantics [51].

1.1.2 Multi-view consistency

Multi-view consistency for UML has been treated in [10,17,18,21,24,25,33]. In multi-view consistency, we would like to know if two totally different views are "consistent" with each other. For example, we would like to know if a UML class diagram is consistent with a collaboration diagram.

There is no standard definition of “consistency” and each research group adopts its own different approach, usually informal. To the best of our knowledge, even where more formal approaches are suggested, no tool has yet been developed that implements the approach.

As an example, consider the lightweight approach to multi-view consistency developed in [17] which presents an approach to determine the consistency between a class model and a scenario model. The work assumes semi-formal, loosely coupled models that are complementary. Scenarios model the external system behaviour and class models specify the internal state dependent functionality. Consistency is achieved by:

- minimizing overlap between the two models; and
- systematically cross-referencing corresponding information.

An example of a “formally checkable” rule in [17] is the *Conformance of References Rule*: “every reference from a scenario to an item in the class model must have a corresponding reference in the class model”. While [17] calls this rule formally checkable, a lot of infrastructure, including the proper definition and elaboration of the underlying metamodel, would be needed.

The work in [53] is interesting because it proposes a formal notion of consistency between a basic state-machine model and a message sequence chart by defining the notion of a system trace. Views can then be transformed to labeled transition systems, and consistency checked by the intersection of the transition systems. It is not clear how this approach would scale up to complete and fully specified UML diagrams.

As mentioned earlier, MDA/MDD are now making quite an impact in industry. There are currently over 40 MDA style tools listed at the OMG website[1]. Surprisingly, none of these tools really support multi-view consistency in the sense that we have been discussing it.

In order to see this, we must understand how MDD is currently being pursued. As pointed out in [55], there are currently two approaches. Both approaches allow the full range of UML diagrams (or views) in their tools including structural class diagrams and behavioural collaboration diagrams. However, both approaches do not incorporate the collaboration diagram into model consistency (and hence model execution) and the final generated code. Thus neither approach truly address multi-view consistency.

In the *translationist* approach [50] the PIM is automatically translated directly into the final code of the system using *generation rules*. The downstream artefacts (PSM and code) are not further elaborated or amended by hand. The PIM is the full source of the generated system.

The generation rules are used to convert the class diagram into the final code. How is the complete behaviour of a class defined? In the translationist approach the

[1] *http://www.omg.org/mda/committed-products.htm*

"behaviour of the system is driven by objects moving from one stage in their lifecycle to another in response to events" [50, page 6]. In this approach, UML statecharts and a device independent Action Language are used to elaborate the behaviour of a class in the PIM.

In the *elaborationist* approach, the definition of the application is built up gradually as you progress from PIM, to PSM and finally to code. Thus, it is possible for the lower level models to get out of step with the higher ones. Thus, "round trip engineering" support is provided to help the designer get the code in synch with the model. Most of the current MDA tools appear to fall into this category.

A recent text [54] on the elaborationist approach also uses class diagrams for structure. However, instead of using statecharts for behaviour, [54] recommends that the "dynamics of the system are represented by pre and post conditions on operations" [54, page 36]. OCL is the recommended language for expressing these contracts. A major advantage of this approach is that later code can be tested against the contracts, a feature missing from the translationist approach. To our knowledge, none of the current MDA tools support OCL in this way. So this approach is still at the conceptual stage.

What do translationists say about elaborationist? Don't' use it "because elaboration is stupid" [50, page 303] as the benefits of MDD automation are lost. Elaborationists write that "Executable UML [= translationists] is suitable within specialized domains [e.g. real-time systems], but even there the benefits are less than you would expect" [54, Page 36]. In commercial applications statecharts are not the preferred modelling view; rather, collaboration diagrams are preferred [55].

In both approaches, basic system structure is defined in a class diagram. In order to define behaviour and details of the class methods, either statecharts [50], or pre/post conditions [54] are used. Roughly speaking, statecharts and contracts provide complementary elaborations of class behaviour. BON class diagrams (see sequel) contain the ability to display the classes and their contracts in a single-view. Thus both current translationist and elaborationist approaches are, in effect, dealing with what is a single view in BON. What is missing? – Collaboration diagrams for showing the run-time interactions and scenarios between objects (called dynamic diagrams in BON). The challenge taken up in this thesis is to develop a tool for checking the consistency of structural class diagrams and behavioural dynamic diagrams.

1.2 Other consistency approaches

The work in [21] provides a framework for managing multi-view consistency called *xlinkit* that is a generic tool for managing the consistency of distributed documents. It consists of a language based on first order logic for expressing constraints between documents, a document management system and an engine that checks documents against constraints.

In [42, 43, 38] a framework is developed in which software development knowledge is portioned into multiple views called "ViewPoints". Inconsistencies between

ViewPoints are managed by explicitly representing relationships between them, and recording both resolved and unresolved inconsistencies. In this approach inconsistencies are not always considered “bad” as it might prove premature to remove them too early in the design. Inconsistency management can in fact be used as a tool for requirements elicitation.

1.3 The contribution of this thesis

In this thesis, we develop the first tool, to our knowledge that does multi-view consistency checking between structural class diagrams and behavioural collaboration diagrams. For reasons explained in the next chapter, we use the description language BON rather than UML.

We formalize the notion *consistency(v1,v2)* of two views *v1* (a static view with contracts) and *v2* (a dynamic view) of a model, based on prior work by Paige and Ostroff in [1;14]. The definition of consistency (which is more comprehensive than the earlier work) is divided into syntactic and contractual (or behavioural) consistency. We develop algorithms to check syntactic consistency, and incorporate these algorithms in a new tool called the BON Consistency Checking Tool (BCCT).

The tool can be used to construct graphical models, features, contracts and detailed body code and automatically run the syntactic consistency tests. The model can be automatically translated to executable Eiffel code, and a testdriver is used to check contractual consistency. The tool can be used to interactively and repeatedly construct models, automatically test for consistency and refactor the models as required. This leads to a design method that we call Consistency Driven Development (CDD), by analogy with Test Driven Development (TDD).

1.4 Thesis outline

The rest of this thesis proceeds as follows:

- In Chapter 2, we describe BON and explain why we choose BON over UML. We describe BON static and dynamic diagrams.
- In Chapter 3 we describe prior work on the BON metamodel. The metamodel is important in the definition of consistency between static and dynamic diagrams.
- In Chapter 4 we describe extensions to the metamodel needed for our tool, and we also define the notion of multi-view consistency using the metamodel. We also provide some of the algorithms involved in doing the consistency checks.
- In Chapter 5 we describe the remaining algorithm needed for consistency checking called the Specified Depth algorithm, which is needed for checking that a message in a dynamic diagram has an associated link in a static diagram.
- In Chapter 6 we describe our BCCT tool that implements the algorithms to check consistency. We also define the notion of Consistency Driven Development.

- Chapter 7 is our concluding chapter.

Chapter 2 Business Object Notation (BON)

In this chapter, we describe the graphical BON design language, including its facility for Design by Contract (DbC), and we also justify the choice of BON/Eiffel instead of UML/Java for our work on consistency checking.

2.1 Overview of BON

BON (Business Object Notation), developed by Jean-Marc Nerson and Kim Waldén [8], is an object-oriented method possessing a recommended development process as well as graphical and textual notations for specifying OO systems[2].

BON builds on three principles, fundamental to the construction of industrial strength quality software: seamlessness, reversibility and software contracts. It was developed as a means of extending the higher-level concepts of the Eiffel programming language into the realm of analysis and design, aided by a graphical and textual notation, and can be integrated into Eiffel seamlessly [30]. In this section, we introduce some BON notation and definitions. The references [8, 9] provide more detailed information on BON/Eiffel.

[2] For the BON method and tool see http://www.bon-method.com

Static and dynamic diagrams

Graphical BON supports two kinds of diagrams: static diagrams, similar to class diagrams in UML and dynamic diagrams similar to UML collaboration diagrams. A sample static diagram and dynamic diagram representing part of a banking system is shown in Figure 2-1 and Figure 2-2 respectively.

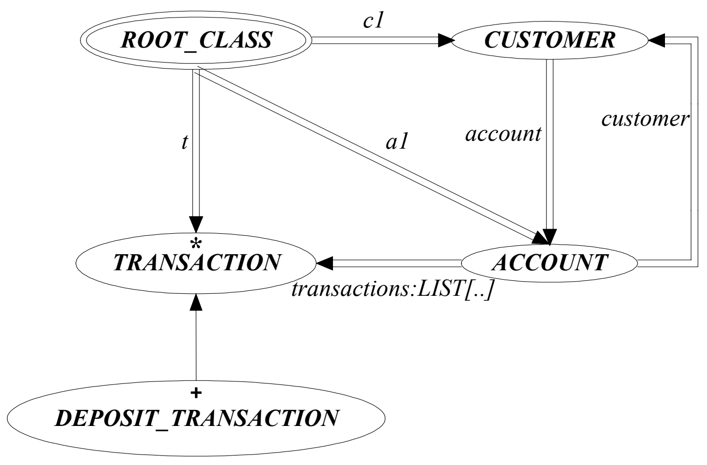
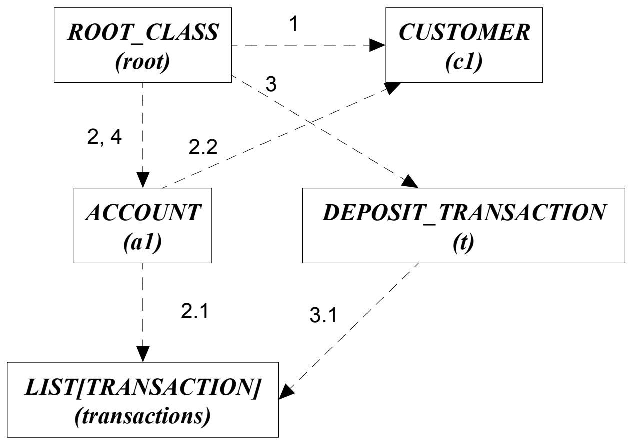


Figure 2-1 BON Static diagram of bank example

Static diagrams describe the structure of a system, i.e. the components and the relationships between these components. In BON static diagrams, classes are grouped into clusters. Likewise, in dynamic diagrams objects may be grouped [8]. Consistency checking does not depend on clusters and groups; hence we do not show clusters and groups in this thesis.



Scenario: deposit

1	create customer *c1*
2	create account *a1* with *c1*
2.1	create transaction list *transactions*
2.2	set customer's account to *a1*
3	create a deposit transaction *t* for account *a1* of $100
3.1	add transaction *t* to *transactions*
4	check balance

Figure 2-2 BON dynamic diagrams of bank example – Deposit

Dynamic diagrams document how the system will behave over time. A dynamic diagram consists of a set of communicating objects. The diagram will therefore contain one or more objects, messages sent between these objects and a *scenario box* to describe messages in free text.

A full execution of the system is simply the invocation of one routine (a constructor such as *make*) on the root object. The make routine in turn calls other routines and so on until termination. A system scenario is just a possible partial system execution.

Classes

The main construct in a BON diagram is a class. In a BON static diagram, there are one or more classes, which have a name, an optional class invariant, and zero or more features (attributes, functional routines and procedural routines). A class may be in one of two relationships with other classes: the inheritance relationship and the client-supplier relationship.

In BON, a class has two views: an *expanded* form and a *compressed* form. In the compressed form, a class header is represented graphically by an ellipse with the class name in the centre of the ellipse using upper case letters like this:

. The class name is an alphanumeric string with possible underscores. In Figure 2-1, there are five classes: *ROOT_CLASS, CUSTOMER, TRANSACTION, ACCOUNT,* and *DEPOSIT_TRANSACTION*. Graphical BON uses different class headers to represent different kind of classes: *root*, *deferred*, *effective*, etc. *ROOT_CLASS* in Figure 2-1 is a root class and it is shown with a double ellipse as:

. A root class is a class of which one instance will be created when an object-oriented process is started, and whose initialization routine (often called *make*) drives the execution.

An expanded form of the class with features and their contracts is shown in Figure 2-3.The expanded form of a class shows more detailed information of a class than the condensed form. It shows the class invariant, and a precondition and postcondition for each feature (e.g. *set_account*).

Class TRANSACTION* is deferred as indicated by the asterisk (*) after the name. This is because this class has at least one deferred routine *make** (i.e. a routine lacking implementation). DEPOSIT_TRANSACTION+ is an effective class as all its routines are implemented (as indicated by the plus sign after the class name).

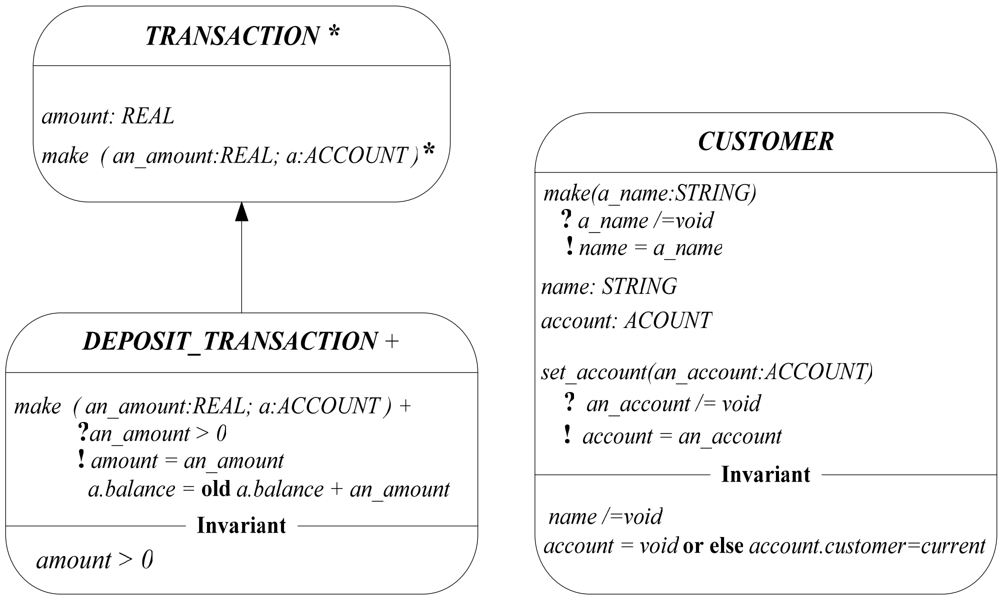


Figure 2-3 Expanded form of CUSTOMER

Features

In BON, a feature is either a *query* or a *command* (a procedural routine). A query returns a value but does not change the system state. A command does not return a value, but may change the state of the system. A query is either an attribute or a function routine.

A static diagram supports two kinds of relationships: *inheritance* and *client-supplier.*

Inheritance Relationship

Inheritance defines a sub-type (*is-a*) relation. It can be defined as the inclusion in a class, called *CHILD*, of operations and contract elements defined in another class *PARENT*. A class that is either a parent or grandparent (recursively) of a class is called an *ancestor* of the class and a class that is either child or grandchild (recursively) of a class is called a *descendant* of the class. An instance of a child class can always be used instead of an instance of the parent class.

A single arrow pointing from the child to its parent, called an *inheritance link*, represents an inheritance relation. In Figure 2-1, class *DEPOSIT_TRANSACTION* inherits from class *TRANSACTION*. Class *DEPOSIT_TRANSACTION* is the child and class *TRANSACTION* is the parent.

Client-supplier Relationship

A *client-supplier* relationship or *client relation* for short, between a *client* class *A* and a *supplier* class *B* means *A* uses services supplied by *B*. In Figure 2-1, there is a client-supplier relationship between class *TRANSACTION* and class *ACOUNT*. *ACCOUNT* is the client and *TRANSANCTION* is the supplier. In a static diagram, a double line extending from the client to the supplier – called a *client link*, represents a client-supplier relationship. There are two possible client-supplier relationships: *association* (represented by a simple arrow) and *aggregation* (represented by an arrowhead with a perpendicular line). Both kinds of association relationships provide the same information with respect to consistency. As a result, we will not treat them separately.

A client link can be labelled with one or several names, which represent names of references to the supplier classes. For example, there is a client link in Figure 2-1 between *ROOT_CLASS* and *CUSTOMER* labelled *c1*, which means that in *ROOT_CLASS* there is an attribute with name *c1* of type *CUSTOMER*.

Objects

Objects are represented graphically in dynamic diagrams by rectangles to differentiate them from ellipses (for classes) in static diagrams. The name of the corresponding class is in upper case in the centre and the name of the object in lower case.

Messages

A message that is sent from one object to another is represented by a dashed arrow extending from the *source object* to the *target object* – this arrow is called a *message link.* The source and target objects should have corresponding classes in the static diagram. The message link is labelled with sequence numbers which represent *time* in the scenario and correspond to entries in the *scenario box* where the role of each call is described using free text. Messages in dynamic diagrams are visual representations of feature calls.

BON Textual Notation

BON also supports a textual notation, which does not contain any description of spatial layout. The following is the specification of the class *CUSTOMER using* textual BON.

```
indexing
 description: "Information about bank customer"
class interface
 CUSTOMER
create
 make
feature
 account: ACCOUNT
 make (a_name: STRING)
   require
     a_name_not_void: a_name /= void
   ensure
     name_assigned: name = a_name
 name: STRING
 set_account (a_account: ACCOUNT)
   ensure
     account_assigned: account = a_account
end -- class CUSTOMER
```

Figure 2-4 BON textual view of class CUSTOMER

The above view has no implementation detail. It is, in fact, the same as the Eiffel contract view, which the EiffelStudio IDE (of Eiffel Software) can automatically extract from the complete implemented text of the class. This is what makes the high-level Eiffel views interchangeable with BON.

Both BON and UML can be used to describe a design irrespective of the implementation language (C, Java, Eiffel etc.). However, UML is not well-integrated with any language. As an example, UML supports multiple inheritance, but Java does not. It is not in general possible to seamlessly move from a UML design to Java code, nor is it possible to automatically reverse engineer the UML diagram from the Java code. By contrast, BON and Eiffel are seamless and reversible at least for static diagrams.

The BCCT tool (to be presented in this thesis), will aid in maintaining consistency between static and dynamic diagrams, a property not supported in any current tool. Obviously, BON/Eiffel is a better framework for such a tool than UML/Java, given that at least static diagrams are consistent with Eiffel code by construction.

2.2 Design by contract

The aim of software engineering is to build reliable software. Design by Contract (DbC) can be seen as an advanced software engineering technique for building quality software [9]. DbC is a principle that states interfaces of modules of a software system (especially mission-critical ones) should be governed by precise specifications, similar to contracts between people or companies. The contracts cover mutual obligations, benefits and consistency constraints (*invariants*). Together these properties are known as *assertions*, and are directly supported in BON and Eiffel [9].

In BON static diagrams, $\boxed{?}$ and $\boxed{!}$indicate pre and postconditions respectively. Invariants are specified in special invariant sections as shown in Figure 2-3.

In Eiffel, the precondition is the *require* part; the postcondition is the *ensure* part and the invariant is the *invariant* part (Figure 2-4).

Java does not support contracts directly, although several tools are available for monitoring behavioural contracts in Java. This is another reason why we choose BON/Eiffel instead of UML/Java for the investigation of consistency.

2.3 Single model principle

UML, a *de facto* standard for modelling languages, is a major step towards standardizing notations for the visual specification and design of object-oriented systems [13]. UML allows the construction of many *views* of the system under description. In terms of the views of a model, UML defines the following graphical diagrams:

- use case diagram
- class diagram
- behaviour diagrams:
  - statechart diagram
  - activity diagram
  - interaction diagrams:
    - sequence diagram
    - collaboration diagram
- implementation diagrams:
  - component diagram
  - deployment diagram

UML allows for multiple views of the system because no single view of the system will suffice. The problem is how to keep consistency between these various views. This is where the single model principle plays an important role.

The *single model principle* is defined in [11] as follows: “A software development follows the single model principle if it requires the use of a seamless and reversible wide-spectrum language for software description, possessing conceptual integrity at both the module and system levels, while maintaining view consistency at different level of abstraction.” Following this principle can make consistency checking simpler.

In [11], UML/Java and BON/Eiffel are compared with respect to the single model principle as shown in Figure 2-5. It is the lack of methods to ensure the consistency of static and dynamic diagrams that prevents BON/Eiffel models from completely satisfying the single model principle (see cell in the table of Figure 2-5with a *Qualified Yes*). Better consistency checking methods between static and dynamic diagrams will be treated in this thesis based on prior work in [1;14].

Criterion	UML/Java	BON/Eiffel
Seamless and reversible wide-spectrum descrip-tions	*No* (e.g., impedance mismatch between OCL and *iCon-tract*, or between statecharts and classes)	*Yes* (by construction)
Conceptual integrity	*No* (e.g. constraints can be expressed on dependency arrows, in notes, via OCL and in statecharts; collabo-ration and sequence diagrams are identical se-mantically)	*Yes* (by construction)
View consistency	*No* (in general, no algorithms or methods available to check the constructive part – classes and statecharts – against the other views, e.g., OCL)	*Qualified Yes* (Static diagrams and code can each be automatically derived from each other. Consistency of static and dynamic diagrams is treated in [1;14] and in this thesis).

Figure 2-5 Single model principle - comparison

Figure 2-6 describes the relationship between the various BON/Eiffel deliverables before the work reported in [1;14] and in this thesis. As described above, fully im-plemented Eiffel source code and BON static diagrams can be automatically derived from each other as indicated by the <<*auto-derive*>> stereotype. However, there is no guarantee of consistency between the dynamic and static diagrams as indicated by the question mark in the stereotype (<<?>>). What we would really like to achieve is the situation described in Figure 2-7.

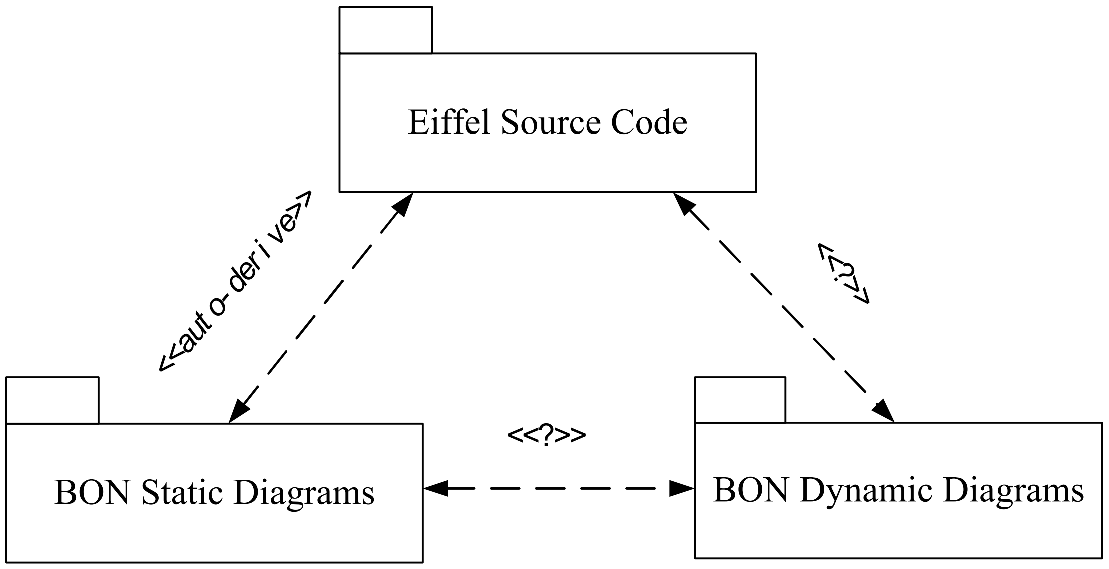


Figure 2-6 Eiffel deliverable dependencies before this thesis

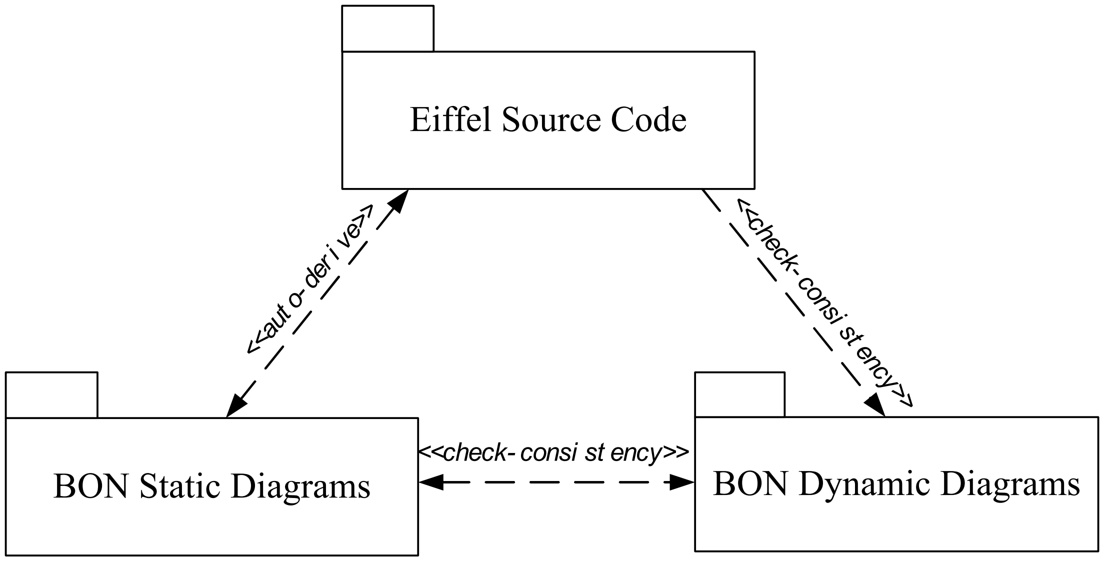


Figure 2-7 Eiffel deliverable dependencies in this thesis

It may not be possible to have an <<*auto-derive*>> relationship for static and dynamic diagrams (static class diagrams can be automatically computed from code and vice versa). However, at the very least, we can aim for an automatic consistency check, as described by the <<*check-consistency*>> stereotype in Figure 2-7, i.e. *given* a pair of diagrams, static and dynamic, press a button in some tool and automatically confirm (yes or no) if the two are consistent. Also, given a dynamic diagram and Eiffel code, we would also at the same time like to confirm that they are consistent with each other. Given any two views *v1* and *v2* (e.g. code and a diagram, or a static and dynamic diagram) we thus have two possibilities:

- <<*auto-derive*>>: *v1* can automatically be derived from *v2* and vice versa;
- <<*check-consistency*>>: given both *v1* and *v2*, we can automatically check that the views are consistent.

A relatively easy way to implement *check-consistency* would be to allow the software developer to develop the static and dynamic diagrams together (hand-in-hand). We could then ensure that the two views are consistent by construction. However, this would remove flexibility for the developer. Many developers might prefer to develop the diagrams independently. We thus define *check-consistency* by stating that two views (perhaps developed independently) are provided and we must automatically check the consistency of these two views. This is a more difficult problem than the hand-in-hand approach.

2.4 Why BON over UML?

From the discussion in the previous sections, BON has the following advantages over UML:

1. *It is simple.* Instead of a variety of behavioural diagrams (collaboration diagrams, sequence diagrams and statecharts) with impedance mismatch between them, BON uses the matching notions of Dynamic Diagrams and Contracts.
2. *It uses a rich assertion language for pre-conditions, post-conditions and invariants.* This allows the user to express constraints on class properties in the text of the program itself, whereas in UML, the OCL is written separately from the Class Diagram or code text. Without rich contract support (including run-time assertion checking) in the program text, consistency checking of OCL with the code is difficult.
3. *It more closely follows the Single Model Principle* [11], which makes it more amenable to consistency checking.
4. *It can be seamlessly integrated with Eiffel* [9].

Because of these advantages, we have decided to use BON/Eiffel for presenting our work on consistency checking. A detailed comparison between BON and UML can be found in [12].

Chapter 3 The BON Metamodel

A modelling language consists of two parts: a *notation* used to write models; and a *metamodel* which expresses the well-formedness constraints that all legitimate models written in the notation must obey. Without a precise metamodel it is difficult to explain the notation and build tools that support the notation.

The construction of the BCCT tool developed in this thesis will use a modified version of the BON metamodel. In this chapter we discuss the BON metamodel as it currently stands, and describe the modifications in the next chapter.

The BON metamodel has been specified precisely using BON itself (informally) and also formally in the higher order logic PVS [31]. The details of the BON metamodel are described in [3] and [44].

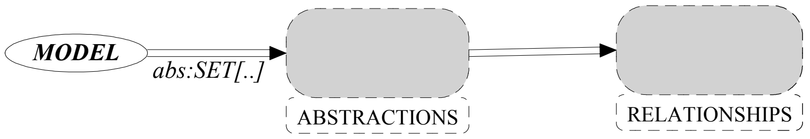


Figure 3-1 The BON metamodel, abstract architecture

A high-level view of the BON metamodel is shown in Figure 3-1. BON models are instances of the class MODEL. Each model has a set of *abstractions*. The abstractions cluster in Figure 3-1 describes either static views (BON static diagrams) or dynamic views (dynamic diagrams). For example, *CLASS* is an example of a static abstraction. A class will have properties such as a name, features, and parents. A class will also have a set of *relationships* with other classes, e.g. a class may inherit from another class. These relationships are described in the relationship cluster, and INHERITANCE is an example of a static relationship. The BON textual view of INHERITANCE is shown below:

```
 class interface INHERITANCE feature
      source: ABSTRACTION
      target: ABSTRACTION

      set_source (s: STATIC_ABSTRACTION) is
              require
                      s /= void
              ensure
                      source = s
              end

      set_target (s: STATIC_ABSTRACTION) is
              require
                      s /= void
              ensure
                      target = s
              end

invariant
      source /= target

end  -- class INHERITANCE
```

An example of a metamodel *well-formedness constraint* is that there should be no cycles in the inheritance graph (an ancestor cannot inherit from their descendant). In the BON metamodel such constraints are expressed via assertions (in this case an invariant). This constraint is formulated as an invariant *no_inheritance_cycles* in MODEL as shown in Figure 3-2.

```
class  MODEL feature

        rel: LIST[RELATIONSHIP]  -- list of all relationships including inheritance relationships

        closure: LIST [INHERITANCE]
                -- list of all direct inheritances as well as inheritances due to transitivity

        no_inheritance_cycles ≅
        $\forall r \in closure \bullet \neg\exists r_1 \in rel \mid r_1 : INHERITANCE \bullet (r.source = r_1.target \land r.target = r_1.source)$

        ...

invariant
        disjoint_clusters
        no_inheritance_cycles
        unique_abstraction_names
        no_bidirectional_aggregations
        unique_root_class
        at_least_one_instance_of_root
        model_covariance
        enable_dynamic_diagram
        ...
end
```

Figure 3-2 Class MODEL with an invariant *no_inheritance_cycles*

There are two categories of metamodel well-formedness constraints. The *no-inheritance-cycles* constraint applies to a single diagram (or view), in this case a BON static diagram. The first category of constraints is of this kind, and we call it *single view consistency*. The original metamodel was limited to this kind of consistency, and a BON case tool was developed to automatically perform these kinds of checks in [16].

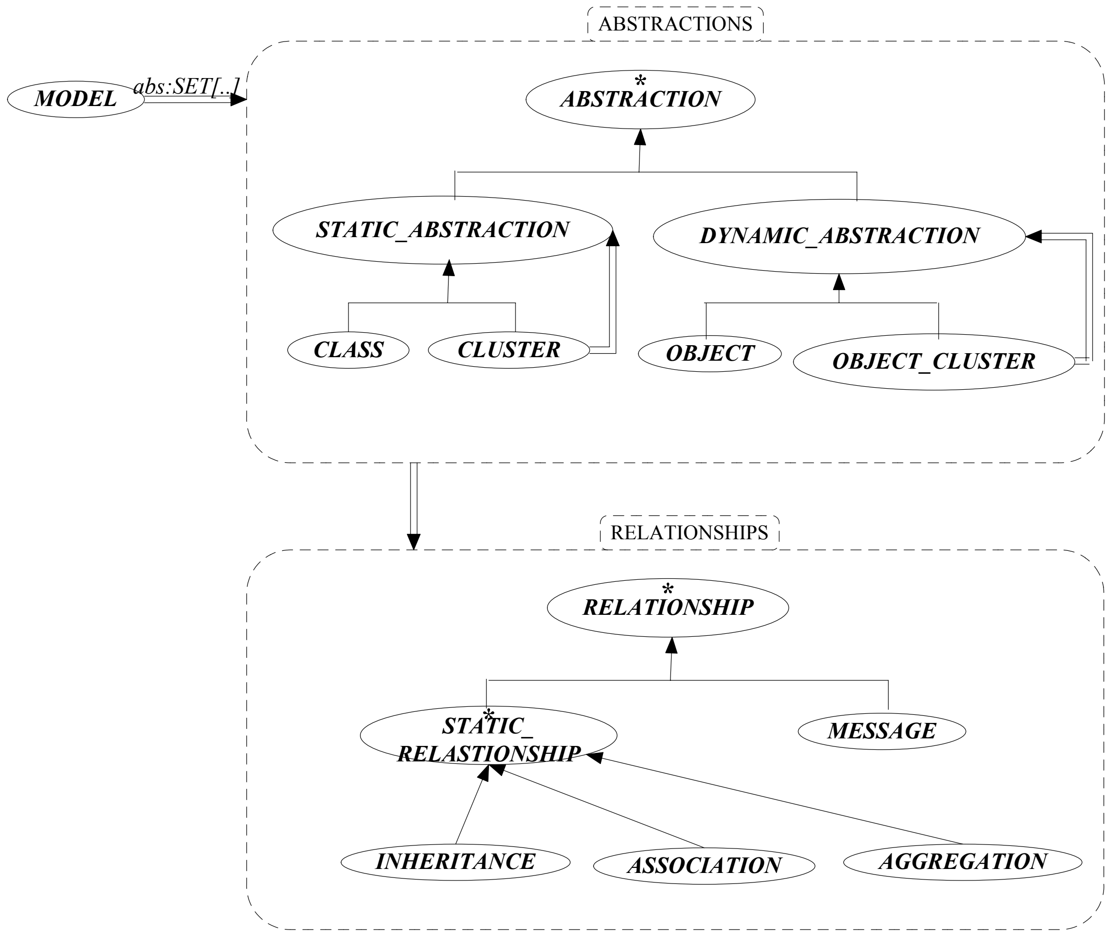


Figure 3-3 Details of the BON metamodel

In [1;14], a second category of well-formedness constraints was developed which we shall call *multiple-view consistency*, in which we are presented with two views of the system (a BON static and a dynamic diagram). We would like to confirm that these two views are consistent with each other. This is more challenging than single-view consistency[3]. However, to the best of our knowledge, no such tool currently exists for mechanized (or fully automatic) checking of view consistency.

The work in [1;14] introduces four constraints for consistency checking between static diagrams and dynamic diagrams using an extended version of the BON meta-model of [3], and describes procedures for consistency checking using the logic PVS. We discuss these consistency constraints using the bank example of the previous chapter.

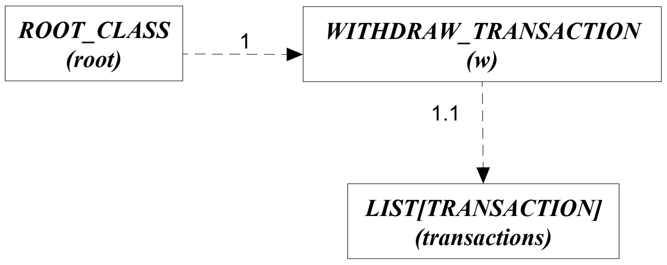
The dynamic diagram of the bank shown in Figure 2-2 is consistent with the static diagram of Figure 2-1. So, we first provide an example of two views that are inconsistent with each other, as that will more clearly illustrate the concepts.

Consider the new withdrawal scenario in the BON Dynamic Diagram (BDD) of Figure 3-4, which we wish to compare with the BON Static Diagram (BSD) of Figure 2-1.

The BDD of Figure 3-4 has an object *w* of type *WITHDRAW_TRANSACTION.* However, there is no corresponding class *WITHDRAW_TRANSACTION* in the BSD of Figure 2-1. There is thus an inconsistency between these two views, which leads to the first consistency constraint:

[3] In [17], these two categories are called *intra-model* and *inter-model* consistency. The rules developed in [17] are informal. In [21] a framework for such tools called XLinkit Is developed, but the user must define their own constraints.

CC1 – Consistency Constraint 1: Each object *o* of type *C* in the BDD has a corresponding type (i.e. class) *C* in the BSD.


ROOT_CLASS (root)
1
WITHDRAW_TRANSACTION (w)
1.1
LIST[TRANSACTION] (transactions)

Scenario: withdraw

1	create a withdraw transaction *w* for account *a1* of $100
1.1	add transaction *w* to *transactions*

Figure 3-4 BON dynamic diagrams of bank example – Withdraw

Suppose we now insert an empty class *WITHDRAW_TRANSACTION* into the BSD. Now, *CC1* is satisfied.

Now consider *message-1* in the BDD (Figure 3-4). This message is invoked by some routine in ROOT_CLASS. Of course, if this class has no routines, then no message can be sent. Thus some routine (say *withdrawal_scenario*) must exist, and this routine must invoke some routine (e.g. *make*) in the target of the message which in our case is the class WITHDRAWAL_TRANSACTION, as shown in the sample code below.

```
withdrawal_scenario: BOOLEAN is
        local
                a: ACCOUNT
                c1: CUSTOMER
                w: WITHDRAW_TRANSACTION
                balance:REAL
        do
                create c1.make ("joe")
                create a.make (c1)
                balance:=a.balance
                create w.make (-100, a)
                Result := c1.account.balance = (balance-100)
        end
```

CC2 – Consistency Constraint 2: Consider a message *m* with source object *src* and target object *tgt*. Let *src* have type SRC and *tgt* have type TGT (by *CC1*, TGT and SRC are guaranteed to exist in the corresponding BSD). Then message *m* must have at least one corresponding feature *r* in class SRC of the BSD, where the body of *r* makes a call *tgt.f*.

Now assume that ROOT_CLASS indeed contains *withdrawal_scenario.* Thus, *CC2* now holds.

However, now there is another problem. There is no guarantee that the feature *make* actually exists in *WITHDRAW_TRANSACTION* (e.g. this class may be empty). What we need is something like the sample code below:

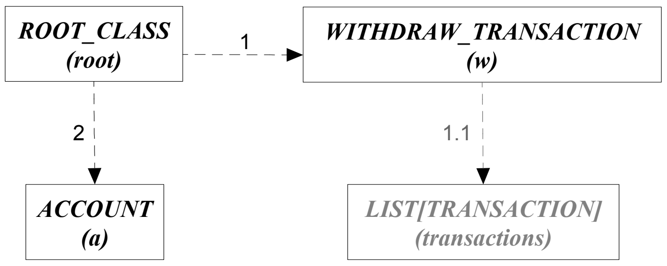
```
make (an_amount: REAL; a: ACCOUNT) is
        require else
                an_amount < 0 and a.balance >= - an_amount
        do
                amount := an_amount
                a.transactions.extend (Current)
        ensure then
                amount = an_amount
                a.balance = old a.balance + an_amount
                a.transactions.has(Current)
                a.transactions.count = old a.transactions.count + 1
        end
```

Suppose we add *make* as above, but export it to *NONE* (i.e. we make it private). Then the source of the message (object *root* in the BDD) still cannot invoke the routine. Thus, *make* must be exported to class *ROOT_CLASS*. Of course, *ROOT_CLASS* is guaranteed to exist by *CC1*.

CC3 – Consistency Constraint 3: Consider a message *m* with source object *src* and target object *tgt*. Let *src* have type SRC and *tgt* have type TGT, and let SRC have routine *r* that makes a call *tgt.f* (as in *CC2*). Then *f* must be a feature of TGT that is exported to SRC[4].

Suppose we now add the feature *make* and export it to *ROOT_CLASS*. Now, *CC2* and *CC3* are satisfied. What else could go wrong in the BDD?

[4] In [1], the rules *CC2* and *CC3* are expressed differently, but the presentation here is clearer from the point of view of understanding the sequel.



Scenario: withdraw

1	create a withdraw transaction *w* for account *a* of $100
1.1	add transaction *w* to transactions
2	check *balance*

Figure 3-5 BON dynamic diagram of bank example – Withdraw 2

Consider the withdraw scenario in the BDD of Figure 3-5 in which message *m1.1* has been greyed out (it is a submessage of message *m1*), and an extra message *m2* has been added, which is a call to the query *balance* of account *a*. The type of *a* is ACCOUNT which has the contract view shown in Figure 3-6. For now, we ignore submessages and focus on the main scenario which is:

$$m1; m2$$

The main idea is that the postcondition of *m1* should be strong enough to entail the precondition of *m2*. We need to assume the existence of a predicate *init* (the system context), which holds initially before *m1* executes, such that

$$init \rightarrow m1.pre$$
where
$$m1.pre \cong make.pre \qquad (1)$$
$$make.pre \cong (an\_amount < 0 \wedge a.balance \geq -an\_amount)$$

where *make* is a feature of the target WITHDRAWAL_TRANSACTION of the message. For example, suppose we select our system context *init* as

$$init \cong an\_amount = -100 \wedge a.balance = 250 \qquad (2)$$

then (1) trivially holds. If (1) does not hold, then this is called a *failed execution*, and init must be strengthened for the procedure to continue.

```
class ACCOUNT feature
          ...
      make (a_customer: CUSTOMER)

      customer: CUSTOMER

      transactions: LIST [TRANSACTION]

      balance: REAL is
              require balance >= 0
              ensure (Result = (∑t ∈ transactions • t.amount)) ∧ (balance = `balance)

end

Note: `balance is an abbreviation for old balance
```

Figure 3-6 Class Account

We must first symbolically "execute" *m1* to obtain its postcondition within the system context, which is

$$init' \cong (\exists \grave{v} \bullet init[v := \grave{v}] \wedge m1.post)$$

i.e. *(3)*

$$init' \equiv (a.balance = 150)$$

using the one point rule to eliminate the existential operator [45]. The single state predicate *init′* is the new system context after message *m1* has been executed. To "execute" message *m2*, we can now repeat the above procedure of (1) and (3), except that instead of *init* in (1) we use the new context *init′*.This procedure can be followed recursively for any sequence of messages

m1; *m2;m3;..;mn.*

CC4 — Consistency Constraint 4: Given a message sequence *m1*; *m2*; *m3*; ..; *mn*, select an initial context *init* and apply the procedure outlined in (1) and (3) recursively, and check that there are no failed executions.

The four consistency constraints can be divided into two categories as follows:

- *Syntactic-consistency*: *CC1*, *CC2* and *CC3*;
- *Contractual-consistency*: *CC4*

Syntactic-consistency is something that can in essence be automatically checked by a suitably complex compiler based on the metamodel for static and dynamic views, and it ensures that the appropriate objects and messages (in the BDD) and classes and relationships (in the BSD) exist and appropriately correlate with each other.

By contrast, contract-consistency cannot be fully automated, because we must prove theorems as shown at steps (1), and (3) above. Proving these theorems, in effect shows the model (consisting of the BDD and BDS) has at least one successful symbolic execution (assuming *init*).

However, we would like to provide the software developer with some mechanized assistance in proving the theorems. The approach followed in [1] is to translate the metamodel and model into predicate logic and to use the PVS theorem prover to semi-automate the proofs. At this point the translation of the model must be done by hand, but a tool could be envisaged that would do the job. The developer will have to interact with the theorem prover during the proof process.

Another semi-automatic approach was outlined in [14]. In this approach, a test driver is constructed from the BDD, using some interaction with the software developer. The test driver and code are compiled together and executed, with the test driver making the calls in the order shown in the scenario box. If during the execution, a contract violation is detected (automatically detected by the runtime), then that contract violation indicates that *CC4* is violated.

However, we believe that there is a flaw in the basic algorithm (see Appendix B). Applying this algorithm for the withdrawal scenario of Figure 3-4, we obtain the test driver in Figure 3-7. The user had to manually enter the code to set up the system context *init*. Since $150 is deposited, a withdrawal of $100 should not trigger a failed execution (and hence a contract violation).

However, if we run the test driver, we get a contract violation. What went wrong? The problem is that *m1.1* is a submessage of *m1*. In *m1*'s target feature (*make* of class

WIHTDRAW_TRANSACTION) was already included the call to *m1.1*'s target feature (*transactions.extend*). In effect, the withdraw transaction has been invoked twice and this leads to a contract violation. We refer to this problem with the algorithm as the *submessaging problem*.

```
class TEST_DRIVER
creation
make
feature
w: WITHDRAW_TRANSACTION
d: DEPOSIT_TRANSACTION
a: ACCOUNT
c: CUSTOMER
feature
make is
   do
  -- manually entered by developer
  -- creates `init'
  create c.make("Joe")
  create a.make(c)
  create d.make (150, a)--initial state

  -- automatically generated by
  -- algorithm in Appendix B
  w.make(-100,a)------------------------------1
  a.transactions.extend (w)-------------------1.1
  check
   a.balance = 50
  end
   end
end -- class TEST_DRIVER
```

Figure 3-7 Test driver generated from Figure 3-4 using the algorithm in Appendix B

Chapter 4 The extended BON Metamodel

In previous chapters we discussed BON and the BON metamodel. In this chapter, we introduce and justify extensions to the BON metamodel which will be needed for the BCCT constraint checking tool.

4.1 BON metamodel used in this thesis

In Chapter 3, we introduced the BON metamodel from [1;3]. In this section, we extend the metamodel so as to better describe multi-view consistency. We present the new metamodel in BON itself as in [3], with consistency constraints written as invariants of the model classes. Algorithms to check these constraints will then be described. These algorithms will be at the heart of the BCCT tool. In the sequel, "metamodel" refers to this new extended model, and "old metamodel" refers to the original one described in the previous chapter.

The authors of reference [1] write "it is not within the spirit of BON to add new views by adding new subclasses of *MODEL*, e.g., *DYNAMIC_MODEL*, etc., as this can easily introduce inconsistency between views". So, they use a class *EXTENDED_MODEL* (Appendix C) that inherits from *MODEL* to describe multiple view consistency and check for consistency by translating all the views to a common model. This approach makes it difficult to introduce inconsistency between views and has traceability problem [25, p34]. For example, if we find an object (in a BDD) that has no corresponding class (in a BSD) as in constraint *CC1*, using only a model, it is difficult to know which view this object belongs to.

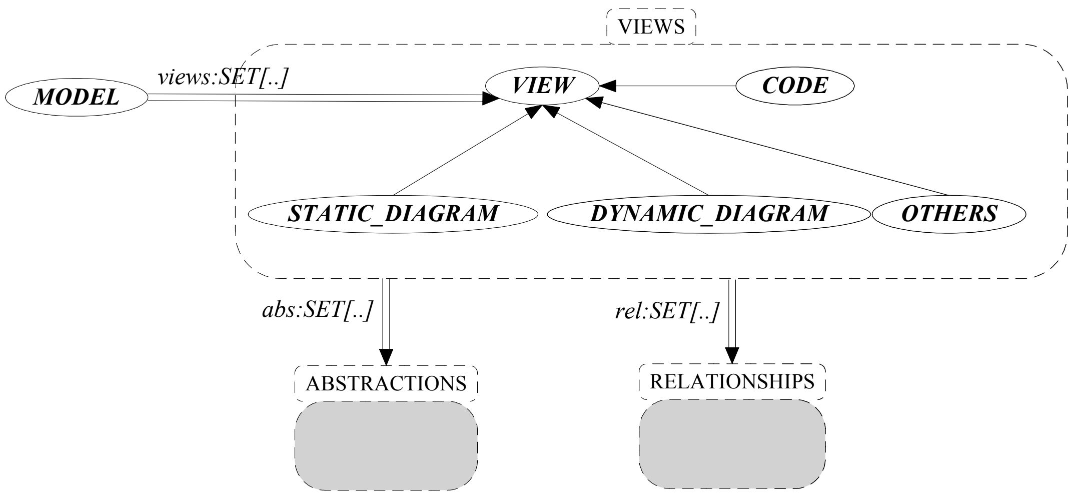


Figure 4-1 New BON metamodel architecture

In the new metamodel we introduce the following:

- A new cluster *VIEWS* to describe the various views of the model. Static and dynamic diagrams now inherit from a class VIEW in this cluster.
- Invariant clauses of class *VIEW* are used to capture single-view consistency, such as the constraint *no_inheritance_cycles* described in the previous chapter.

- Class *MODEL* contains a set of VIEW. Invariant clauses in *MODEL* are used to capture constraints (*CC1-CC4*) and the new constraints introduced in this chapter for multiple view consistency checking.
- In the old metamodel (Figure 3-1;Figure 3-3), the *ABSTRACTIONS* cluster used the *RELATIONSHIPS* cluster. In the new metamodel, the *VIEWS* cluster uses both *ABSTRACTIONS* and *RELATIONSHIPS* (Figure 4-1), to allow simpler capture of multi-view consistency constraints.

In the views cluster, *STATIC_DIAGRAM, DYNAMIC_DIAGRAM* and *CODE* are views. But we can now add new views in future work, such as use cases and state charts.

In the sequel, we focus on static diagrams and dynamic diagrams. A static diagram is an instance of the class *STATIC_DIAGRAM* and a dynamic diagram is an instance of the class *DYNAMIC_DIAGRAM*. MODEL has the following property

views: SET[VIEW]

For simplicity, but without loss of generality[5], we will assume that there is a single static diagram (view *v1*) and a single dynamic diagram (view *v2*) in *views*. We would like to more formally define the notion of $consistency(v1, v2)$, i.e. *v1* and *v2* are con-

[5] The BCCT tool has the ability to store projects; each project can have many static and dynamic diagrams.

sistent precisely when they satisfy consistency constraints such as *CC1*, *CC2*, *CC3* and *CC4*.

In the old metamodel, messages are between dynamic abstractions as shown below [3]:

```
class MESSAGE feature
 source, target: OBJECT
 routine: ROUTINE
 number:INTEGER
invariant
 number >=1
end
```

The target routine (referred to in constraints CC2 and *CC3* of the previous chapter) must be constructed from the source *routine*, where *routine* has a property

calls: SET[CALL]

representing the calls made in the source routine body.

In the new metamodel, we follow the simpler approach of directly including both the source and target features of *MESSAGE* as shown in Figure 4-2. The source routine is *source_feature* and the target feature called by the message is *target_feature.* The *source_feature* is a feature in the *source_object* that invokes this message (i.e. calls the *target_feature*) and the *target_feature* is the feature in the *target_object* that executes this message (i.e. the feature called by the *source_feature*).

```
class MESSAGE feature
 number           : STRING
 message          : STRING
 source_object    : OBJECT
 target_object    : OBJECT
 source_feature   : FEATURE
 target_feature   : FEATURE
invariant
 source_object_exists
 target_object_exists
 source_feature_assigned
 target_feature_assigned
end -- MESSAGE
```

Figure 4-2 Interface of class MESSAGE in the new metamodel

In order to support submessages (see previous chapter), the property *number* of MESSAGE is now a string from which message and submessage numbers can be extracted. Thus, a message “1” may have submessages “1.1” and “1.2”. The message’s *number* and *message* represent the message information in the scenario box. The invariants in *MESSAGE* are to ensure the source object and the target object exist and the source feature and the target feature have been assigned.

For consistency checking we will also need the ability to refer to the suppliers of a class in a BSD. We therefore add a property *suppliers* in the metamodel construct CLASS of Figure 3-3 as follows:

```
suppliers: SET[CLASS]
```

4.2 Mapping the consistency constraints to the new metamodel

In the previous chapter, we indicated that the old metamodel suffers from the submessaging problem. Consistency constraint *CC4* requires that the postcondition of a message (within the system context) must entail the precondition of the next message in the BDD as shown in equations (1) and (3) of Chapter 4. To check *CC4*, the proposal was made in [14] that a testdriver corresponding to the BDD could be derived; the execution of the testdriver without contract failure would indicate that *CC4* holds. As we pointed out, the procedure to semi-automatically deriving the testdriver is flawed.

In this chapter we will provide an algorithm (Algorithm 4-3) that automatically translates the model (BDD and BSD views) into Eiffel code (the complete code or *text-view*). We will also suggest how the testdriver can be semi-automatically derived from the generated code.

Also, the consistency constraints *CC1-CC4* are not complete. What we are lacking is to check that an extended client-supplier path exists between the target class and the source class. A new consistency constraint *messages-invokable* captures this new constraint, and will be described in section 4.3.

Table 4-1 lists the old constraints and how they map to the constraints in the new metamodel. For each constraint, we provide the formal predicate logic description of the constraint, followed by an algorithm to check that the constraint holds.

Constraints		**Algorithm**
Old	**New**	
CC1	*object-class*	Algorithm 4-1
CC2	*message-feature*	Algorithm 4-2
CC3		
CC4	*contractual-consistency*	Algorithm 4-3 (generate code)
	messages-invokable	Algorithm 5-4 (specified depth)

Table 4-1 Constraints and algorithms

The metamodel constructs STATIC_DIAGRAM, DYNAMIC_DIAGRAM, CLASS, OBJECT, and FEATURE are shown in Figure 4-3 to Figure 4-7.

```
class STATIC_DIAGRAM feature
 name       : STRING
 classes    : SET[CLASS]
 rel        : SET[STATIC_RELATIONSHIP]
 csrels     : SET[CLIENT_SUPPLIER_REL]
 irels      : SET[INHERITANCE_REL]
 closure_cs : SET[CLIENT_SUPPLIER_REL]
    -- set of all client-supplier closures
end
```

Figure 4-3 Interface of STATIC_DIAGRAM

```
class DYNAMIC_DIAGRAM feature
 name        : STRING
 objects     : SET[OBJECT]
 messages    : SET[MESSAGES]
 message_seq: LIST[MESSAGES]
 scenario    : SCENARIO
end -- DYNAMIC_DIAGRAM
```

Figure 4-4 Interface of DYNAMIC_DIAGRAM

```
class CLASS feature
 name              : STRING
 features         : SET[FEATURE]
 invariant        : BOOLEAN
 suppliers        : SET[CLASS]
end -- CLASS
```

Figure 4-5 Interface of CLASS

```
class OBJECT feature
 name        : STRING
 class       : CLASS
end -- OBJECT
```

Figure 4-6 Interface of OBJECT

```
class FEATURE feature
 name              : STRING
 type              : [ATTRIBUTE,QUERY,COMMAND]
 accessors         : SET[CLASS]
 exported_type     : [NONE, ANY, SELECTED]
 pre_condition     : BOOLEAN
 post_condition    : BOOLEAN
 calls             : SET[CALL]
end - FEATURE
```

Figure 4-7 Interface of FEATURE

Constraint *CC1* asserts that each object in view *v2* in the BDD must have a corresponding class in the BSD (in view *v1*). This constraint can be described as follows:

Constraint object-class:

$v1 \in STATIC\_DIAGRAM$

$v2 \in DYNAMIC\_DIAGRAM$

$object\_class(v1, v2) \cong \forall o \in v2.objects \bullet \exists c \in v1.classes \bullet o.class = c$

where STATIC_DIAGRAM and DYNAMIC_DIAGRAM are shown in Figure 4-3 and Figure 4-4 respectively.

The constraints *CC2* and *CC3* in the old metamodel (Chapter 4) are both related to messages and features. In [1], *CC2* is formalized as a PVS logical description, while *CC3* is left an informal part of the BON metamodel. There was no need to formalize *CC3* as this is in essence checked by any Eiffel compiler.

Since we want our BCCT tool to do this check we must formalize *CC3* as well. In the new metamodel it is convenient to treat *CC2* and *CC3* as a single constraint called *message-feature*(*v1*, *v2*):

Constraint message-feature:

$v1 \in STATIC\_DIAGRAM$

$v2 \in DYNAMIC\_DIAGRAM$

$$
\begin{aligned}
& message\_feature(v1, v2) \cong \\
& \quad object\_class(v1, v2) \\
& \quad \wedge \forall m \in v2.messages \bullet \\
& \qquad (\exists sf \in m.source\_object.class.features \bullet \\
& \qquad\quad (\exists tf \in m.target\_object.class.features \bullet \\
& \qquad\qquad (m.source\_feature = sf \wedge m.target\_feature = tf \wedge tf \in sf.calls)))
\end{aligned}
$$

where

$object\_class(v1, v2) \rightarrow m.target\_object.class \in v1.classes$

$object\_class(v1, v2) \rightarrow m.source\_object.class \in v1.classes$

Constraint *message-feature* omits one aspect of CC3 – it does not check that *tf* is exported to *source_object.class*. This check is now done in the *messages-invokable* constraint which we describe in the next subsection. The inner predicate of *message-feature* could in fact be written in class MESSAGE.

4.3 Path closure problem

Consider the two views of a model (a BSD and a BDD) in Figure 4-8 in which object *a* sends message *m1* to object *c*. The sending routine r has a call $b.z.c.f$, where $z = (a.b)*$, i.e. there may be zero or more double dotted calls *a.b*. The call $b.z.c.f$ is an element of property *calls* in the metamodel construct FEATURE in Figure 4-7. The constraint *message_feature* indeed checks that appropriate routines *r* and *c.f* exist.

But, there is still no guarantee that there is a path in the class diagram (i.e. in the BSD) between A and C that corresponds to a call such as *b.z.c.f.* We call this the *path-closure* problem.

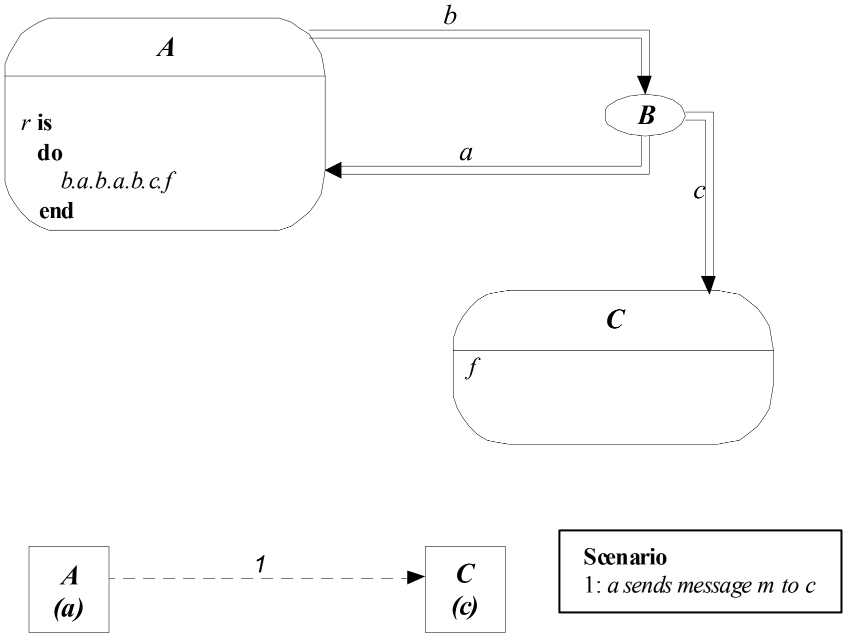


Figure 4-8 Recursive message check

Obviously, we must add an additional consistency constraint to check that there is an appropriate path in the class diagram corresponding to a call such as *b.z.c.f.* We define a new consistency constraint in addition to the already enumerated constraints (see Table 4-1). This new constraint is called the *messages-invokable* consistency constraint.

Constraint messages-invokable

$v1 \in STATIC\_DIAGRAM$
$v2 \in DYNAMIC\_DIAGRAM$

$messages\_invokable(v1, v2) \cong$
$\quad object\_class(v1, v2) \wedge (\forall m \in v2.messages \bullet client\_supplier \wedge export)$

where
$client\_supplier \cong$
$\quad (\exists r \in v1.closures\_cs \bullet r.source = m.source\_object.class \wedge r.target = m.target\_object.class)$

$export \cong (m.source\_object.class \in m.target\_feature.accessors)$

$closures\_cs : SET[CLIENT\_SUPPLIER\_REL]$
$\quad$ **ensure** $Result = \{r \in rel \mid r : CLIENT\_SUPPLIER\_REL\} \cup$
$\qquad \{r : CLIENT\_SUPPLIER\_REL \mid$
$\qquad\quad (\exists r1, r2 \in rel \mid r1, r2 : CLIENT\_SUPPLIER\_REL \wedge r1.source = r2.target) \bullet$
$\qquad\quad (r.source = r2.source \wedge r.target = r1.target)\}$

In the previous chapter we defined the contractual consistency of a message sequence *m1*; *m2*; … *mn* in a BDD via the recursive application of the precondition/postcondition constraints (1) and (3) in Chapter 3. In these constraints, the messages are elements of view *v2* (the BDD), and the preconditions and postconditions of the source and target features of the messages are elements of view *v1* (the BSD). Thus *contractual_consistency*(*v1*, *v2*) is defined by the recursive application of (1) and (3), and we thus have:

$$consistency(v1, v2) \cong$$
$$\quad syntactic\_consistency(v1, v2) \wedge contractual\_consistency(v1, v2)$$

where

$$contractual\_consistency(v1, v2) \cong$$
$$\quad object\_class(v1, v2)$$
$$\wedge\, message\_feature(v1, v2)$$
$$\wedge\, messages\_invokable(v1, v2)$$

4.4 Proposed BCCT tool

In the next section we provide the algorithms for checking $consistency(v1, v2)$. These algorithms will be part of the BCCT tool. To use the proposed BCCT tool, a software developer proceeds as follows:

- Construct a model consisting of a BSD and BDD.
- Add detailed code to the bodies of routines in the model until the syntactic consistency checks pass.
- Use Algorithm 4-3 to automatically translate from the model to Eiffel code, and use a testdriver (developed in part by hand as described in the *with-drawal-scenario* in Chapter 3) to execute the code. A successful run of the testdriver without contract violations confirms that there is at least one execution of the system that conforms to the message sequence in the BDD.

4.5 Consistency checking algorithms

In the previous sections we defined the constraints needed to check syntactic and contractual consistency between dynamic and static diagrams. We now supply algorithms for syntactic consistency as well as code generation for contractual consistency checking via testdrivers.

The algorithm for the *object_class* constraint is provided in Algorithm 4-1. We need only traverse the dynamic diagram, object by object, and check if each object has a corresponding class. The set of objects that have no corresponding class is stored in entity *object_no_class*.

The algorithm for the *message-feature* constraint is provided in Algorithm 4-2. We simply traverse the dynamic diagram, message by message, and check for appropriate source and target features. The entity *message_no_feature* is used to store messages that do not have the appropriate source and target feature match.

```
class MODEL feature
        …
        object_no_class:SET[OBJECT]

        object_class (v1,v2): BOOLEAN is
          --where v1:STATIC_DIAGRAM, V2:DYNAMIC_DIAGRAM
        local
          j:INTEGER
        do
          from
           j:=v2.objects.lower
          Result := true
          until j = v2.objects.upper
          loop
            if (v1.classes.not_occur(v2.objects@j.class))
              Result := false
              object_no_class.add(v2.objects@j)
            end
              j:=j+1
          end
        end
        …
end
```

Algorithm 4-1 Algorithm *object-class* for checking if each object has a corresponding class

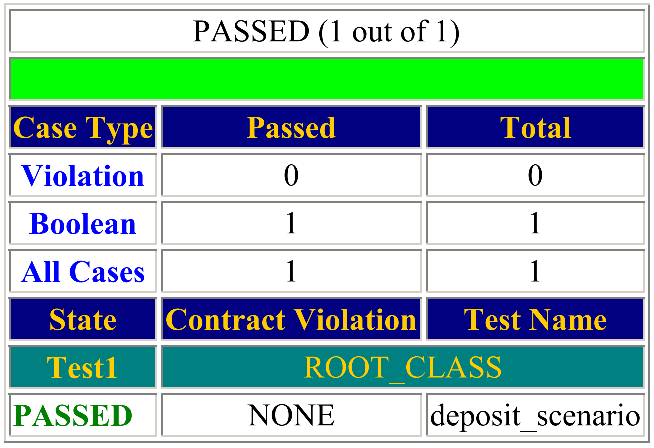
```
class MODEL feature
  …
  message_no_feature: SET[MESSAGE]

  message_feature(v1,v2):BOOLEAN is
    --v1:STATIC_DIAGRAM,v2:DYNAMIC_DIAGRAM
  local
   j:INTEGER
   m:MESSAGE
  do
   from
     Result := true
     j:=v2.messages.lower
   until j = v2.messages.upper
   loop
     m := v2.messages@j
     if(m.source_object.class.features.not_occur(m.source_feature)
        Result := false
        message_no_feature.add(m)
     elseif(m.target_object.class.features.not_occur(m.target_feature)
        Result := false
        message_no_feature.add(m)
     elseif(m.source_feature.calls.not_occur(m.target_feature))
        Result := false
        message_no_feature.add(m)
     end
     j:=j+1
   end
  end
  …
end -- MODEL
```

Algorithm 4-2 Algorithm *message-feature* for checking source and target feature match

The algorithm for *messages_invokable is* described in Chapter 5, where we also discuss the possibility of supplying the software developer with all calls from the source routine to a target routine to a specified depth.

Algorithm 4-3 describes how we generate code for each class in the model. On the basis of this code we can develop a testdriver to check contractual consistency. Etester [15] provides a unit testing framework for developing testdrivers. For example, the deposit scenario in Figure 2-2 is easily converted to an Etester test. The report for such a test is shown in Figure 4-9.



PASSED (1 out of 1)		
🟩		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	1	1
All Cases	1	1
State	**Contract Violation**	**Test Name**
Test1	ROOT_CLASS	
PASSED	NONE	deposit_scenario

Figure 4-9 Report of unit test

If there is no object's corresponding class is a root class in the static diagram, users then have to use Etester to generate a test driver to test the generated code.

```
class MODEL feature
        …
        generate_code (v1:STATIC_DIAGRAM) is
        local
          code      : CODE
          classes   : SET[CLASS]
          j         : INTEGER
          class     : CLASS
        do
          generate_system_code --generate .ace file
          from j:=v1.classes.lower
          until j = v1.classes.upper
          loop
            class := v1.classes@j
            code := creat_new_class_code (class.name)
            class.generate_descriptions(code)
            code.add ("CLASS",class.name)
            code.generate_create_procedure(class.create)
            class.generate_inherit_code(code)
            class.generate_suppliers_code(code)
            class.generate_features(code)
            class.generate_invariant(code)
            code.write_to_file
            j:=j+1
          end
        end
        …
end -- class MODEL
```

Algorithm 4-3 Algorithm for code generating

In this chapter, we have discussed algorithms for consistency checking. The implementation of these algorithms described will be introduced in Appendix D.

Chapter 5 Specified Depth Algorithm

In the previous chapter (section 4.3) we illustrated and discussed the *path-closure problem*. As part of consistency checking we must check that if (in a BDD view *v2*) an object *a* (of type A) sends a message to object *c* (of type C), then in the BSD view *v1*, a client-supplier path must exist from A to C.

A consistency constraint $messages\_invokable(v1, v2)$ was developed in the previous chapter to describe the appropriate constraint. In this chapter we develop Algorithm 5-4 to check this constraint. The constraint is given by

$$\begin{aligned} &messages\_invokable(v1, v2) \cong \\ &\quad object\_class(v1, v2) \wedge (\forall m \in v2.messages \bullet client\_supplier \wedge export) \end{aligned}$$

We first illustrate checking the *export* predicate in the constraint. Figure 5-1 shows a static diagram with six classes, and Figure 5-2 shows a dynamic diagram with two objects. In the dynamic diagram, object *a* is of type A and *b* is of type *B*. *A* and *B* are specified in the static diagram. There are two checks that must be done for *messages_invokable*. Object *a* sends message *m1* to *b*. Message *m1*'s source feature is *sf* and target feature is *tf*. If *A* is in the accessors' list of *tf* (i.e. feature *tf* is exported to A) then *export* is satisfied.

We noticed towards the end of this thesis that *export* needs a more detailed check. For example, if the call is *a.b.c.tf*, then *tf* must be exported to C, *c* to B, and *b* to A. However, this additional constraints has not been implemented in BCCT.

We must now check *client_supplier* – is there a client-supplier relationship between *A* and *B*? There are many client-supplier links (directly and indirectly) between *A* and *B*. Because there are cycles such as *E* can call *F* and *F* can call *E,* many possible calls in the source routine are possible. Here are some of them:

```
b.tf
e.f.b.tf
e.f.e.f.b.tf
e.f.e.f.e.f.b.tf
e.f.e.f.e.f………b.tf
…
```

There are an infinite number of legal multi-dot calls. With inheritance the problem is more complex.

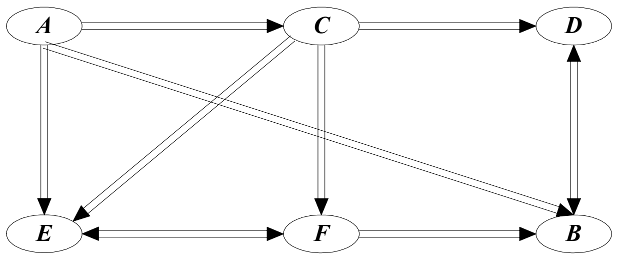


Figure 5-1 Static Diagram with six classes

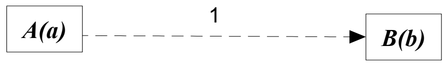


Figure 5-2 Dynamic Diagram with two objects

5.1 Links

Figure 5-3 shows three different kinds of client-supplier *links* between a class *A* and a class *C*: a *direct link*, an *indirect-link* and an *ancestor-link*.

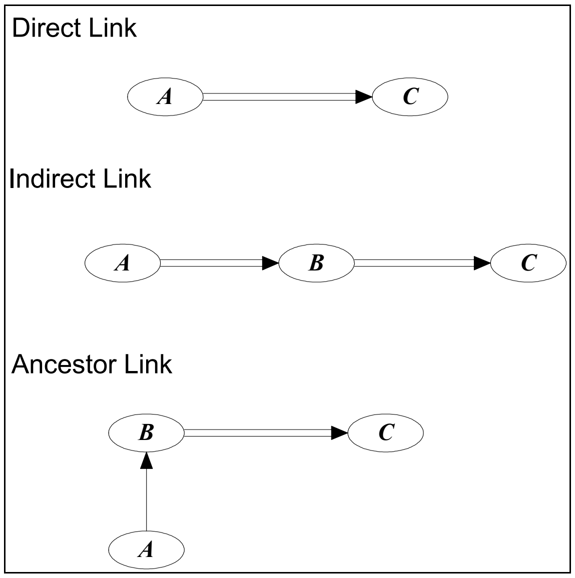


Figure 5-3 Kind of links

We will provide algorithms for checking each of these links separately, which then makes it easier to describe the final procedure in Algorithm 5-4. These algorithms refer to properties of the metamodel constructs developed in the previous chapter.

5.1.1 Direct links

Figure 5-4 shows a direct link. In the static diagram, the source class (*ROOT_CLASS*) is a direct client of the target class (*CUSTOMER*) and the source class will call a feature of the target class directly to execute message *m1* (which is *c1.make*).

This is the simplest kind of link. It is only necessary to check that *ROOT_CLASS* is one of CUSTOMER's client classes. The procedure for this kind of link is shown in Algorithm 5-1. The complexity of Algorithm 5-1 is *O*(*V*) with *V* classes in the static diagram.

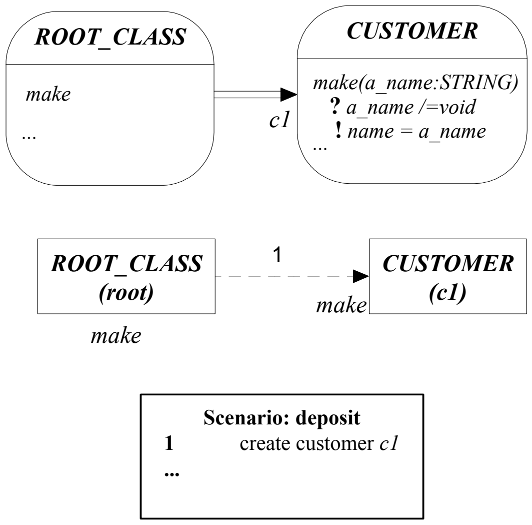


Figure 5-4 An example of directly linked

```
direct_link_check(a,b:CLASS):BOOLEAN is
 do
   if a.suppliers.occurrences(b) > 0
     Result :=  true
   else
     Result := false
   end
 end
```

Algorithm 5-1 Algorithm for direct linked checking

5.1.2 Indirect links

Figure 5-5 shows two classes (*ROOT_CLASS* and *ACCOUNT*) that have a direct client-supplier relationship and an indirect one (through a third class *CUSTOMER*). There are thus two paths between *ROOT_CLASS* and *ACCOUNT*. In addition, there is also a cycle between CUSTOMER and ACCOUNT.

In the dynamic diagram, the source feature *make* of object *root* sends messages *m2* and *m4* to object *a1*. Message *m2* refers to the creation command

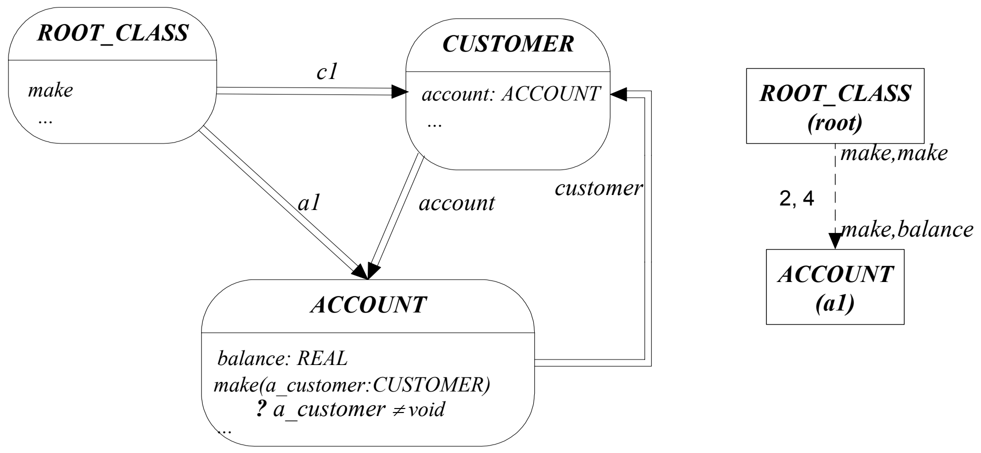
```
create a1.make(c1)
```

which calls the feature *make* of ACCOUNT . Message *m4* refers to the assignment

```
c1_balance := c1.account.balance
```

which includes a call to the query *balance* of ACCOUNT. Message *m4* is executed through an indirect link.

In standard BON dynamic diagrams, source and target features are never described. However, our consistency checking does require these features to be identified. We therefore extend the BON diagram with this information (which is managed by the BCCT tool based on user input). In the dynamic diagram of Figure 5-5, the source feature for message *m2* (respectively *m4*) is *make* and the target feature for m2 (respectively *m4*) is *make* (respectively *balance*).



Scenario: deposit

...
2 create account *a1* with *c1*
...
4 check balance

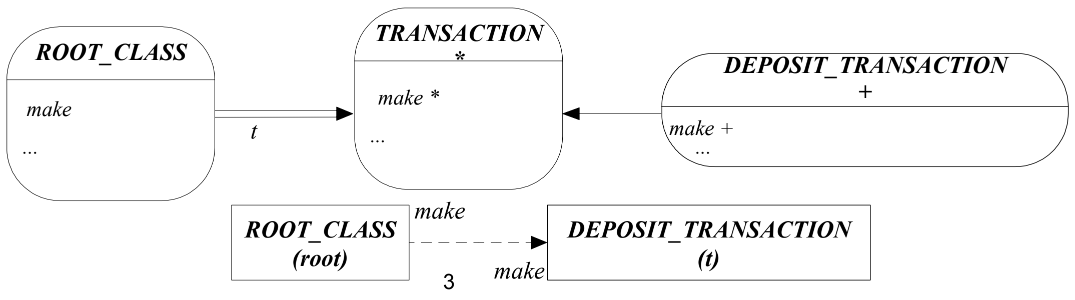
Figure 5-5 An example of an indirect link

We can modify Algorithm 5-1 to get Algorithm 5-2 to check if there is a (possibly indirect) path between two classes *A* and *B*. In order to ensure termination within a reasonable time frame, we conduct a search only to a pre-specified depth *M*. In practice, we can get an upper bound on *M* by examining the feature body for multi-dot calls. We first check if *B* is the direct supplier of *A;* if not, we will check if *B* is *A*'s supplier's supplier class and so on recursively. The complexity of Algorithm 5-2 is $O(M.V^3)$ because the union of two sets of size $O(V)$ is cubed.

```
indirect_link_check(a,b: CLASS):BOOLEAN is
 local
   classes: SET[CLASS]
   i: INTEGER
   j: INTEGER
 do
   classes := a.suppliers
   from
     i:=1
     Result := false
   until i > M
     if classes.occurrences(b) > 0
     then
       Result := true
       i := M+1
     else
       classes := $(\bigcup s.suppliers \mid s \in classes)$
      i := i+1
   end
end
```

Algorithm 5-2 Algorithm for indirectly linked checking

5.1.3 Ancestor links



Scenario: deposit

...	
3	create a deposit transaction *t* for account *a1* of $100
...	

Figure 5-6 An example of an ancestor link

In Figure 5-6, we show message *m3,* taken from Figure 2-1 and Figure 2-2. The source feature is *make* in ROOT_CLASS and the target feature is (apparently) the deferred feature *make* in TRANSACTION. However, the actual target feature is the routine make in the descendant DEPOSIT_TRANSACTION, i.e. the message corresponds to a feature call

```
create {DEPOSIT_TRANSACTION} t.make (100,a1)
```

Thus, in the dynamic diagram, message *m3* is sent from object *root* (corresponding to class *ROOT_CLASS*) to object *t* (an instance of DEPOSIT_TRANSACTION). Although object *root*'s corresponding class *is* not a client of object *t*'s corresponding class, the static and dynamic diagrams are nevertheless consistent. This is because of the *feature call rule* in [9, P473] – "if a feature call *x.f,* where the type of x is based on a class *C*, feature *f* must be defined in one of the ancestors of *C". Ancestors of C,* in this rule, include *C* itself. Thus, **`create`** `{DEPOSIT_TRANSACTION} t.make (100,a1)` is a valid call.

Thus, when checking if two classes are linked or not, we must take their ancestors into account. Algorithm 5-3 shows an algorithm, based on Algorithm 5-1 (direct links), which considers the case in which two classes are linked by their ancestors. This algorithm uses two loops to check if the source class's ancestor is the target class's ancestors' client class. The complexity of Algorithm 5-3 is $O(V^3)$. This is because there is an inner loop within the outer loop (over classes in the BSD); the inner loop uses the feature *occurrences* which itself enumerates over classes in the BSD.

When we take the indirectly linked cases into account, the algorithm will be more complicated.

Algorithm 5-1 to Algorithm 5-3 returns a true if there is at least one path that links the source class to the target class. However, we would like a procedure that will actually return a list of all possible paths. In the next section, we describe the specified depth algorithm .

```
directly_linked_ancestor_check(a,b:CLASS):BOOLEAN is
  local
   i,j:INTEGER
  do
   from
     i:= a.ancestors.lower
     Result := false
   until i = a.ancestors.upper
   loop
     from j := b.ancestors.lower
     until j > b.ancestors.upper
     loop
      if (a.ancestor)@i.suppliers.occurrences((b.ancestors)@j)>0
      then
          Result :=  true
          j := b.ancestors.upper
      else j:=j+1
      end
     end
     if Result then i:=a.ancestors.upper
     else i:=i+1
     end
   end
  end
```

Algorithm 5-3 Algorithm for direct and ancestor links

5.2 Specified depth algorithm

In the previous sections of this chapter we described the various kinds of links between two classes in a BON static diagram, i.e. direct, indirect and ancestor links. We now need an algorithm to generate all the links between A and B to a specified depth *M* for the *messages-invokable* constraint.

Consider the BON static diagram in Figure 5-7 which allows for many links between two classes A and B. A inherits from O and B inherits from S. There are actually 7 links in this example to depth of *M* = 4, i.e.

```
A.B
O.S
A.C.F.B
A.E.F.B
A.C.D.B
A.B.D.B
A.C.E.F.B
```

The largest multi-dot is A.C.E.F.B which has four levels of "dots" corresponding to *M* = 4. Normally the dot notation is used with entities. For the purposes of this algorithm we use the dot notation to indicate links between classes. Thus *A.B* means that A has some attribute (say b) that is of type B.

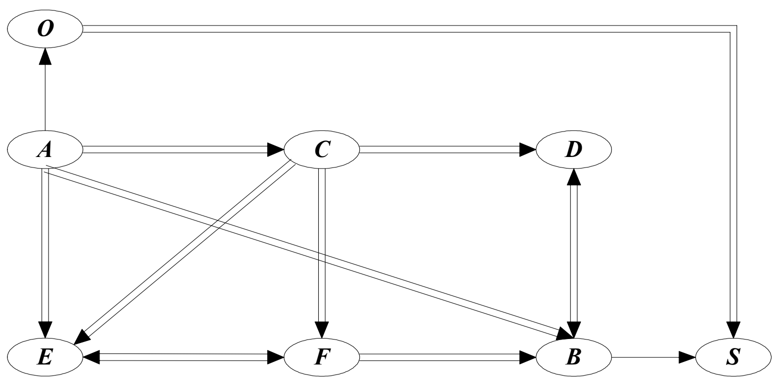


Figure 5-7 A BON class diagram

Because we want to record all the links from A to B, we introduce a data structure *ITEM* shown in Figure 5-8 to record the classes on the path. The value of the attribute *class* is a class in the static diagram, like *A*, *B* or other classes. This class has two queries: *linked* which is a *SET*[*ITEM*] corresponding to the direct links of the target *class*. For example, consider the target class *C* in Figure 5-7. An instance of *ITEM* corresponding to *C* would have *class* = *C* and *linked* = {item corresponding to *D*, item corresponding to *E*, item corresponding to *F*}, as classes D, E and F are the direct links. A SET of ITEM is sufficient as we are merely generating all links such as C.D, C.E and C.F, the order in which these links are enumerated not being relevant.

```
class ITEM feature
 class: CLASS
  -- a CLASS in a static diagram
 linked: SET[ITEM]
  -- items whose `class' are direct client classes

 make(c: CLASS, i: ITEM)is
  require
   c /= Void and i /= Void
  ensure
   class = c
   linked.has(i)
   linked.count = 1

 add(a_linked like linked)is
  require
   a_linked /= Void
  ensure
   linked = old linked + a_linked
     -- extend `linked' with items in `a_linked'
 end
```

Figure 5-8 The interface of data structure *ITEM*

In the section dealing with ancestor-links, we indicated that if ancestors of the source class and target class of a message have a client-supplier relationship, then this is also a link that may have a corresponding message in the dynamic diagram.

```
class LINKS feature
 a,b: CLASS
   -- source and target class

 sources: LIST[ITEM]
   -- includes `a' and all its ancestors

 target: LIST[ITEM]
   -- includes `b' and all its ancestors

 m: INTEGER
   -- depth of client-supplier links

 links: LIST[ITEM] is
   require
     a /= Void and b /= Void
   ensure
     Result = all links between `a' and 'b'
     -- see Algorithm 6-4
 ancestors(l:LIST[ITEM]):LIST[ITEM] is
   require
     l /= Void
   ensure
     Result = all ancestors of 'l'
invariant
 a /= Void and b /= Void
 m > 0
end
```

Figure 5-9 Data for the calculation of all possible links to depth *M*

The specified depth algorithm is described as a function *links* in Figure 5-9 in which *a* refers to the source class A and *b* to the target class B. The function *links* returns all the links between *a* and *b*. The function *sources* includes *a* and all its ancestors, and *targets* includes *b* and all its ancestors as discussed in the section dealing with ancestor-links.

We illustrate the algorithm for *links* by using the class diagram in Figure 5-7 as an example. Figure 5-10 and Figure 5-11 illustrate how all the links between A and B are calculated to a depth of *m* = 4. Figure 5-10 has 7 columns each column being a list of ITEM:

- The first column is *sources* (i.e. *a* and all its ancestors)
- The next 4 columns are *mid* [1], *mid* [2], *mid* [3] and *mid* [4]. The column *mid* [1] is a list of all classes one-step away from *sources*. The column *mid* [2] is a list of all classes that are two steps away from sources, as well as information that allows us to re-trace the path back to *sources* (see *linked* in Figure 5-8). There are 4 such columns because *M* = 4.
- The 6th column is *targets* (i.e. *b* and all its ancestors).
- The last column is the *result* of routine *links*, from which the set of all links can be reconstructed as shown in Figure 5-11.

The algorithm proceeds as follows. The first item in *sources* is class A. The direct suppliers of A are C, E and B (see Figure 5-7). Thus in *mid* [1] we must store

A.C
A.E
A.B

The other item in *sources* is O from which we obtain

O.S

Since B and S (the end of the path) are in *targets*, we copy over A.B and O.S to the column *result*, and we have our first two (of seven) links.

We must now construct all paths two steps away from the source, i.e. *mid* [2]. The first item in *mid* [1] is A.C (i.e. the *class* of the item is C, and the *linked* of the item contains just A). The two-step path is

A.C.D
A.C.F
A.C.E

Obviously, there is no need to keep track of the source A. We know that C is one step away from the source A. Thus we need record only C and all links one-step away from C, i.e.

C.D
C.F
C.E

Thus, C.D is stored as an item in *mid* [2] with D the *class* and C an item in *linked* (C's item has its own field *linked* which points back to A). Thus item C.D allows us to re-trace the path to the source A. Proceeding in a similar way for the remaining items in *mid* [1], we obtain for A.E

E.F

for A.B we obtain,

B.D

and for O.S there is no item as S has no suppliers.

We note that items C.D and B.D in *mid* [2] both represent paths that terminate in D. For efficiency, we can merge these two items into a single item CB.D where D is the *class* of the item and C and B are in the field *linked* of the item. Since both items terminate in D, we need only explore (for *mid* [3]) from D and onwards. Similarly, items C.F and E.F can be merged into CE.F. Thus the items now stored in *mid* [2] are:

BC.D
CE.F
C.E

The terminating classes for these items (D, F and E) are not in *targets*, thus *result* stays the same. The same procedure can now be followed for *mid* [3] and then *mid*[4].

This will terminate with the 7th column *result* which contains the items:

A.B
O.S
FD.B
F.B

From these items, we can reproduce the seven paths from the targets B and S, to *sources* as shown in Figure 5-11, by following the *linked* field of each item. The complete algorithm is shown in Algorithm 5-4.

From Figure 5-11, we directly obtain all paths links when the call depth is 4. The complexity of the algorithm is $O(MV^3)$ because there are 3 loops.

```
links: LIST[ITEM] is
  local
    mid[1..m], t_list: LIST[ITEM]
    item,item1,item2,item3: ITEM
    i,j,k: INTEGER
  do
   t_list := sources
    from i:=1 until i > m loop
      from j := t_list.lower until j = t_list.upper loop
       item:=t_list@j
       from k := item.class.suppliers.lower
       until k = item.class.suppliers.upper loop
        create item1.make(item.class.suppliers@k, item);
        if ∃item2 ∈ mid@i • item2.class = item1.class then
          item2.add(item1.linked);
        else
          mid@i.extend(item1);
          if ∃item3 ∈ targets • item3.class = item1.class then
            result.extend(item1)
          end
        end; k:=k+1
      end; j:=j+1
    end
    mid@i := ancestors(mid@i)
    t_list := mid@i; i:=i+1
  end
```

Algorithm 5-4 Algorithm to calculate all links between classes A and B (see Figure 5-9)

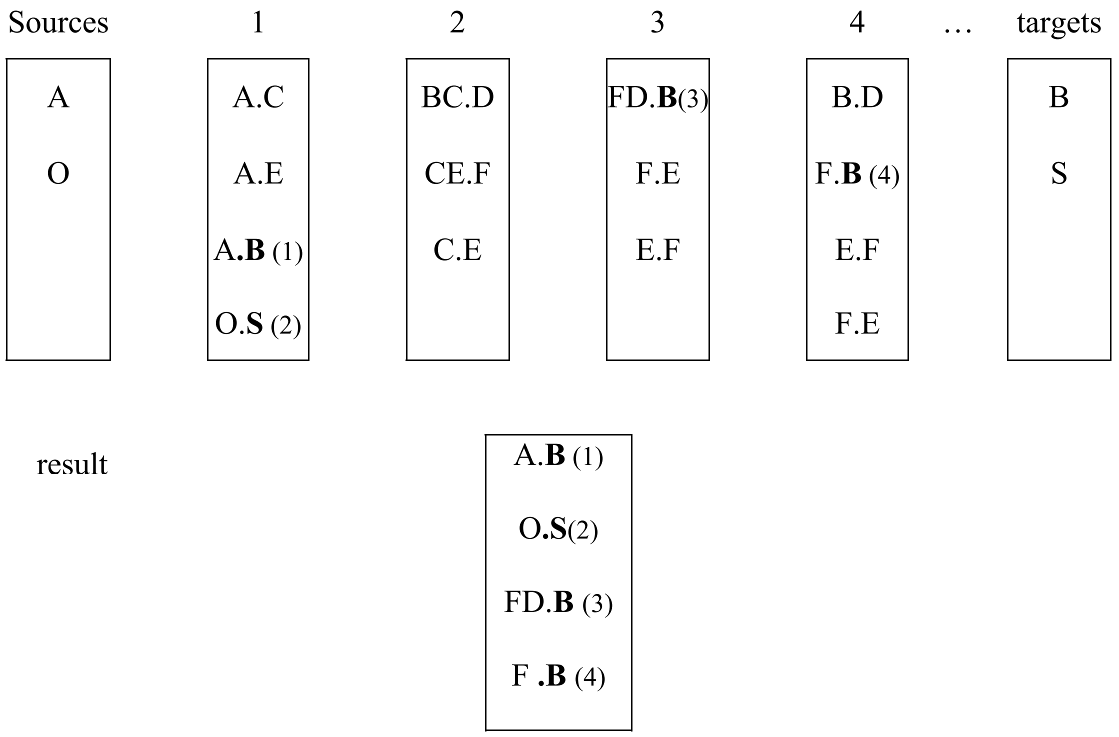


Figure 5-10 the running result of Algorithm 5-4

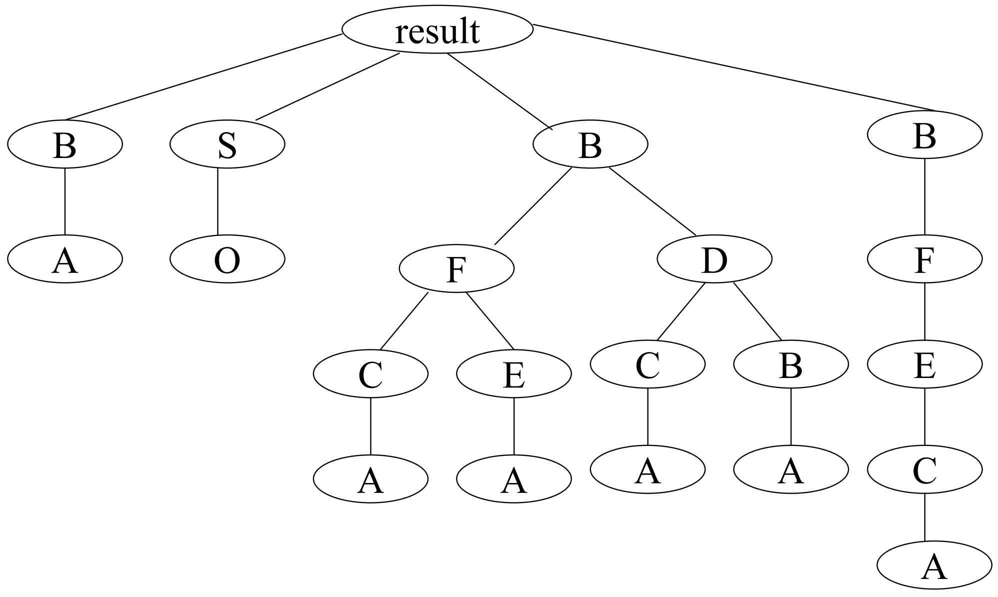


Figure 5-11 Tree structure of the result

5.3 Tolerant versus Strict consistency checking

Consider the static and dynamic diagrams shown in Figure 5-12 (*view1* and *view2*), and suppose that no further information about the classes A, B and C is available, i.e. we do not know what features exist in these classes and what their export status is. What we do know from the class diagram is that B is a supplier of A, and C a supplier of B. Thus, the class diagram could *potentially* be completed to be consistent. This would yield a *tolerant consistency* criterion for partial views.

The definition of consistency adopted in this thesis is *strict*. Thus, source and destination features must exist for each message.

An advantage of the specified depth algorithm is that it can be used in a tolerant version of consistency, as it merely checks client-supplier links in a class diagram.

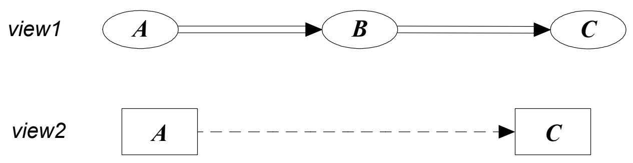


Figure 5-12 Tolerant Consistency between *view1* and *view2*

Chapter 6 The Bon Consistency Checking Tool

In this chapter we describe BCTT, i.e. the BON Consistency Checking Tool. The tool is based on the consistency criteria and algorithms for consistency checking developed in Chapter 4 and Chapter 5. Table 4-1 (Chapter 4 page 49) provides the four constraints that must be checked to show that a BON static diagram (*view1*) is consistent with a BON dynamic diagram (*view2*). The constraints are:

- *object-class* (i.e. every object in *view2* has a corresponding class in *view1*);
- *message-feature* (i.e. each message has an associated target feature that is called from an appropriate source feature and exported to it);
- *messages-invokable* (i.e. there is an appropriate client-supplier link in *view1* associated with each message in *view2*);
- *contractual-consistency* (i.e. the postcondition of each message entails the precondition of the succeeding message).

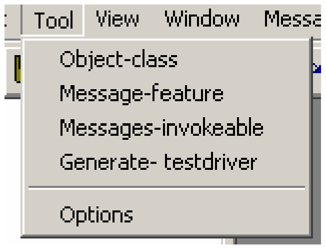
In the BCCT tool, the first three constraints are automatically checked via algorithms supplied in Chapter 4 and Chapter 5. The BCTT tool accomplishes the following:

- automatically checks if the constraint is satisfied (yes or no); and
- provides the user with precise feedback where the consistency fails; and
- provides the user with some mechanized support to fix the problems so that consistency can be restored.

As explained in Chapter 4, the *contractual-consistency* constraint is not checked automatically by BCCT. Instead, some mechanized support is supplied in the following way:

- Algorithm 5-3 is used to automatically generate the code associated with the model; and
- The designer uses the generated code to construct an ETester test corresponding to the dynamic diagram. If the test succeeds, then we have demonstrated at least one execution that satisfies the *contractual-consistency* constraint.

As an illustration of consistency checking, consider the simple model in Figure 6-1 consisting of a static and dynamic BON diagram. The message in the dynamic diagram does not have an associated client-supplier link in the static diagram. Invoking the *Tools* menu provides the following options:



If we invoke the first option *object-class*, the tool immediately reports *Pass* as this constraint obviously holds. However, if we check *messages-invokable*, then the tool provides the error report in Figure 6-2, which indicates that there is no client-supplier link from class A to class D, as required by the message.

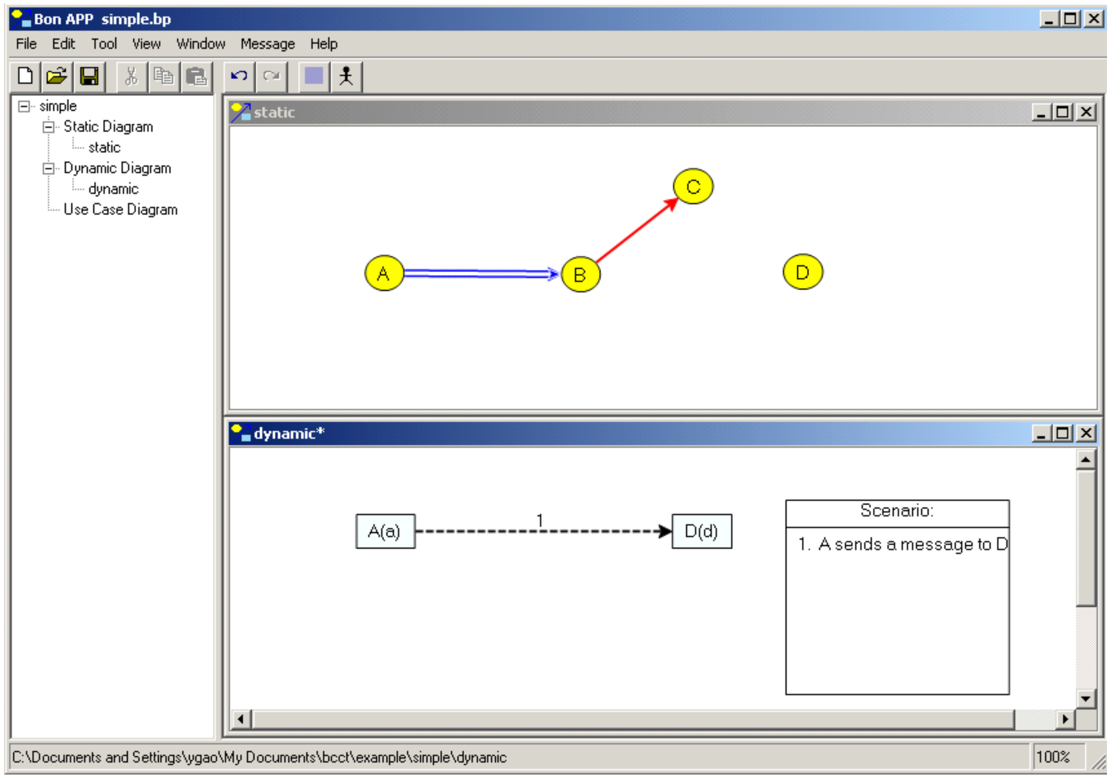
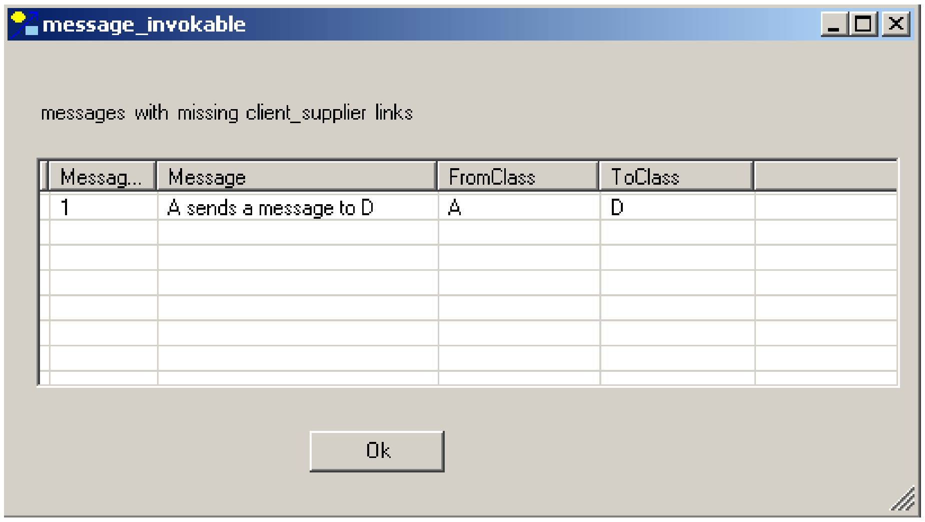

Bon APP simple.bp
File Edit Tool View Window Message Help
simple
Static Diagram
static
Dynamic Diagram
dynamic
Use Case Diagram
static
A
B
C
D
dynamic*
A(a)
1
D(d)
Scenario:
1. A sends a message to D
C:\Documents and Settings\ygao\My Documents\bcct\example\simple\dynamic
100%

Figure 6-1 Two views of a model



messages with missing client_supplier links

Messag...	Message	FromClass	ToClass
1	A sends a message to D	A	D

Figure 6-2 Report of missing client-supplier links for the model in Figure 6-1

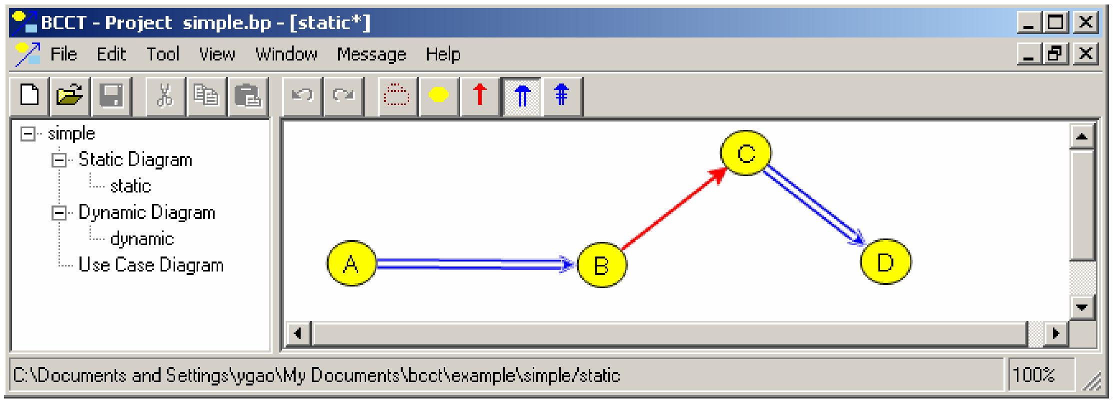
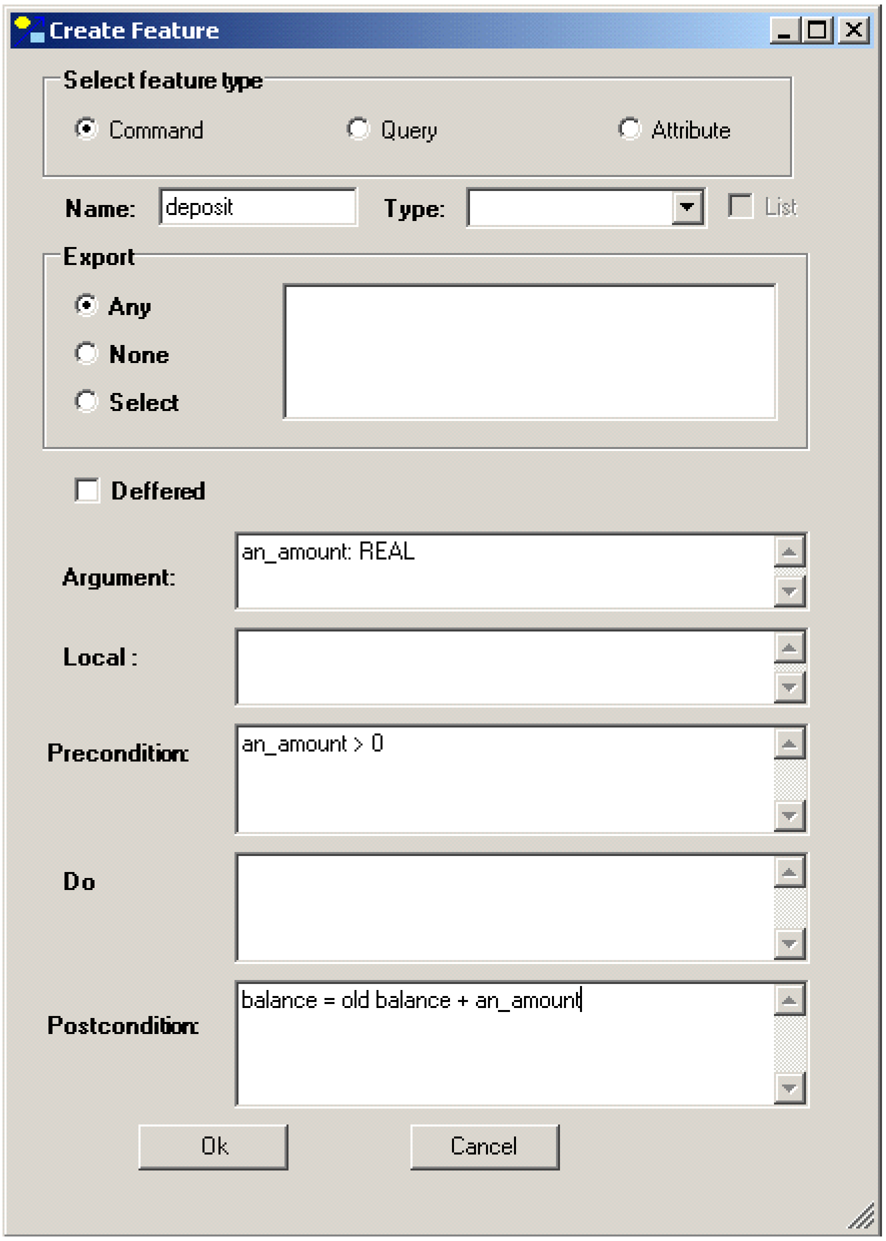

BCCT - Project simple.bp - [static*]
File Edit Tool View Window Message Help
simple
Static Diagram
static
Dynamic Diagram
dynamic
Use Case Diagram
A
B
C
D
C:\Documents and Settings\ygao\My Documents\bcct\example\simple/static
100%

Figure 6-3 Refactoring the model

Suppose we now insert a client-supplier relationship between C and D as in Figure 6-3. Then the tool reports a *Pass* for the constraint *messages-invokable*.

If the designer invokes *Tools* -> *Message-feature*, then the tool supplies an error report which states that the message does not have a source and target feature in the appropriate classes. The tool can now be used to add appropriate features. For example, the user may right click on class D, and invoke *add-feature*. The tool allows the designer to add a new feature as shown in Figure 6-4.

If the designer adds a feature (say the command *deposit*) as in Figure 6-3, then the tool still reports that the message feature constraint is not satisfied as in Figure 6-4. However, if the designer clicks on the error report for this message, a new report appears that indicates that the destination feature is present, but that the source feature is missing (Figure 6-5).


Create Feature

Select feature type

[✓ Command] Query Attribute

Name: deposit Type: List

Export

[✓ Any] None Select

Deffered

Argument: an_amount: REAL

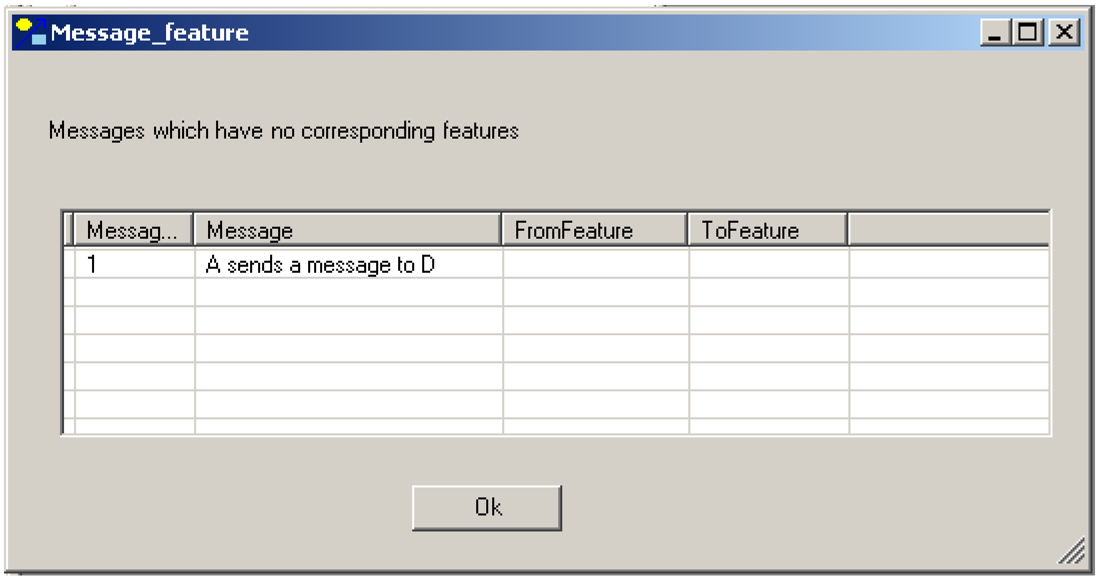
Local :

Precondition: an_amount > 0

Do

Postcondition: balance = old balance + an_amount

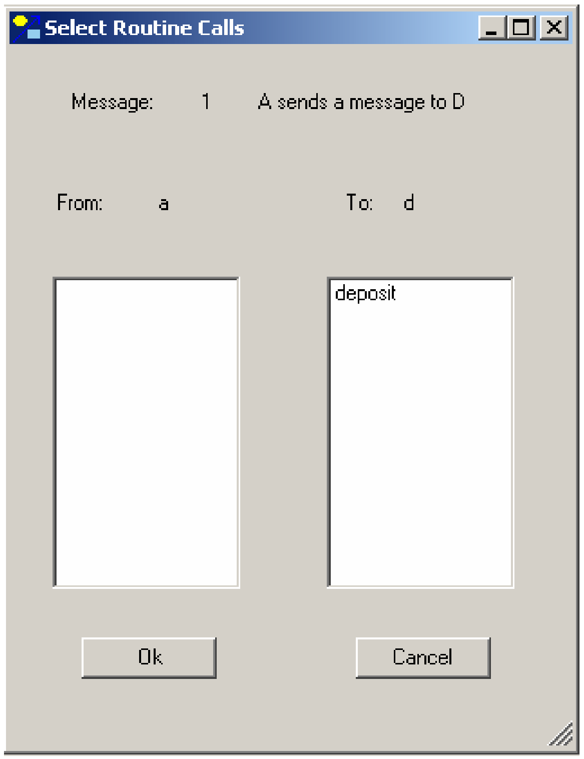
Figure 6-4 Adding a new feature to class C.



Messages which have no corresponding features

Messag...	Message	FromFeature	ToFeature	
1	A sends a message to D			

Figure 6-5 Error report for *message-feature* constraint



Message: 1 A sends a message to D

From: a To: d

deposit

Figure 6-6 Missing source routine in Message-feature error report

We may now add a source routine for the message to class A. In the body of the A routine we must add a call to the deposit feature in class D, i.e. the body must contain a call such as *b.d.deposit*(*200*). The message-feature constraint will still report an error. We need to link the message in the dynamic diagram to the appropriate source and destination routines. This is done with a click as shown in Figure 6-7. The designer selects the appropriate source and target routines, and then clicks *ok* to confirm the selection. All tests will now pass.

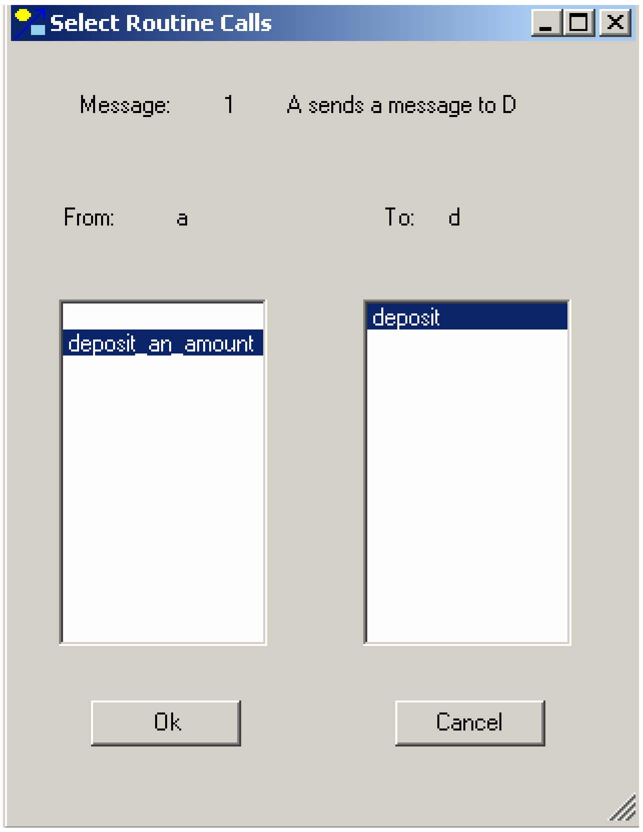


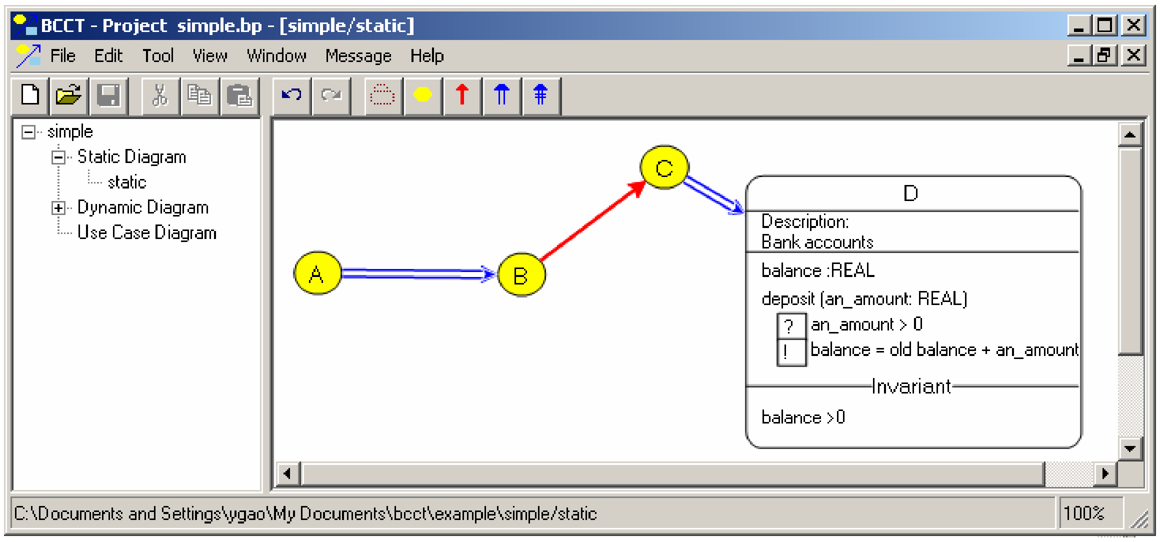
Figure 6-7 Associating a message with source and target features

Once all syntactic consistency constraint checks pass, the designer can generate the code, write the appropriate test, and run the test to check for contractual consistency.

An important property of the BCCT tool is the ease with which models can be constructed and checked for consistency. The mere fact that a test fails is not the end of the story. The tool reports the specific errors and allows the designer to interactively change the model and keeps checking until all the checks pass. By analogy with Test Driven Design [47], we call this property of our tool Consistency Driven Design (CDD). CDD works as follows:

1. Construct some small part of the model;
2. Run the consistency checks (which will usually fail as the model is incomplete or inconsistent);
3. Refactor the model to get the consistency check to pass, and re-run the checks.

CDD works at a higher level of abstraction than Test Driven Design (TDD) because the designer works at the level of the model and not at the lower level of code only. Of course, when checking contractual-consistency, the designer uses tests in the style of TDD. Thus design and coding can be done incrementally with constant feedback and refactoring via tests and checks, by using CDD and DbC at the design level and TDD at the implementation level.


BCCT - Project simple.bp - [simple/static]
File Edit Tool View Window Message Help
simple
Static Diagram
static
Dynamic Diagram
Use Case Diagram
A
B
C
D
100%

D

Description:
Bank accounts

balance :REAL

deposit (an_amount: REAL)

? an_amount > 0

! balance = old balance + an_amount

Invariant

balance >0

C:\Documents and Settings\ygao\My Documents\bcct\example\simple/static

Figure 6-8 The BON static diagram with class *D* expanded to contract view

Contracts are an important part of modelling and design. Thus, BCCT static views also support the ability to expand a class into its contract view as shown in Figure 6-8. These are the contracts that get exercised when the tests are executed.

The BCTT tool is not ready for industrial strength usage. Rather it is a prototype tool that allowed us to explore tool support for multi-view consistency checking and Consistency Driven Design. The tool was implemented in C# using .NET and a diagramming component for .NET called *GoDiagram*. Even though the tool is a prototype, its design and implementation presented many challenges as it had to support the various graphical features of BON as well as the extended BON metamodel defined in Chapter 4. The tool includes the following components:

- *A BON Diagram Editor.* Before consistency can be checked, an editor is needed to draw the static diagrams and dynamic diagrams.

- *A BON Diagram Parser.* This tool parses BON Diagrams to extract the information needed to check consistency. The tool stores BON Diagrams for a project in XML files (persistence).
- *A Consistency Checker.* The tool checks the various consistency constraints using the algorithms in Chapter 4 and Chapter 5.
- *A Code Generator.* The tool translates the model to executable code which facilitates the writing of tests cases using ETester (a unit testing feature which is also used for contract testing).

A detailed description of the BCTT tool is presented in Appendix D.

Chapter 7 Conclusions and Future Work

7.1 Conclusions

In chapter 1, we described how Model Driven Development (MDD) is currently being promoted for addressing the complexities of software development. Model Driven Architecture (MDA) is a flagship initiative of the OMG (Object Management Group) for defining common infrastructures for building UML based MDD tools.

As stated in [55]: “The vision of MDA is both simple and grand. Its objective is to decouple the way that application systems are defined from the technology they run on. The purpose of this decoupling is to ensure that investments made in building systems can be preserved when the underlying technology platforms change”. The Platform Independent Model is a representation of business functionality undistorted by technology details.

As described in chapter 1, the current batch of MDD/MDA tools do not allow the designer to check multi-view consistency between static class diagrams and dynamic collaboration diagrams. Yet it is precisely these two diagrams that are often used by designers.

The main contribution of this thesis was to develop the BCCT tool (the first to our knowledge) that checks multi-view consistency between structural class diagrams and behavioural collaboration diagrams. Such a tool also opens up the possibility of using consistency to drive development as described in the previous chapter (Consistency Driven Development). The following steps were needed to make the tool a reality:

- We formalized the notion *consistency(v1,v2)* of two views *v1* (a static view with contracts) and *v2* (a dynamic view) of a model. This definition of consistency includes notions of syntactic and contractual consistency.
- We developed algorithms to check syntactic consistency, and incorporated these algorithms into BCCT.
- The tool has a graphical editor which is used to construct graphical models, specify features, and contracts down to detailed body code.
- The syntactic consistency checks are run automatically and provide details of where failures occur.
- The model can be automatically translated to executable Eiffel code, and a testdriver is used to check contractual consistency.

Based on this thesis, there are a few opportunities for further research which we describe in the next section.

7.2 Future work

7.2.1 Automatic generation of testdrivers

In Chapter 3 we mentioned that an earlier approach to automated testdriver generation from BON Dynamic Diagrams (BDD) is flawed as outlined in the submessaging problem. A top-level feature message in turn invokes submessages in the BDD. When generating a corresponding testdriver, the only generated routine call associated with a message should be the top call.

This problem should be solvable. Possibly a mark-and-sweep approach could be attempted. Define a singleton linked list or set containing all messages in the dynamic diagram, and then mark each message as code is generated (in order). This should prevent redundant calls and automate testdriver generation and hence more automated support for contractual consistency checking.

7.2.2 Specified Depth Algorithm

The specified depth algorithm (chapter 5) was used to check that for each message in a BDD there is an associated link in the BSD. The algorithm actually generates all possible calls from the target feature to a specified depth, but this full capability is not used in the tool. The tool currently checks if the call actually specified in the feature body is in the generated list.

However, the algorithm could also be used to provide the designer with a list of possible calls in the list, allowing the designer to select which call to use.

7.2.3 BCCT tool

The current BCCT tool is only a prototype to explore consistency checking. As such it only implements part of BON (e.g. support for expanded classes and aggregations). To be a really useful tool, it needs more work so as to support the full range of BON modelling constructs.

Some other areas that could be usefully developed are described below.

- In section 5.3 we described BCCT as effecting strict consistency. It would be useful to enhance BCCT to deal with tolerant consistency in which partial models, that could potentially be made consistent, would pass the checks.
- As mentioned at the beginning of chapter 5, the check for *export* in the *messages-invoked* constraint needs to be tightened up.
- The notion of Consistency Driven Development needs to be explored on big systems, and more refactoring features may need to be included in the tool to make this type of development useful.
- The BCCT tool is complementary to the work being pursued by Ali Taleghani [46], in which contractual consistency is tested with a theorem prover. The two approaches could be usefully merged into a single tool.

Appendix A Code generated for bank example

```
                    -- Automatic generation produced by ISE Eiffel --
indexing
        description: ""

class
        ACCOUNT

create
        make

feature

        make (a_customer: CUSTOMER) is
                require
                        a_customer /= void
                do
                        create  {LINKED_LIST [TRANSACTION]} transactions.make
                        customer := a_customer
                        customer.set_account (Current)
                end

        customer: CUSTOMER

        transactions: LIST [TRANSACTION]

        balance: REAL is
                do
                        from
                                Result := 0
                                transactions.start
                        until
                                transactions.after
                        loop
                                Result := Result + transactions.item.amount
                                transactions.forth
                        end
                end

        set_transactions (a_transactions: LINKED_LIST [TRANSACTION]) is
                require
                        a_transactions_not_void: a_transactions /= void
                do
                        transactions := a_transactions
                ensure
                        transactions_assigned: transactions = a_transactions
                end

end  -- class ACCOUNT
                        -- Generated by ISE Eiffel --
                        -- For more details: http://www.eiffel.com --
                    -- Automatic generation produced by ISE Eiffel --
indexing
    description: "Information about bank customer"

class
    CUSTOMER

create
    make

feature

    make (a_name: STRING) is
        require
            a_name_not_void: a_name /= void
        do
            name := a_name
        ensure
            name_assigned: name = a_name
        end

    account: ACCOUNT

    name: STRING

    set_account (a_account: ACCOUNT) is
        do
            account := a_account
        ensure
            account_assigned: account = a_account
        end

end  -- class CUSTOMER
                    -- Generated by ISE Eiffel --
                    -- For more details: http://www.eiffel.com --
                    -- Automatic generation produced by ISE Eiffel --
indexing
    description: ""

class
    DEPOSIT_TRANSACTION

inherit
    TRANSACTION

create
    make

feature

    make (an_amount: REAL; a: ACCOUNT) is
        require
            an_amount > 0
        do
            amount := an_amount
            a.transactions.extend (Current)
        ensure
            amount = an_amount
            a.balance = old  a.balance + an_amount
        end

end  -- class DEPOSIT_TRANSACTION
                    -- Generated by ISE Eiffel --
                    -- For more details: http://www.eiffel.com --
                    -- Automatic generation produced by ISE Eiffel --
indexing
        description: ""

class
        ROOT_CLASS

create
        make

feature

        a1: ACCOUNT

        c1: CUSTOMER

        make is
                do
                        create  c1.make ("joe")
                        create  a1.make (c1)
                        create  {DEPOSIT_TRANSACTION} t.make (100, a1)
                        c1_balance := c1.account.balance
                        check
                                c1_balance = 100
                        end
                        print ("Balance is: ")
                        print (c1_balance)
                        create  {WITHDRAW_TRANSACTION} t.make (- 100, a1)
                        c1_balance := c1.account.balance
                        check
                                c1_balance = 0
                        end
                        print ("Balance is: ")
                        print (c1_balance)
                end

        c1_balance: REAL

        t: TRANSACTION

end  -- class ROOT_CLASS
                        -- Generated by ISE Eiffel --
                        -- For more details: http://www.eiffel.com --
```

```
                                    -- Automatic generation produced by ISE Eiffel --
indexing
        description: "Interface of the deposit and withdraw transactions"

deferred  class
        TRANSACTION

feature

        amount: REAL

        make (an_amount: REAL; a: ACCOUNT) is
                deferred
                end

end  -- class TRANSACTION
                                    -- Generated by ISE Eiffel --
                                    -- For more details: http://www.eiffel.com --
```

```
                    -- Automatic generation produced by ISE Eiffel --
indexing
    description: ""

class
    WITHDRAW_TRANSACTION

inherit
    TRANSACTION

create
    make

feature

    make (an_amount: REAL; a: ACCOUNT) is
        require
            an_amount < 0 and  a.balance >= - an_amount
        do
            amount := an_amount
            a.transactions.extend (Current)
        ensure
            amount = an_amount
            a.balance = old  a.balance + an_amount
        end

end  -- class WITHDRAW_TRANSACTION
                    -- Generated by ISE Eiffel --
                    -- For more details: http://www.eiffel.com --
```

Appendix B Algorithm for generating test driver from the dynamic diagram

```
class GENERATOR feature
        ...
        generate_test_driver(c:COLLABORATION_DIAGRAM) is
        local
        i: INTEGER;
        m: MESSAGE;
        f: FEATURE;
          do
            generate_driver_header;
            generate_declarations(c.objects);
            generate_check_statement(c.initial);
            from i:=1
              until i>c.messages.length
              loop
                m:=messages.item(i);
                if m.has_guard then
                  generate_ifthen(m.guard)
                end
                if m.has_multiplicity then
                  generate_loop(m.multiplicity)
                end
                if m.feature.is_create then
                  f:=select_create_feature(m.target);
                else
                  f:=select_feature(m.target);
                end
                generate_feature_call(m.target,f);
                generate_close_branches;
                i:=i+1;
            **end**
            generate_check_statement(s.final);
            generate_driver_footer;
          **end**
          ...
        **end** – GENERATOR
```

Appendix C Extended_Model used in [1]

```
class EXTENDED_MODEL inherit MODEL
feature {NONE}
 occurs: SET[OBJECT]
 sequence:SEQUENCE[MESSAGE]
 scenario_box:TEXT
 calls: SEQUENCE[ROUTINE]
feature{ANY}
 class_diagram,collab_diagram:EXTENDED_MODEL
invariant
 msgs_in_rels;calls_linked_to_msgs;
 object_in_occurs; objects_in_abs;
 same_lengths
end – EXTENDED_MODEL
```

Appendix D The BCCT tool

In this appendix, we will provide an overview of BCCT (BON Consistency Checking Tool). The tool is written in C# under .NET and uses a component called *GoDiagram* for coding the diagram editor, and XML for saving the diagrams. The .NET environment was chosen because GoDiagram provides a high-level graphics library which was used to code the BON diagram editor. Thus features such as graphical grouping, undo, moving and scaling are all supported.

1. The tool overview

Figure D-1 shows a screenshot of the tool showing a static diagram and a dynamic diagram. The BON-CASE tool [2,7] is a precursor to this tool, but our metamodel differs as described in Chapter 4, and BON-CASE does not support multi-view consistency. This is the first tool, to our knowledge, that supports multi-view consistency. A complementary tool is being developed by [46].

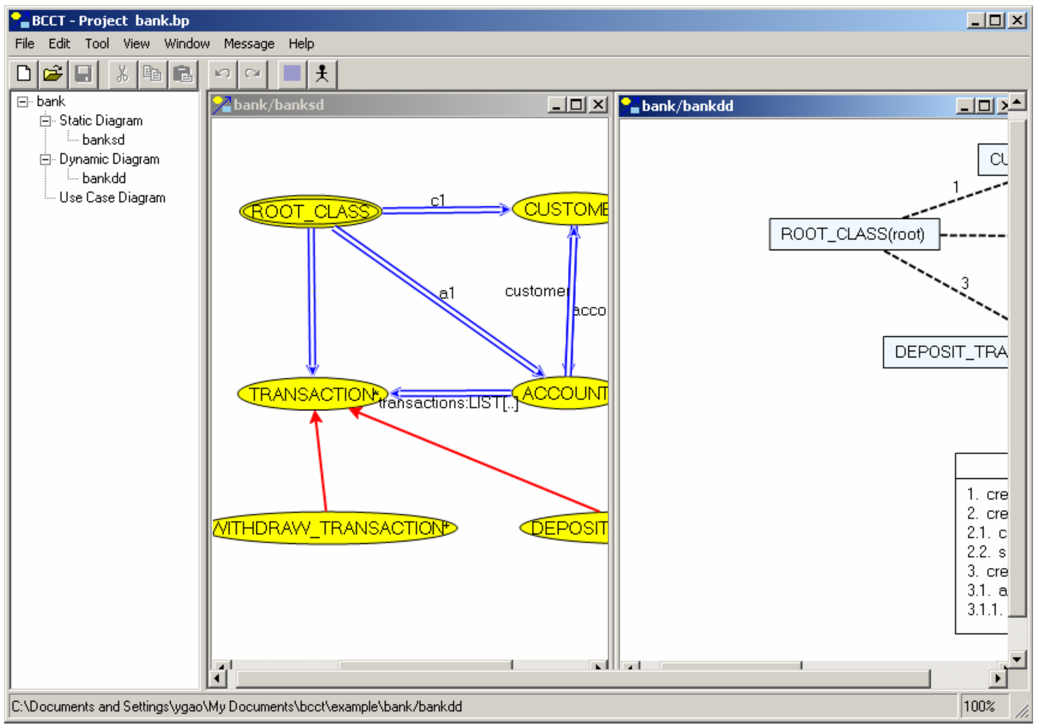


Figure D-1 Overview of the tool

The BON CASE tool [2,7] is an extensible CASE tool for formal specification and reasoning. It has a BON diagram editor and can generate text-based BON and code compatible with the JML verification tool. It does, however, not provide functionality for consistency checking. We had the option of extending this tool or developing a new BON tool. We chose the latter because of the lack of documentation for the BON CASE tool and the friendlier user interface provided by GoDiagram [32]. Also a prototype tool had been developed by an undergraduate student Ali Taleghani using C# and GoDiagram, which was helpful in getting this tool started.

We designed a new icon for this tool. The yellow ellipse represents a class in BON static diagram and the blue square represents an object in a BON dynamic diagrams. The line with an arrow is used in BON to present relationships and messages. In our tool it indicates that we are dealing with relationships between static diagrams and dynamic diagrams, i.e. it deals with consistency checking.

The BCCT tool is developed using C# and GoDiagam under the .Net framework and provides an integrated development environment for the construction of BON diagrams and for consistency checking between static diagrams and dynamic diagrams. The tool provides user-friendly graphical interfaces that let users use the tool without any special training. Undo/redo, zoom in/out, and other features are included as well. It supports Design by Contract, which is not supported by many other tools, especially those for UML. The major functionality, however, is the tool's ability to check consistency between static and dynamic diagrams.

The design of it is extendable, which allows it to be extended to meet other requirements. Below is an outline of the various components of this tool:

- **A BON Diagram Editor.** Before consistency can be checked, an editor is needed to draw the static diagrams and dynamic diagrams.
- **A BON Diagram Parser.** This tool will parse BON Diagrams to extract the information needed to check consistency. Using this parser, we will store the BON Diagrams into XML files and can restore diagrams from the XML files.
- **A Consistency Checker.** This component will provide consistency checking capabilities for BON diagrams. This function is based on algo-

rithms given in Chapter 4 and Chapter 5.

- **A Code Generator**. This will generate the code associated with the diagrams, which the user can use to generate the final test case.

The following sections of this appendix will describe in detail the design of this tool, concentrating on those components we just mentioned. We start with a short overview of GoDiagram which is used throughout the tool.

2. GoDiagram for .NET

GoDiagram for .NET [32] is a product of Northwoods Software. The GoDiagram library, written entirely in C#, is a set of controls and classes built on the .NET platform. It provides a variety of basic graphical objects such as rectangles, ellipses, polygons, text, images, and lines. The user can group these objects together to form more complex objects as a *GoGroup* and can customize their appearances and behaviours by setting properties and overriding methods.

GoDiagram uses a model-view-controller [34] architecture. *GoDocument* serves as the model, i.e. a container providing the abstract representation of the items the user sees in a view. *Goview* serves as the view and the controller – it provides a window displaying objects in a document and it also handles events raised by interactions or other programs.

A GoDocument model provides runtime storage for displayable objects. Adding an object to the document makes it visible in the document's views. Users can organize objects in layers. Class *GoDocument* inherits from *System.Object.GoDocument* and supports one event, *Changed*, to notify observers of changes to the document or to any of its objects.

A *GoView* view provides a window in which the graphical objects stored in a document are shown. A view defines how the user sees the objects and interacts with them. It supports mouse-based object manipulation, including selecting, resizing, moving and copying using drag-and-drop. Each view handles its document's *Changed* event so that it can keep its window up-to-date with all of the objects in the document. The view also supports in-place editing, which can change its corresponding documents, printing, and grids.

The relationship of *GoView* and *GoDocument* is shown in Figure D-2

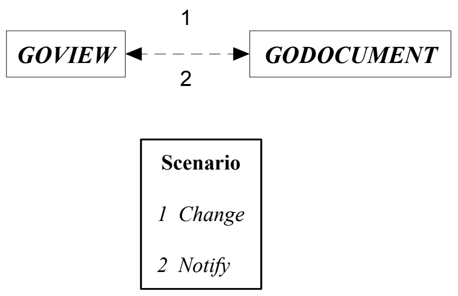


Figure D-2 Relationship between GoView and GoDocumnet

GoDiagram provides support for composing and manipulating graphs, also known as diagrams, where *nodes* have *ports* that are connected by *links*, also known as arcs or edges. GoDiagram provides this functionality with the *GoNode*, *GoPort* and *GoLink* classes. Nodes are groups containing one or more ports. Links are strokes that connect two ports.

The design of the BCCT tool is mainly based on the concepts: Godocument, Go-View, GoNode, GoPort, GoLink, GoGroup. GoDiagram provides other functionalities that we did not mention here, but interested readers can refer to [32] for more details.

3. A BON diagram editor

To check the consistency between a static diagram and a dynamic diagram, we need to draw such diagrams. So, firstly the BCCT tool is a drawing tool for BON diagrams which will then be checked for consistency. In this section, we introduce the GUI and the components from the GUI point of view.

The GUI

As a drawing tool, we designed BCCT's interface similar to the interface of many other drawing programs. Figure D-3 is a screen shot of this tool. The screen is divided into three parts: tool bar, tree view, and drawing area.

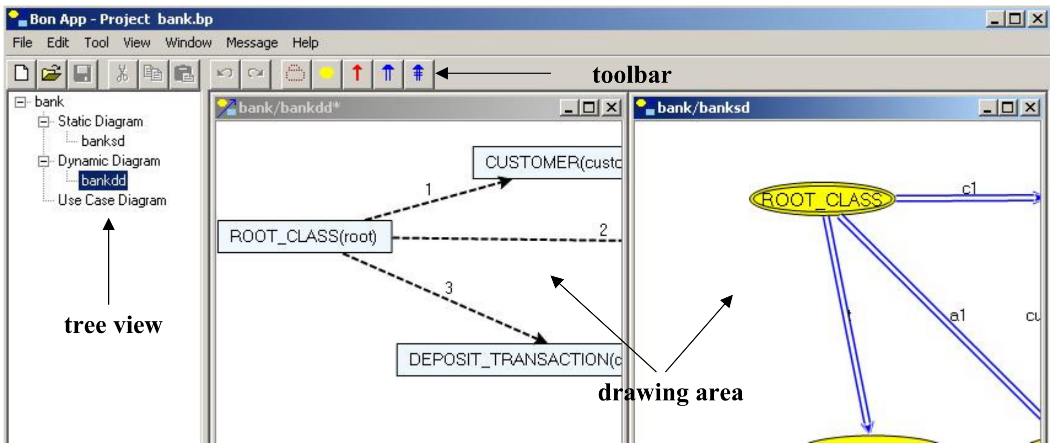


Figure D-3 Diagram editor

The toolbar provides the elements needed to create BON diagrams. The toolbar is for static diagrams and the toolbar is for dynamic diagrams. These toolbars are hidden and shown dynamically depending on which kind of diagram is active. When a user wants to add an element into the diagram, they press the appropriate button on the toolbar (unless the appropriate button has been already pressed); move the mouse to the drawing area; and click or drag to add the element. For example, if users want to add a class to the static diagram, they should first press the button and then click at the position of the drawing area where the class should be added. If this button is already pushed then the user can move the mouse to the drawing area directly and click to add a class.

Adding objects is similar to adding a class. If users want to add a client-supplier relationship to the static diagram, they can click the button and move the mouse to the drawing area and drag from the client class to the supplier class to add a client-supplier relationship. Similarly, the user can add a message link.

The Tree view serves as a navigation tool. Users can use it to open, create diagrams and travel between diagrams. To open a diagram, the user can double click on the diagram's name shown in the tree view. If this diagram has been opened, it will be activated and shows on the top of other windows. To add a new diagram, right click the mouse, choose from the context menu as shown in Figure D-4 and create a new diagram.

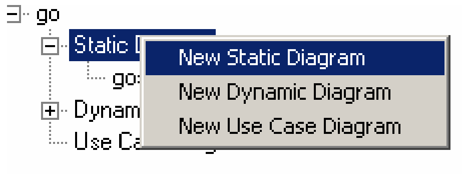


Figure D-4 Context menu of tree view

The drawing area works like a canvas in diagramming tools. Users can draw static diagrams or dynamic diagrams on it. Users can add classes, objects, and other component on it. Users can also select, delete and modify properties of elements on it by using double clicking or context-click for other functions. Moreover, users can save diagrams into XML files or load diagrams saved as XML files by using *File->Save* and *File->open*. The conversion between BON diagrams and XML files will be discussed in section 4.

Diagram elements

As mentioned earlier, the BCCT tool uses GoDiagram and the BON metamodel and algorithms described in Chapter 4 and Chapter 5. In this section, we will give more details about the implementation of the metamodel in our tool.

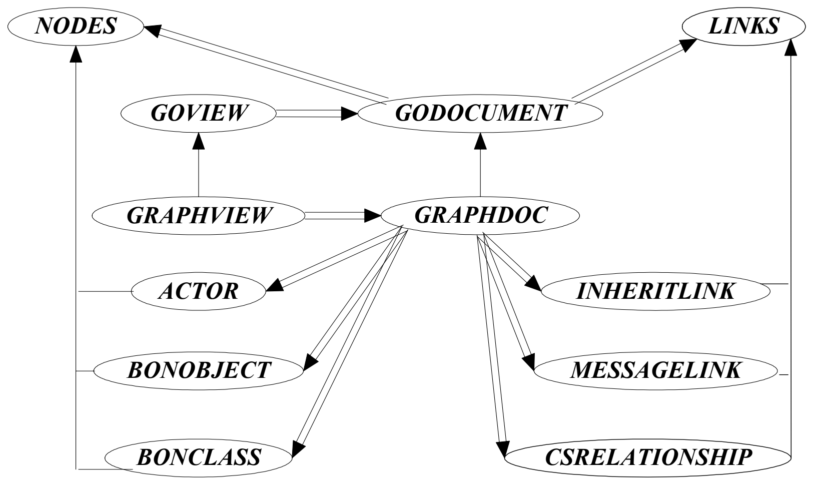


Figure D-5 Elements of BON diagrams editor

GODOCUMENT was extended to *GRAPHDOC* (a new BCCT class) to serve as the model and *GOVIEW* was extended to *GRAPHVIEW* to serve as the view-controller in the editor. The elements of the diagram editor and their relationship are shown in Figure D-5.

From the graph's point of view, a document consists of nodes and links. While from the BON diagram's point of view, a static diagram consists of *BONCLASSes* and relationships between these classes; a dynamic diagram consists of *BONOBJECTs* and messages passed between these objects.

Use Case diagrams are not supported by BON. However, the original intention was to include a Use Case view. The idea was that:

- Requirements are captured by Use Cases; and
- Use Cases are specified by dynamic diagrams; and
- Dynamic Diagrams are implemented by test cases.

However, we did not get as far as actually implementing Use Case views, although the BCCT tool is designed to allow other views.

We do allow actors (a Use Case element) in BON dynamic diagrams as shown in Figure D-6.

We thus add *ACTOR* elements as shown in Figure D-5. This allows us to model external objects in dynamic diagrams that are ignored in consistency checking. Thus diagram elements (*BONCLASS, BONOBJECT, ACTOR*) are based on classes *NODES* provided by GoDiagram and *INHERITLINK, CSRELATIONSHIP, MESSAGELINK* based on the GoDiagram class *LINKS.*

BON diagram elements in Figure D-5 represent the model and the data in a single diagram element. For example, *BONCLASS* represents information about all class features and all other properties related to a BON class. To provide functions to edit the data and properties, we extend *GOVIEW* to *GRAPHVIEW*. *GRAPHVIEW* over-rides some methods provided by *GOVIEW* to edit properties of BON diagram elements, such as *add-features*.

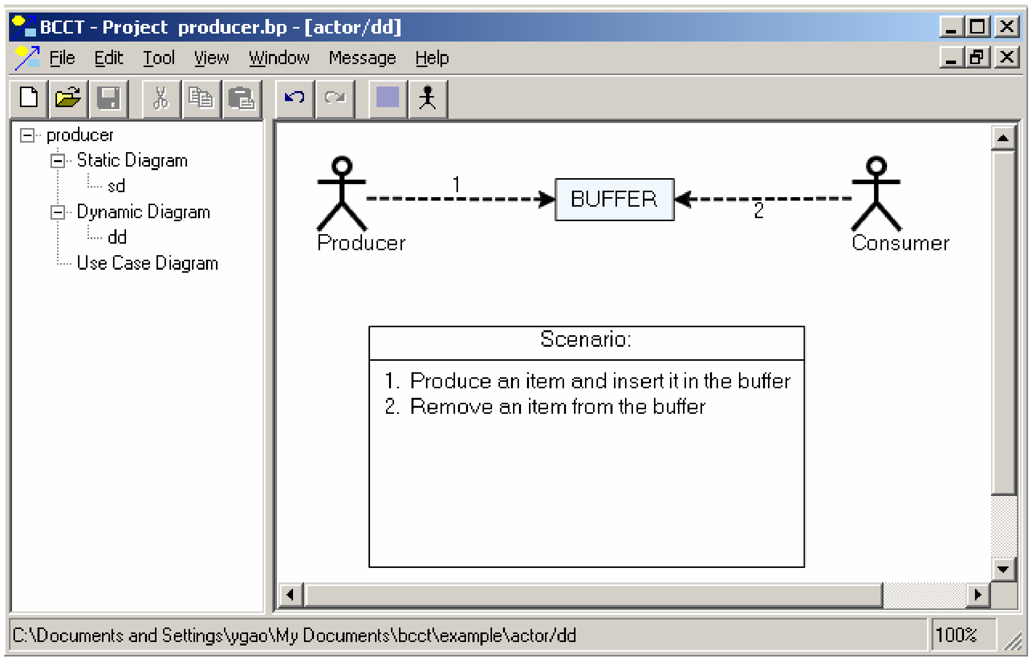

BCCT - Project producer.bp - [actor/dd]
File Edit Tool View Window Message Help
producer
Static Diagram
sd
Dynamic Diagram
dd
Use Case Diagram
Producer
1
BUFFER
2
Consumer
Scenario:
1. Produce an item and insert it in the buffer
2. Remove an item from the buffer
C:\Documents and Settings\ygao\My Documents\bcct\example\actor/dd
100%

Figure D-6 Actors in BON dynamic diagrams

Methods *store* and *load* in class *GRAPHDOC* are used to store BON diagrams to XML files [35] and load XML files to BON diagrams. In the next section, we will depict how we parse a BON Diagram to an XML file and how to parse an XML file to a BON diagram.

4. A BON Diagram Parser

GoDiagram does not have a standard file format that users have to use. The built-in *GoDocument* and *GoObject* classes are serializable. Users can use serialization for short-term persistence and communication using the same version of the GoDiagram library. For long-term persistence to save diagrams, communicate with other applications, serialization is not a good choice. In consistency checking, the diagrams and the metamodel data associated with the diagrams are needed. As a result, we implemented *store* and *load* methods for both *GRAPHDOC* and each BON diagram element. In the *store* method, we will parse a BON diagram or a diagram element (classes, objects, etc.) to an XML file; and in the *load* method, we will parse an XML file to a BON diagram or a diagram element.

XML [35], eXtensible Markup Language, is a markup language much like HTML. XML provides a set of rules for creating semantic tags used to describe data. XML is extendible because its tags are not predefined; users can define their own tags.

An XML *element* is made up of a start tag, an end tag, and data in between. The start and end tags describe the data within the tags, which is considered the value of the element. An element can optionally contain one or more *attributes*. An attribute is a name-value pair separated by an equal sign (=). A basic XML *document* is simply an XML element that can, but might not, include nested XML elements.

We treat a project, a static diagram and a dynamic diagram as an XML document. A project file has an extension "*.bp*", and a static diagram file or a dynamic diagram do not have an extension name. To reload, the user opens a project file. Other views are then automatically reachable from there via the tree view.

The project file contains an element *project* and nests two other elements *static* and *dynamic* that store the metamodel information of diagrams included in this project. Because the main part of this thesis is consistency checking, we did make the XML persistence part as efficient as possible. Other XML technologies such as XPath and XML Schema [36] should be considered for better efficiency.

Consider a project name is *bank* with a static diagram *banksd* and a dynamic diagram *bankdd*. The XML project file *bank.bp* looks as follows:

```
<Project name ="bank">
  <Static>
    <File Name="banksd" />
  </Static>
  <Dynamic>
    <File Name="bankdd" />
  </Dynamic>
</Project>
```

A static or dynamic diagram file is slightly more complicated. The static diagram file has a structure as shown in Figure D-7. The dynamic diagram file has a similar structure like the static one. It has an element *Static* that indicates this is a static diagram and the element *Static* nests other elements: *Class, InheritLink* and *ClientSupplierLink* that are diagram elements of a static diagram. The *Class* element nests *Graph* that stores the graphic information of the class and *Features* that stores what feature the class has.

```
<Static name="…" … …>
  <Class ClassName="…" ID="1" Inherit="…" ……>
    <Graph x="…" … … />
    <Features>
      <Feature Name="make"… … >
      … …
    </Features>
  </Class>
   … …
  <InheritLink from="3" to="4" />
   … …
  <ClientSupplierLink from="1" to="2" Name="…" … …
/>
 … …
</Static>
```

Figure D-7 XML file structure for a static diagram

The *Class* element has an attribute *ID* whose value is unique in the diagram. When the information of links (inherit link, client-supplier link) is stored, the class ID is stored instead of class name. This will avoid inconsistencies that can arise between the diagram and the XML file when class names are changed.

Thus far the parser can only parse information of attributes of diagram elements. The *invariant* of the class, the *precondition*, *postcondition* and *code* information of features is stored as string and do not have any specific XML structure. Future work will consist of finding ways of storing this information more efficiently.

5. A Code generator

As we discussed in section 4.5, in order to check consistency, we need to run Eiffel code generated from the static diagrams to check if the precondition of each feature that is connected to a message is enabled. As already shown, the BCTT tool allows the user to construct models easily. The BCCT tool also allows the designer to enter features, their contracts, feature implementation detail and class invariants. Once sufficient detail is added, the designer can automatically translate the model to executable code. This is analogous to Model-Driven-Development [37] in which the designer constructs models at a high level of abstraction, and the code is automatically generated from the model.

Information editing

Algorithm 4-3 is a sketch for code generation. It generates code from detailed information such as invariants and features. The BCCT tool provides an easy-to-use interface for users to enter this detailed information. As Figure D-8 shows, users can easily edit related information through context menus – to change properties of a class, add features to a class or edit features of a class.

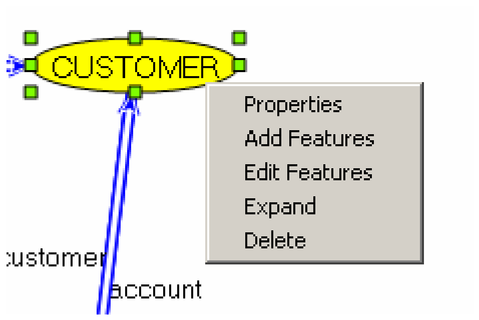
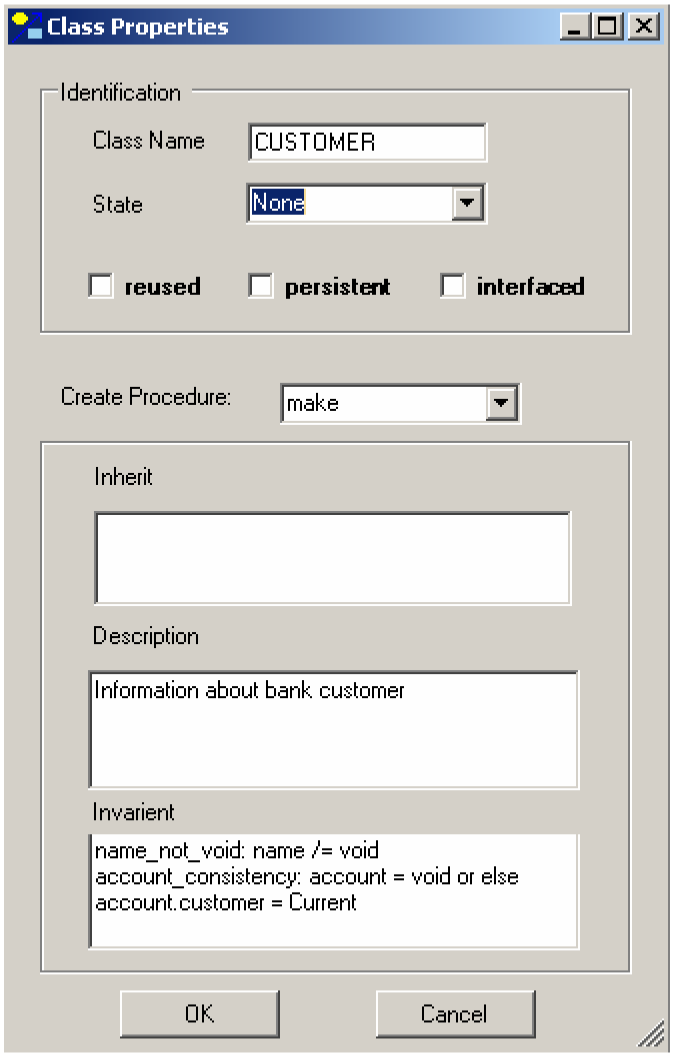


Figure D-8 Context menu of class

Clicking context menu *Properties* can edit properties of a class. Figure D-9 shows the user interface for editing class properties. Users can use it to edit the class name, change the status of the class (root, deferred, effective), create procedures (the default is make), inherit classes. Not all classes that a class inherits from can be presented in the same static diagram where this class appears. Users need the ability to add the information of parent classes other than drawing an inherit link. Also, the user can use *Properties* to enter descriptions and invariants. The user interface of our application was designed similar to that of EiffelStudio. Users familiar with EiffelStudio should be able to use our tool easily.



Class Name: CUSTOMER
State: None
☐ reused ☐ persistent ☐ interfaced
Create Procedure: make
Description: Information about bank customer
Invarient:

```
name_not_void: name /= void
account_consistency: account = void or else
account.customer = Current
```

Figure D-9 User interface for editing class properties

The BCCT tool also provides an easy-to-use interface for adding features. Users can add features by invoking "*Add features*" and can edit existing features by using "*Edit Features*". Once users click on "*Edit Features*", they will be presented with a list of features of the class they are editing as shown in Figure D-10.

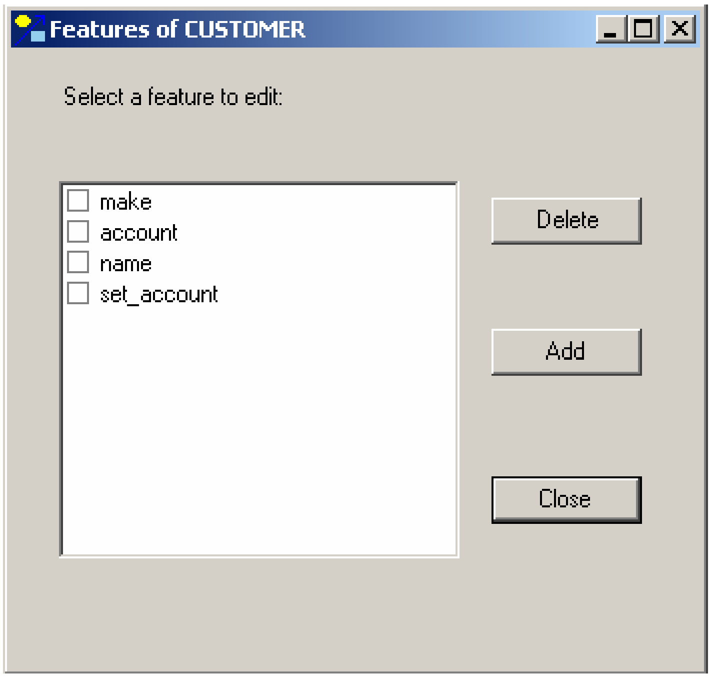
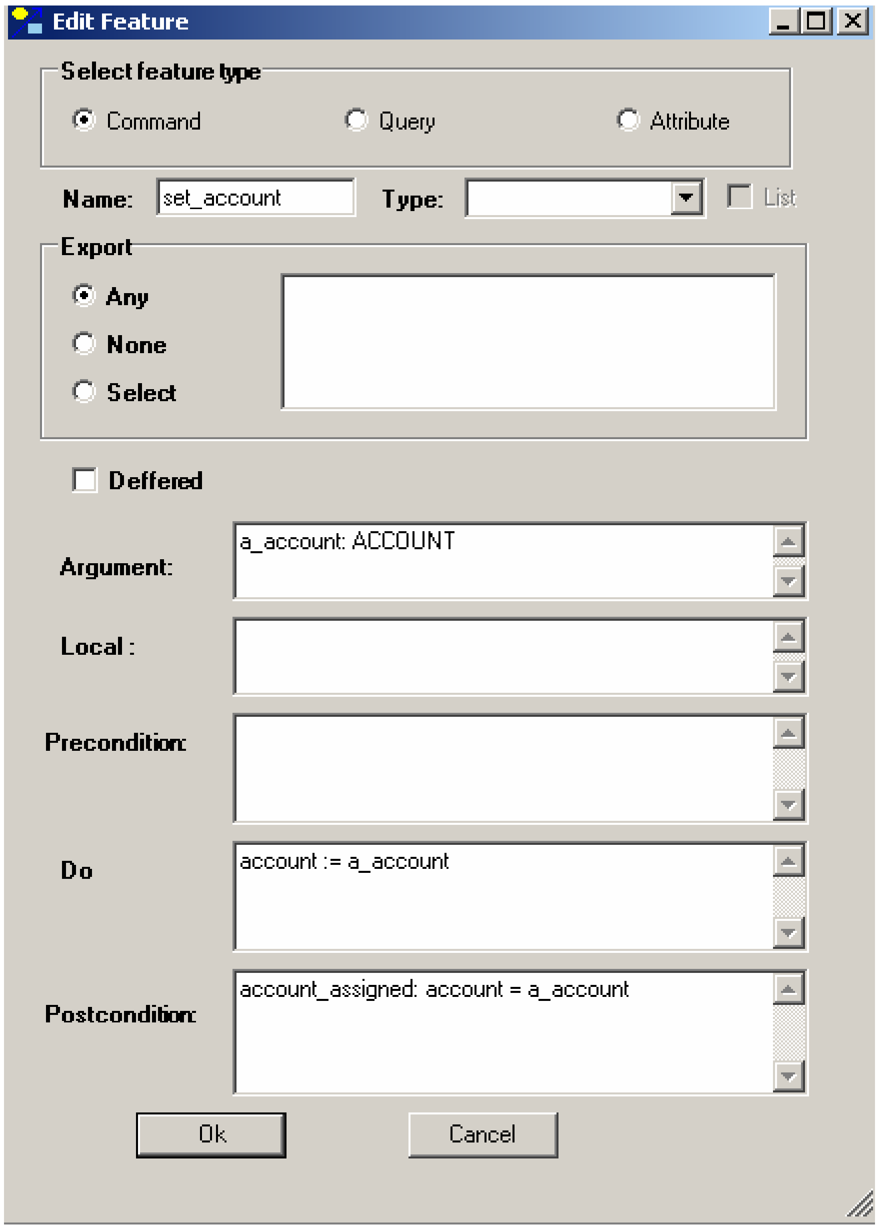


Figure D-10 View of features

Users can choose one or more features to delete; add a feature; or double click to edit selected features as shown in Figure D-11. The add feature user interface is very similar to the edit feature user interface. Through this interface, users can:

- Choose the type of the feature (Command, Query, Attribute);
- Choose the export type (None, Any, Selected) – Selected allows the user to enter a set of classes in the accompanying edit box;
- State whether the feature is deferred or not;
- Edit the arguments, local variables, pre-, post- conditions, and the code (Do).



[✓ Command] Query Attribute

Name: set_account

Type:

List

Export

[✓ Any] None Select

Deffered

Argument: a_account: ACCOUNT

Local :

Precondition:

Do account := a_account

Postcondition: account_assigned: account = a_account

Figure D-11 User interface for editing features

Code Generation

The previous section introduced how users can edit information needed to generate code. This section we will introduce how we can get code from information users entered. We refer to Algorithm 4-3 (Chapter 4, page 60) line by line to show how we can generate code from the static diagram.

- *generate_system_code* generates an .ace file needed by EiffelStudio to run a program.
- ***Foreach*** *bonclass* ***in*** *s.cs* does an iteration to generate code for every *bon-class* in the static diagrams.
- *code := creat_new_class_code (bonclass.name)* creates a new *code* using the name of the *bonclass* in the static diagram.
- *bonclass.generate_descriptions(code)* generates the "index description" part of an Eiffel class. Information for this part comes from field *description* of Figure D-9. The code for this part of Figure D-9 will be:

```
indexing
   description:" information about bank customer".
```

- *code.add ("CLASS*", *bonclass.name*) adds a class name to the code, e.g. for class *CUSTOMER*, it will be

```
class
 CUSTOMER
```

- *code.generate_create_procedure(bonclass.create)* creates the *create* part of the code. For class CUSTOMER of Figure D-9, it will be

```
create
 make
```

- *bonclass.generate_inherit_code(code)* will generate code for each class this class inherits from. It adds an inheritance declaration for each ancestor in the diagram as well as the additional inherit clauses in the inherit field of *inherit* in Figure D-9.
- *bonclass.generate_suppliers_code(code)* generates code from the client-supplier links.
- *bonclass.generate_features(code)* generates code for features with the information provided by Figure D-11.
- *bonclass.generate_invariant(code)* creates the code from field *invariant* of Figure D-9.
- *code.write_to_file* writes code generated to a *.e* file.

6. The Consistency Checker

Chapter 4 and Chapter 5 describe the algorithms used in this thesis to do consistency checking. In this section, we will state how this tool will support consistency checking.

In the BCCT tool, all consistency checking work can be done one by one by using menus shown in Figure D-12. Each menu in Figure D-12 corresponds to the four constraints in Chapter 4 and implements algorithms described in Chapter 4 and Chapter 5.

In order to check that a dynamic diagram is consistent with a static diagram, the tool should also provide other functions such as assigning a class to an object and assigning features to messages. We will introduce these in the following subsections.

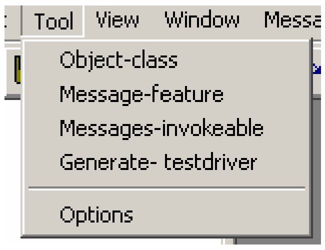
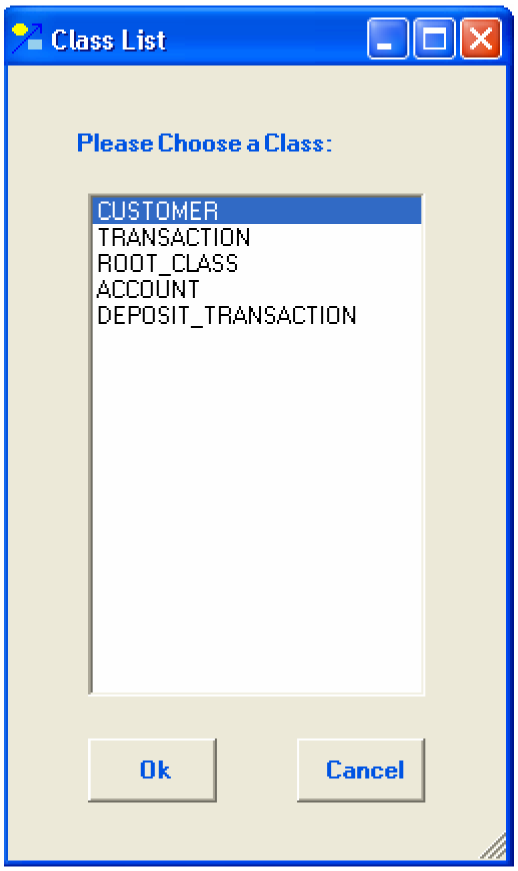


Figure D-12 Menu for consistency checking

Assign a class to an object

As we stated in section 4.1, every object in a dynamic diagram must have a corresponding class in the static diagram. Menu *Tool* → *Object-class* is used to check if each object in a dynamic diagram has a corresponding class by implementing Algorithm 4-1.

To assign an object to a class, there are two approaches that can be used in BCCT tool. One is that users can right click on the object and then choose a *class* from a list of classes in the static diagram as shown in Figure D-13.



CUSTOMER
TRANSACTION
ROOT_CLASS
ACCOUNT
DEPOSIT_TRANSACTION

Figure D-13 Classes List

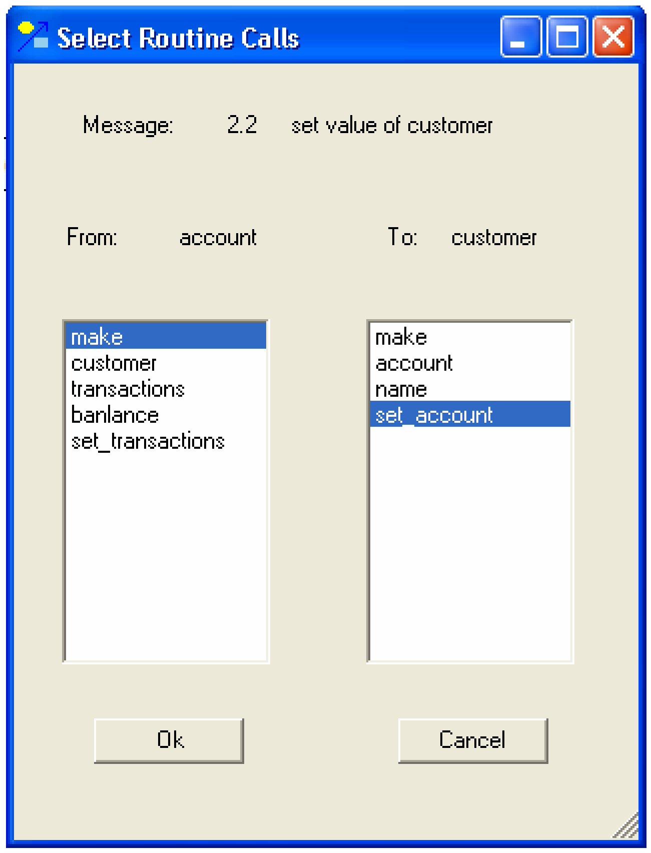
Another approach is that after the check *object_class,* a list of objects which have no corresponding class will be displayed to the user. Double-clicking the name of the object will display a list of available classes.

Assign features to a message

The *message-feature* check requires that each message in the dynamic diagram must have a corresponding feature belonging to the target class to execute this message and there must be a feature in the source class that calls this feature in the static diagram.

So, the BCCT tool should also support assigning a source feature and a target feature to a message.

Like assigning a class to an object, there are also two approaches to assign features. Figure D-14 shows the user-interface to let the user choose features related to a message. The user can choose a source feature (from the list under *from*) and a target feature (from the list under *to*) through this interface to assign *sourcefeature* and *targetfeature* feature to the message.



Message: 2.2 set value of customer

From: account To: customer

make	make
customer	account
transactions	name
banlance	set_account
set_transactions	

Figure D-14 Features List

E-tester

In order to check contractual consistency we generate the code from the model, and allow a user to complete a test that checks the scenario in the dynamic diagram. While user input is required at this stage, most of the infrastructure for the test will already be there from the generated code.

In fact, the test can be fully constructed within the BCCT tool itself by allowing one of the objects in the dynamic diagram to correspond to the *ROOT_CLASS*.

ETester, developed by Dave Makalsky, is a unit-testing framework for Eiffel [15]. It consists of three clusters, which, when added to your system, will allow for easy development of test suites [15].

Because the BCCT tool was developed as a prototype to implement consistency algorithms, we did not concentrate on some aspects such as supporting a full BON notation and intra-diagram constraints. Additional work is necessary to make this tool useful in practice.

References

1. R. F. Paige, J. S. Ostroff, and P. J. Brooke. *Checking the Consistency of Collaboration and class Diagrams using PVS*. In Proc. Fourth Workshop on Rigorous Object-Oriented Methods (ROOM4), London, England, British Computer Society, March 2002.

2. R. F. Paige and L. Kaminskaya. *A Tool Supported Integration of BON and JML,* Technical Report CS-TR-2001-04, Department of Computer Science, York University, July 2001.

3. R. F. Paige and J. S. Ostroff. *Metamodelling and conformance checking with PVS.* In Proc. Fundamental Aspects of Software Engineering 2001, LNCS 2029, Springer-Verlag, April 2001.

4. G. K. Evans. *Model and Source in Sync.* SoftwareDevelopment, June, 2002.

5. L. Briand and Y. Labiche. *A UML-Based Approach to System Testing*. In Proc. UML 2001, LNCS 2185, Springer-Verlag, 2001.

6. A. Tsiolakis. *Semantic Analysis and Consistency Checking of UML Sequence Disgrams*. Diplomarbeit, TU-Berlin, TR 2001-06, April 2001.

7. R.F. Paige, L. Kaminskaya, J.S. Ostroff, and J. Lancaric. *BON-CASE*: *an Extensible CASE Tool for Formal Specification and Reasoning*. In *Proc. TOOLS USA 2002*, Santa Barbara, CA, USA, July 2002.

8. K. Walden and J. Nerson. *Seamless Object-Oriented Software Architecture*. Prentice Hall, 1995.

9. B. Meyer. *Object-Oriented Software Construction (Second Edition)*. Prentice Hall, 1997.

10. W. Liu, S. M. Easterbrook and J. Mylopoulos. *Rule-Based Detection of Inconsistency in UML Models*. Presented at the Workshop on Consistency Problems in UML-Based Software Development, at the Fifth International Conference on the Unified Modeling Language, Dresden, Germany, October 1, 2002.

11. R. Paige, J. S. Ostroff. *The Single Model Principle*. In Journal of Object Tech-

nology, vol.1, no. 5, pages 63-81, *http://www.jot.fm/issues/issue_2002-11/column6*.

12. R. Paige, J. S. Ostroff. *A comparison of BON and UML*. In Proc. UML'99, Lecture Notes in Computer Science, Springer-Verlag.

13. OMG. *Unified Modeling Language Specification: Version 1.5*. March 2003. *http://www.omg.org/docs/formal/03-03-01.pdf*.

14. R. F. Paige, J. S. Ostroff, Phillip J. Brooke. *A Test-Based Agile Approach to Checking the Consistency of Class and Collaboration Diagrams*. UK Software Testing Workshop, University of York, 4-5 September 2003.

15. D. Makalsky. *E-Tester*. *https://sourceforge.net/project/showfiles.php?group_id=73928* (last accessed Jun 23, 03).

16. BON CASE TOOL. *http://www.cs.yorku.ca/~eiffel/bon_case_tool/index.html*.

17. Martin Glinz. *A lightweight Approach to Consistency of Scenarios and Class Models*. Proceedings of the Fourth International Conference on Requirements Engineering, Schaumburg, Illinois, June 10-23, 2000.

18. B. Hnatkowska, Z. Huzar, J. Magott. *Consistency Checking in UML Models.* In J. Zendulka (ed.), Proceedings of the Conference "Information System Modelling", 35-40, Ostrava, 2001.

19. J.L.Sourrouille, G. Caplat. *Constraint Checking in UML Modeling*. Int. Conf. SEKE'02, ACM-SIGSOFT, pp217-224.

20. J.L.Sourrouille, G. Caplat. C*hecking UML Model Consistency*. Presented at the Workshop on Consistency Problems in UML-Based Software Development, at the Fifth International Conference on the Unified Modeling Language, Dresden, Germany, October 1, 2002.

21. C. Gryce, A. Frinkelstein, C. Nentwich. *Lightweight Checking for UML Based Software Development*. Presented at the Workshop on Consistency Problems in UML-Based Software Development, at the Fifth International Conference on the Unified Modeling Language, Dresden, Germany, October 1, 2002.

22. B.Hnatkowska, Z. Huzar, L. Kuzniara, L.Tuzinkiewicz. *A systematic Approach to consistency within UML based software development process*. Presented at the Workshop on Consistency Problems in UML-Based Software Development, at the Fifth International Conference on the Unified Modeling Language, Dresden,

Germany, October 1, 2002.

23. T.H. Cormen, C. E. Leiserson, R.L. Rivest and C. Stein. *Introduction to Algorithms*. 2nd edition, MIT Press & McGraw-Hill, 2001.

24. P. Krishnan. *Consistency Checks for UML*. Proceedings of the 7th Asia-Pacific Software Engineering Conference, 162-169, IEEE, December 2000.

25. J. Derrick and D. Akehurst and E. Boiten. *A framework for UML consistency*. Presented at the Workshop on Consistency Problems in UML-Based Software Development, at the Fifth International Conference on the Unified Modeling Language, Dresden, Germany, October 1, 2002.

26. P. Andre, A. Romanczuk and J. Royer. *Checking the Consistency of UML Class Diagrams Using Larch Prover.* Draft Proceeding of Third Workshop on Rigorous Object Oriented Methods. ROOM'2000, York, UK, January 2000.

27. L.Jacobson, G.Booch, J.Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

28. P. Kruchten. *The Rational Unified Process — An Introduction*. Addison Wesley Longman Icnc., 1999.

29. K. Walden, E. Data, Sweden. *Business Object Notation(BON).* Published as chapter 10 in "Handbook of Object Technology", CRC Press, 1998.

30. K. Walden. *BON Software Architecture Illustrator(BONsai) User's guide version 1.4*, available at *http://www.bon-method.com/index_normal.htm*, August 2003 .

31. S. Owre, N. Shankar, J.Rushby, and D.Stringer-Calvert. *PVS Systerm Guide 2.4*. CSL, SRI International, November 2001.

32. Northwoods Software Corporation. *GoDiagram for .NET Interactive Diagram Classes, User Guide*. January 2003.

33. A. Egyed. *Scalable Consistency Checking between Diagrams – the ViewIntegra Approach*. Published in the Proceeding of the 16th IEEE International Conference on Automated Software Engineering(ASE), San Diego, USA, November 2001, pp. forthcoming .

34. E. Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, Reading, Mass., 1995.

35. T. Bray, J. Paoli. *Extensible Markup Language (XML) 1.1*. C. M. Sperberg-McQueen, Eve Maler, and John Cowan eds.
Available at *http://www.w3.org/TR/2003/PR-xml11-20031105/.*

36. http://www.w3.org/

37. B. Selic. *The Pragmatics of Model-Driven Development*. IEEE Software, Sep/Oct 2003.

38. A. Finkelstein, D.Gabbay, A.Hunter, J.Kramer and B.Nuseibeh. *Inconsistency Handling Multi-Perspective Specification.* IEEE trans. Software Engineering 20(8), August 1994.

39. P.Zave and M.Jackson. *Conjunctions as Composition*. ACM Transactions on Software Engineering and Methodology 2(4), October 1993.

40. Jon Whittle, Johann Schumann. *Generating Statechart Designs From Scenarios*. International Conference on Software Engineering, 2000.

41. ITU Recommendation, X.901-904 – ISO/IEC 10746 1-4. *Open Distributed Processing. –Reference Model – Parts 1-4,* July 1995.

42. A. Finkelstein, J. Kramer, B.Nuseibeh, L. Finkelstein, M. Goedicke. V*iewpoints: A Framework for Integrating Multiple Perspectives in System Development*. In International Journal of Software Engineering and Knowledge Engineering 2(1):31-38, March 1992, World Scientific Publishing Co.

43. S. Easterbrook, B. Nuseibeh. *Managing Inconsistencies in an Evolving Specification*. Second IEEE International Symposium on Requirements Engineering, 1995.

44. R. Paige and J. Ostroff. *Precise and Formal metamodelling with the Business Object Notation and PVS*. Technical Report CS-2000-03, York University, August 2000.

45. E. C. R. Hehner. *A Practical Theory of Programming.* Springer-Verlag, New York, 1993.

46. A. Taleghani. *Contractual Consistency between BON Static and dynamic diagrams*. Master thesis, to be submitted 2004.

47. K. Beck. *Test Driven Development - By Example*. Addison-Wesley, 2003.

48. K.Beck. *Extreme Programming Explained.* Addison-Wesley, 1999.

49. J.H. Johnson. *Micro Projects Cause Constant Change*. 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering. Cagliari, Italy, 2001. *http://www.agilealliance.com/articles/ articles/Chapter30-Johnson.pdf*

50. S.J.Mellor, M.J.Balcer. *Executable UML – A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

51. T. Clark, A.Evans, S.Kent. *The metamodelling language calculus: foundation semantics for UML.* In Proc. Fundamental Aspects of Software Engineering, LNCS Vol.2029. Springer-Verlag, 2001.

52. UML 2.0 Infrastructure Specification, document on *http://www.omg.org.*

53. P. Bhaduri, T. Venkatesh. *Formal consistency of models in multi-view modelling.* In workshop on consistency problems in UML-bases Software Development. 2002.

54. A. Kleppe, J. Warmer, W. Bast. *MDA Explained the model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

55. A. Mcneile MDA: The vision with the hole? *http://www.metamaxim.com/download/documents/MDAv1.pdf.*