Contractual Consistency of BON Static and Dynamic Diagrams

Ali Taleghani, Jonathan Ostroff
ataleghani@uwaterloo.ca, jonathan@cs.yorku.ca

October 10, 2004

Abstract

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

1 Introduction

Today, computers and computer programs infiltrate our lives as never before. As the amount and scope of usage increase so does the pressure on software vendors to produce high-quality reliable software. Different development techniques have been suggested to automate many tasks and therefore lift some of the burden from the developer.

One technique that has evolved recently is that of Model-Driven Development (MDD) [11]. In MDD, models are the primary artifact of development and are kept and maintained throughout the development process. Models are executable and the developer works at the model level rather than at the code level [17]. It is argued that using models for development raises the abstraction level considerably and as a result, simplifies the task of software development for programmers [2].

Modelling in Model-Driven Development is achieved through the use of a modelling language such as the Unified Modeling Language (UML) [3, 13]. UML 2.0 offers nine different views that can be used by the developer to demonstrate various aspects about a software system. In particular, state machines, class diagrams and collaboration diagrams can be used to model an object-oriented system in UML.

The use of various views in modelling introduces the problem of consistency as information in different views can be contradictory. In order to achieve the full potential of MDD, several requirements, including model consistency, have to be guaranteed [17, 18]. In this work we will concentrate on the requirement of *model consistency* and as will be evident, we will introduce a theoretical approach that uses *symbolic model execution* to check for *contractual consistency*.

1.1 Multi-view Consistency

As mentioned above, the consistency between separately constructed views of a software system is one of the requirements of MDD for the construction of error free software. We can divide model consistency into *single-view* and *multi-view* consistency.

Single-view consistency refers to the constraints that must hold within a single view. A single-view constraint for a class diagram, for example, could be to disallow classes with the same name. Many modelling tools support single-view consistency.

Multi-view consistency refers to the satisfaction of constraints between two or more different views of a modelled system. In general, views of a system have overlapping information and we have to ensure that this information is not contradictory in order to preserve consistency. Several different conditions can exist between two views that can cause inconsistencies. In this work we concentrate on the kind of inconsistency that can arise if the contracts in a class diagram are not satisfied by the sequence of routines being executed. The next section defines this concept more precisely.

1.2 Contractual Consistency

The purpose of this work is to develop a method for checking the consistency of (static) class diagrams and (dynamic) object collaboration diagrams. In particular, we will concentrate on a form of consistency that we will refer to as *contractual consistency*.

The behavior of a system is modelled through the use of contracts. These contracts specify operations via pre- and postconditions and class invariants. In this approach, a class diagram with routine contracts and a collaboration diagram are used as the static and dynamic view respectively. Informally, when checking for *contractual consistency* between a class diagram and a collaboration diagram we have to ensure that all contracts are satisfied each time a message from the collaboration diagram executes. If this condition applies to all messages then we will consider the two diagrams *contractually consistent*.

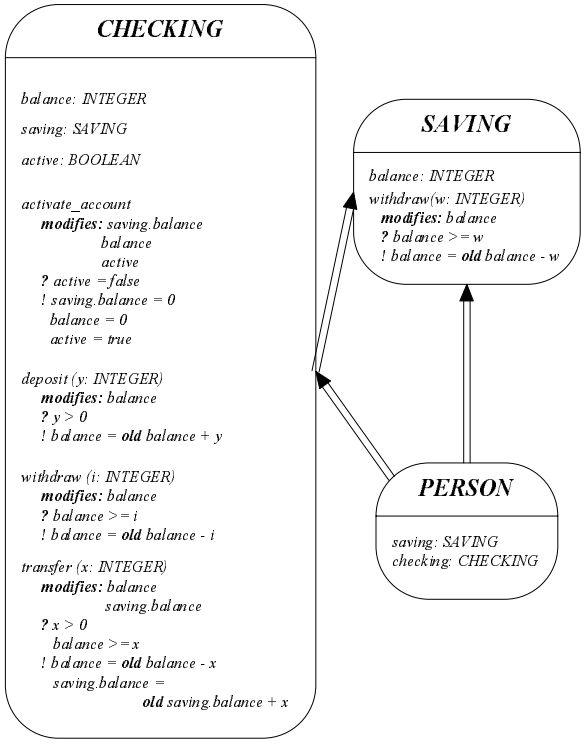

Figure 1: Static diagram of example demonstrating contractual consistency. Three classes are shown and their client supplier relationships.

The following example illustrates the idea behind contractual consistency. Our example uses the modelling language BON [21] rather than UML, since we believe that BON is a simpler modelling language to understand, but at the same time offers most of the features of UML that are needed for our discussion. BON will be discussed in more detail in Section 2.

Figure 1 shows the static class diagram of three classes. Class SAVING and CHECKING are shown in their expanded form to show all their features. Class PERSON is shown in its compact form since its features are of no interest at the moment. In this example, we concentrate only on the classes PERSON and CHECKING. Class PERSON has a client-supplier relationship with CHECKING, which is indicated by the double-arrow.

Class CHECKING has several features, including *balance*, *active*, *activate_account* and *withdraw*. Routines *activate_acount* and *withdraw* have pre and postconditions. The precondition of *activate_account* asserts that *active* is false and one of its postconditions asserts that *balance* is set to zero. Routine *withdraw*'s precondition asserts that *balance* is greater or equal to the amount i that is being withdrawn and its postcondition specifies that after the execution of *withdraw balance* is decremented by i. The expression ***old*** *balance* refers to the state of *balance* before the execution of *withdraw*.

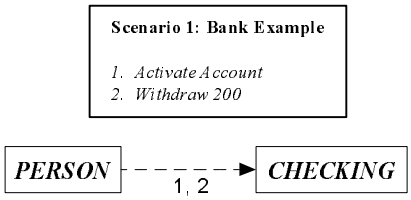

Figure 2: Dynamic diagram of our example. Two objects and two messages are shown. The description of the messages is given in the scenario box.

Figure 2 shows the dynamic view of our model. In this view, we can see two run-time objects corresponding to the classes in Figure 1. The object of type PERSON sends two messages to the object of type CHECKING and those messages are explained in the scenario box. The first message calls *activate_account* and the second calls *withdraw* with argument $200 on the object of type CHECKING.

Lets assume that before the of execution of *activate_account* *active* has the value *false*. As a result, the routine *activate_account* can be executed since its precondition is met. The postcondition of it asserts that *balance* is set to 0. When executing the second message, however, the precondition of *withdraw* asserts that *balance* ≥ 200 and as a result, the precondition of *withdraw* is not satisfied. We therefore get a contract violation. Since not all messages can be successfully executed, we conclude that the two

views are *contractually inconsistent*. Though small, this example demonstrates the idea behind *contractual consistency*.

This work is organized as follows: Section 2 introduces the modelling language BON. Section 3 presents the major theoretical work on *contractual consistency*. Section 4 presents our prototype BON Development Tool (BDT) that implements the ideas presented in this thesis. An early description of this tool was introduced in [20]. Section 5 concludes this work.

2 Business Object Notation - BON

In order to present our method for checking contractual consistency, we had to choose a modelling language. We had the choice between UML and Business Object Notation (BON) [21]. Both languages support static class diagrams and dynamic collaboration diagrams - the two views we are interested in. In addition, both allow for the integration of contracts within routines and classes. We chose BON over UML because of the following two reasons:

- There is no standard uncluttered way to represent OCL contracts in UML class diagrams. They might reside in notes or in other text descriptions. Thus one cannot understand the contract view in one glance. By contrast, BON has a simple, uncluttered and standard way for representing contracts in the class diagrams themselves. [14] offers an extensive comparison between BON and UML.

- None of the current UML tools supports automatic translation of OCL to code. For example, contracts in Java could be described with JML [6]. However, there is an impedance mismatch between OCL and these other specification languages (e.g., see the mapping from OCL to JML provided in [1]). Translating BON class diagrams with their contracts to Eiffel [12] code is seamless.

2.1 Static Diagrams

The fundamental construct in a BON static diagram is a class. A class has a name, an optional class invariant and zero or more features (features can be attributes, queries or commands). Classes can be viewed in two forms: In the compact view a class is represented by an ellipse with the class name in the center of the ellipse. The second view consists of a rounded rectangle with all features and invariants of the class visible. Figure 1 shows both views. Classes can be further organized in clusters, which make construction of larger systems easier.

On top of specifying the signature of the feature, pre and postconditions can be specified for commands and queries. Figure 1 shows an examples of this. In graphical BON, "**?**" is the graphical notation for preconditions and "**!**" the notation for postconditions.

Classes and clusters interact via two kinds of relationships:

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

2. **Client-supplier**: The user has two options for a client-supplier relationship, association and aggregation. Association refers to reference relationships and aggregation to sub object (subpart) relationships. Association relationships are represented by a double line with an arrow towards the supplier.The representation of aggregation is similar to that of association, but with an additional perpendicular line close to the arrow.

2.2 Dynamic Diagrams

BON also provides notation for collaboration diagrams, which show the communication between objects. Collaboration diagrams consist of rectangles representing runtime objects and arrow lines between them representing messages sent from one object to other object(s). An example of a BON collaboration diagram is given in Figure 2. Messages are numbered for two purposes: First, they represent time in the scenario - that is the order in which calls are made. Second, they correspond to entries in a scenario box where the role of each call may be described using free text. In [21], it is understood that a message in a dynamic diagram corresponds to *some* routine in the static diagram. It is not necessary to specify the precise routine in the initial design phase. A textual description in the scenario box suffices. But, this is the critical point at which the static and dynamic

views overlap (i.e. the correspondence between messages in the dynamic diagram with feature calls in the static diagram). In an actual BON consistency tool such as BDT, the mapping between messages and features will need to be more precise in order to be able to check for *contractual consistency*.

3 Contractual Consistency

The purpose of this work is to precisely define *contractual consistency* between a BON static diagram and a dynamic diagram and introduce a theoretical approach for checking this kind of consistency. In this section we will present the theoretical work necessary to define this concept. This work is in addition to that presented in [16]. We will define the concept more precisely and present the BON Development Tool (BDT) in the next section. It will be evident that checking for *contractual consistency* is closely related to *symbolic model execution*.

Informally, when checking for *contractual consistency* between a static and dynamic diagram in BON, we check whether the precondition of a routine associated with a message in the dynamic diagram is satisfied before the message is executed. In addition, the postcondition of the routine should not evaluate to false. If these conditions apply to all messages in the dynamic diagram, then we consider the static and dynamic diagram *contractually consistent*.

The procedure for checking for contractual consistency will be reduced to checking the validity of a predicate. If we had access to an ideal oracle that could return *valid* precisely when the predicate is a theorem, and *invalid* otherwise, our procedure would be sound and complete. However, in the real world, we must pass the predicate to a theorem prover. The theorem prover is sound but not complete. Hence, our tool will be incomplete. We will discuss the sources for incompleteness in Section 4.1.2.

3.1 The BON Model

We will check *contractual consistency* between a BON static diagram and dynamic diagram. In this work we concentrate on contractual consistency only and therefore assume other forms of consistency already exist between the two views. Thus, the static diagram must satisfy the following conditions:

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

The following conditions must be satisfied by the dynamic diagram:

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

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

3.2 Example

In this section, we will continue with the example introduced above. We will, however, add a new dynamic diagram in order to demonstrate a more advanced scenario. In this example we will model a bank that provides customers with checking and saving accounts. In the following two subsections, we will introduce the static and dynamic structure of this example.

3.2.1 Static Structure

Figure 1 shows the classes PERSON, CHECKING and SAVING. The class PERSON has two attributes *checking* and *saving* of type CHECKING and SAVING respectively. Class SAVING has an attribute *balance* of type INTEGER. Further, this class has one routine *withdraw(w: INTEGER)*. The precondition of *withdraw* states that when calling this routine, *balance* must be greater or equal to w. The postcondition ensures that *balance* is decreased by w after completing execution. In addition, the modifies

clause (Δ) is specified for this class. The modifies clause identifies the attributes that this routine can modify. The meaning of the modifies clause and its importance in this discussion are explained below.

Figure 1 also shows the detailed view of class CHECKING. This class has three attributes: *balance* of type INTEGER, *saving* of type SAVING and *active* of type BOOLEAN. It has four commands that change the state of the object. The routines and their contracts are as follows:

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

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

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

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

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

At the specification level, we need a precise account of which variables change and which stay the same as will be seen later in this section in equation (1). We could try to infer the changes by analyzing the postcondition, but this is a risky business. Instead, we require that the designer specifies this clearly and precisely up front in the *modifies* clause. The modifies clause will thus list changes to variables.

3.2.2 Dynamic Structure

Figure 3 shows the dynamic diagram of our bank example. There are three runtime objects in the system: *checking* of type CHECKING, *saving* of type SAVING and $p1$ of type PERSON. In this example, four messages are sent between these objects as described in the scenario box in Figure 3. Table 1 lists

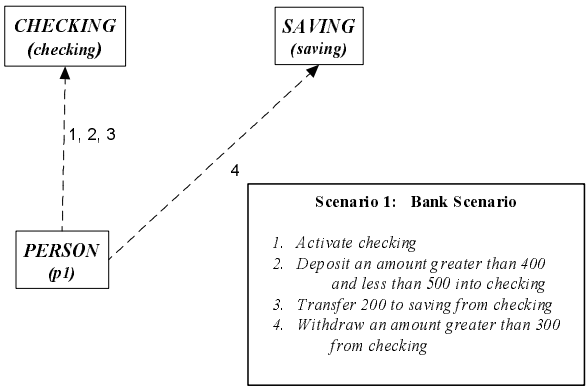

Figure 3: Dynamic diagram for Bank scenario. There are three objects and four messages. The scenario box describes the messages in more detail.

Message	Routine	$args_i$
m_1	*checking.activate_account*	$args_1 = true$
m_2	*checking.deposit(y)*	$args_2 = (y > 400 \wedge y < 500)$
m_3	*checking.transfer(x)*	$args_3 = (x = 200)$
m_4	*checking.withdraw(i)*	$args_4 = (i > 300)$

Table 1: Table showing mapping between messages in DD and associated routines in SD. arg_i is the constraint on the arguments of routine r_i.

the mapping between the messages in the dynamic diagram and routines in the static diagram.

Given this $model(SD, DD)$, we would like to know whether it is *contractually consistent*. In order to simplify the task of writing object and feature names, we will use the following abbreviations for the remainder of this section: *ch* for *checking*, *b* for *balance*, *a* for *active* and *s* for *saving*.

3.3 System State Constraint

In the discussion of consistency, the notion of a *system state* is of importance. The system state (SS) is defined by all the entities and their values in all runtime objects at a certain moment during execution. However, we are not dealing with implementations in which the state is fully defined. All we have are contracts (single state, and double-state predicates in the variables). Thus, instead what we might have at an instance of time is the System State Constraint SSC:

$$SSC \stackrel{def}{=} a_1 = 5 \wedge a_2 > a_1 \wedge a_3 = Void$$

where $SS \Rightarrow SSC$.

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

arguments of these routines. This predicate is a constraint on the state at a certain instance of time.

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

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

where $1 \leq i \leq n$ where n is the number of messages in the DD. The initial state constraint is SSC_0, which represents the system state constraint before the execution of any messages. This state is either the predicate *true* (meaning that the attributes may have any value) or is user specified.

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

$$SSC_0 \ = \ (ch.b > 400 \wedge ch.a = false)$$

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

3.4 Post SSC

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

Routine r_i which is associated with message m_i has a precondition $r_i.pre$, a postcondition $r_i.post$ and a modifies set $r_i.mod$. Let $r_i.mod = \{v_1, \cdots, v_m\}$. Then

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

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

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

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

2. The postcondition of r_i is used to append the constraints on the variables that have changed;

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

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

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

$$\begin{aligned} SSC_i \ = \ & (SSC_{i-1}[v_1 := v_1', \cdots, v_m := v_m'] \\ & \wedge \ r_i.post[\mathbf{old}\, v_1 := v_1', \cdots, \mathbf{old}\, v_m := v_m']) \end{aligned} \tag{2}$$

(2) is stronger than (1), but retains information regarding past states of the variables. Our tool BDT implements (2) in order to provide more valuable feedback to the user.

We can now use (1) to determine the post state for our bank example. Assume we started with SSC_0 as described above and executed m_1. As a result, SSC_1 has to be determined.

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

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

The modifies clause of r_1 is as follows:

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

As a result, we know that the attributes in $r_1.mod$ are modified and have to be renamed in the construction of SSC_1. Using equation 1 we obtain the following:

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

Evaluating this expression, we obtain the following SSC_1:

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

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

3.5 Prover

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

$$Prover(p) = \begin{cases} T & \text{if } \models p \\ C & \text{if } p \text{ is contingent} \\ F & \text{if } p \text{ is a contradiction} \end{cases} \quad (3)$$

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

3.6 Symbolic Execution Step

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

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

$$\begin{aligned} &step(SSC_{i-1},\ r_i,\ SSC_i) \text{ iff} \quad (4) \\ &Prover((SSC_{i-1} \wedge args_i) \Rightarrow r_i.pre) = T \wedge \\ &Prover(SSC_i) \neq F \end{aligned}$$

which defines under what conditions the symbolic execution of the original message $o_k \xrightarrow{m_i} o_{k+1}$ in the DD is successful. $args_i$ is the constraint on the formal arguments of routine r_i (see Table 1). Note that we constructed (4) from information contained in both DD and SD (e.g. the contracts come from the SD).

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

Continuing with our bank example, we have to complete a step that we omitted above. We have to check whether the execution of m_1 results in a successful step using (4).

We have the following values:

$SSC_0 = (ch.b > 400 \wedge ch.a = false)$
$r_1.pre = (ch.a = false)$
$SSC_1 = (ch.b = 0 \wedge ch.a = true)$

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

$$\begin{aligned} &step(SSC_0, r_1, SSC_1) = \\ &Prover((SSC_0 \wedge args_1) \Rightarrow r_1.pre) \wedge \\ &Prover(SSC_1) \end{aligned}$$

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

$Prover(SSC_0 \Rightarrow r_1.pre) = T$
$Prover(SSC_1) \neq F$

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

3.7 Contractual Consistency of model(SD, DD)

In the section above we defined the meaning of a single *symbolic execution step*. A step is the symbolic execution of a single message in the dynamic diagram and a successful execution step was defined by (4).

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

$$\begin{aligned} &\text{CC}(model(SD, DD)) \stackrel{def}{=} \quad (5) \\ &\forall i \mid 1 \leq i \leq n \bullet step\ (SSC_{i-1},\ r_i,\ SSC_i) \end{aligned}$$

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

In order to check for contractual consistency for our bank example, we have to symbolically execute the remaining three messages and if all of them can

be successfully executed then we can claim that our $model(SD, DD)$ is contractually consistent.

In order to check whether message m_2 can be successfully executed, we have to check $step(SSC_1, r_2, SSC_2)$, which translates into the following two conditions (r_2 corresponds to the routine *deposit(y)* in Table 1):

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

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

We have the following values:

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

$r_2.pre = (y > 0)$

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

$SSC_1 \wedge args_2 = (ch.b = 0 \wedge ch.a = true \wedge$
$s.b = 0 \wedge y > 400 \wedge y < 500)$

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

$SSC_2 = (ch.a = true \wedge ch.b = 0 + y \wedge$
$s.b = 0 \wedge y > 400 \wedge y < 500)$

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

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

$SSC_3 = (ch.a = true \wedge ch.b = y - 200 \wedge$
$s.b = 200 \wedge y > 400 \wedge y < 500)$

The last message to be symbolically executed is m_4. This message tries to withdraw an amount that is greater than 300 from s (i.e. $args_5 = (i > 300)$). We have to therefore check the following two conditions as described in (4):

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

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

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

$SSC_3 \wedge args_4 = (ch.a = true \wedge ch.b = y - 200 \wedge$
$s.b = 200 \wedge y > 400 \wedge y < 500 \wedge i > 300)$

The value for $r_3.pre$ is

$r_3.pre = ch.b \geq i$

Checking condition (1), we can see that *Prover* would not return F. The attribute $ch.b$ has a value between 201 and 299 at the time of the execution which is not greater or equal to i, which is greater than 300. This failure signals a *contract violation* and as a result, we conclude that $model(SD, DD)$ is *contractually inconsistent.*

4 BON Development Tool

In this section, we will introduce and discuss the BON Development Tool (BDT). BDT was developed as a prototype to demonstrate the implement-ability of our approach to contractual consistency. BDT is a software modelling environment allowing for the construction of static and dynamic diagrams in BON. In addition, BDT incorporates most of the concepts of consistency checking that were discussed in the previous sections. BDT was developed as an extension (plug-in) for Eclipse [8] in order to simplify the development of a graphical application and at the same time make BDT available in an environment that is widely used. The graphical framework used for the implementation of BDT was GEF [7].

BDT consists of three parts: a static diagramming module, a dynamic diagramming module and a consistency checking module. In this section, we will explain important concepts of the consistency module. The structure of the static and dynamic diagramming modules are described elsewhere [19].

4.1 Consistency Module

The module for checking contractual consistency consists of a small number of classes. We will not discuss the architecture of this module, but refer the reader to [19] for a more detailed discussion on the class structure of the consistency module. In this section we will concentrate on the interaction between this module and the theorem prover, the system state constraint and the incompleteness of BDT.

4.1.1 Theorem Prover - Simplify

In section 3.5, we described the function *Prover* that we used as an oracle for the development of our theoretical approach to contractual consistency checking. The definition of *Prover* was given by (3). In order to implement BDT, we have to replace this oracle by a theorem prover that provides us with similar abilities.

We have decided to use the theorem prover *Simplify* [4] which is the prover used in the Extended Static Checker ESC/Java [9] for the following reasons:

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

- *Simplify* tries to find simple proofs *rapidly* if they exist. This provides an advantage for software developers who are not interested in long waiting times when testing models.
- *Simplify* simplifies the use of the multi-dot notation that is extensively used in object-oriented programming.
- An important property of the theorem-prover is that it is refutation-based: to prove a formula P, it tries to satisfy the negation of P. If the prover finds a set of variable assignments that satisfy the negation of P, those variable assignments are returned as feedback. This feature is very useful as it can be used to give valuable feedback about contract violations.

4.1.2 Incompleteness of BDT

We use *Simplify* in BDT to check for a successful symbolic execution step as defined by (4). *Simplify* is only an approximation to our oracle *Prover* described in Section 3.5. This theorem prover was "engineered to be automatic rather than complete" [5] and as a result this theorem prover is incomplete. Due to this restriction, BDT is also incomplete as it can provide spurious warnings and label a model as contractually inconsistent, but in fact the model could be contractually consistent.

Simplify has two possible outputs given a predicate P:

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

Given a predicate P as the input to *Simplify*, the output *Valid* signals that P is definitely a theorem (assuming that the theorem prover was implemented correctly). An output of *Invalid* signals incompleteness as it cannot be determined whether P is indeed not a theorem. In these cases, *Simplify* tries to find a counter example that would make P evaluate to $false$. There are, however, cases in which the theorem prover does not return a counter example as it did not find one (even though one does exist). Table 2 shows how *Simplify* is used in order to evaluate (4).

If $Simplify(SSC_i \rightarrow r_i.pre)$ returns *Valid* then the precondition of $r_i.pre$ is satisfied and the user is given the appropriate feedback. However, if the theorem prover returns $Invalid$ then it is the user's responsibility to decide based on the context returned whether a contract violation will occur. Similarly, if $Simplify(\neg SSC_{i+1})$ returns $Valid$ then the system state constraint SSC_{i+1} is a contradiction and the user is notified (note that we call $Simplify$ on the negation of SSC_{i+1}). An output of $Invalid$ can be used with the context to determine whether the system state constraint is in fact a contradiction.

	Valid	Invalid
$Simplify(X)$	$Prover(X) = T$. The precondition is enabled under the current state constraint.	*Incomplete.* Context usually provides enough feedback to help the designer.
$Simplify(\neg Y)$	$Prover(Y) = F$. SSC_{i+1} is a contradiction; thus there is no poststate with which to continue the symbolic execution.	*Incomplete.* Context usually provides enough feedback to help the designer.

Table 2: Table explaining the mapping between the output from *Simplify* and *Prover*. $X = SSC_i \rightarrow r_i.pre$ and $Y = SSC_{i+1}$. Note that we use $\neg Y$ for the second condition.

5 Conclusion

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

First, our BDT tool requires improvements and additions to make it an industrial-strength tool. These improvements include the following:

- Support class invariants and sub-messages,
- Support all legal contracts including quantifications and object equality. A limited version of object equality is already supported. Simplify supports quantifications.
- Support multi-dot notations. BDT currently supports single-dot notations only.
- Provide support for genericity and inheriting contracts.

Second, BDT can be incorporated with the Eiffel Development Tool (EDT) [10]. EDT is an integrated development environment (IDE) for software production using Eiffel. EDT has also been developed as a

plug-in for Eclipse and as a result can be integrated with BDT.

Finally, BDT/EDT can be expanded to provide program verification similar to ESC/Java [9]. The Eiffel Refinement Calculus (ERC) [15] could be used to ensure that program code satisfies the specifications of a class.

References

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

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

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

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

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

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

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

[8] ———, *Eclipse*, http://www.eclipse.org, 2004.

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

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

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

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

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

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

[15] ———, *ERC: an Object-Oriented Refinement Calculus for Eiffel*, Formal Aspects of Computing **16** (2004), 51–79.

[16] Richard Paige, Jonathan Ostroff, and Phillip Brooke, *Checking the Consistency of Collaboration and Class Diagrams using PVS*, Fourth Workshop on Rigorous Object-Oriented Methods (London, England), 2002.

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

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

[19] Ali Taleghani, *Contractual Consistency of BON Static and Dynamic Diagrams*, Master's thesis, York University, July 2004.

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

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