Static Verification of UML Model Consistency

A. Baruzzo[1] and M. Comini[1]

Dipartimento di Matematica e Informatica (DIMI), University of Udine,
Via delle Scienze 206, 33100 Udine, Italy.

Abstract In a UML model, different aspects of a system are covered by different types of diagrams and this bears the risk that an overall system specification becomes inconsistent or incomplete. Hence, it is important to provide means to check the consistency and completeness of a UML model.

Many approaches for model validation and verification rely on generation of suitable code which dynamically (i.e., at run-time) checks the validity of OCL constraints. This approach has several well-known drawbacks. For example, it cannot generally guarantee that a constraint will never be violated, unless an infinite number of tests is performed. On the other hand, even more desirable static approaches are not immune from weaknesses. The techniques based on model checking suffer from the state explosion problem and thus cannot scale to most system sizes. Moreover, the static approach is in general undecidable.

In this paper we propose a static verification framework based of Abstract Interpretation techniques, a theory of approximation of mathematical structures. This framework, by using OCL constraints together with Class Diagrams, certifies that the dynamic part (Sequence Diagrams) of the model is satisfied. Our approach keeps the advantages of static verification and, at the same time, avoids the weaknesses of the other mentioned methods as it does not require users to build test scenarios and also it can make undecidable methods decidable, up to a specified level of precision.

1 Introduction

Finding program bugs is a long-standing problem in software construction. There has been considerable theoretical research activities and published results starting from the mid-nineties about using formal specifications to help the debugging phase. All these theoretical efforts have produced also relevant practical results. Indeed, the software engineering community has nowadays accepted the fact that the specification of various kinds of pieces of software is not only a topic of theoretical interest but also one of practical importance. A steadily increasing number of papers dealing with the concepts and the practical use of assertions in general have been published during the last few years (amongst all [28]). The basic foundations have been laid by Bertrand Meyer with his concept of *Design by Contract* (DbC) as realized in the Eiffel language (see [20,19]). This approach has then rapidly spread to other languages, for instance there emerged lot of support for assertions for Java (e.g. Jass[1], the iContract tool[2], *etc.*) and C++ (e.g. C[2] [3], *etc.*). Most importantly, the Unified Modeling Language (UML) has now, as one of its integral parts, the Object Constraint Language (OCL), which has its root in the Syntropy method. With OCL we can naturally implement the contract mechanism of Eiffel.

In a UML model, different aspects of a system are covered by different types of diagrams and this bears the risk that an overall system specification becomes inconsistent or incomplete. Hence, it is important to provide means to check the consistency and completeness of a UML model.

Many approaches for system debugging and for model validation/verification rely on generation of suitable code which dynamically (i.e., at run-time) checks the validity of OCL constraints (i.e., the compliance of the system status w.r.t. the constraint). This approach has several drawbacks. For example, it undoubtedly slows down performance and can potentially alter the behavior (if the inserted code has side effects by mistake). But most of all it does not ensure to reveal a bug unless the specific run of the system effectively enters a state which is not compliant w.r.t. the specification. One can argue that not all runs are actually needed to manifest an error, since most symptoms (wrong traces) are caused by the same error. However also the generation of just a *significative* finite subset of the possible runs is not so feasible because, on one hand, a considerable manual effort is needed even to produce a single test scenario and, on the other hand, test-case generation is well-known to be a hard problem.

On the contrary *static* (semantics-based) tools could guarantee that *any* run will be compliant w.r.t. the specification, without even adding extra overhead. The problem with this approach is that it is in general (well-known to be) undecidable and, in any case, much more difficult to tackle.

Many researchers are proposing static approaches based on Model Checking, but this suffers of the state explosion problem and thus (while suitable for protocols and small hardware systems) cannot scale to most software system sizes, typical of (commercial) software production. Moreover there is also an inherent limit to verification of a single specific property of the system at a time.

This paper is motivated by the fact that we believe we can attack the undecidability of the static approach by using Abstract Interpretation techniques [10,11,12,13,9,3]. Abstract Interpretation is a theory of approximation of mathematical structures, in particular those involved in the semantic models of computer systems. Abstract interpretation can be applied to the systematic construction of methods and effective algorithms to approximate undecidable or very complex problems in computer science such as the semantics, the proof, the static analysis, the verification, the safety and the security of software or hardware computer systems. In particular, abstract interpretation-based static analysis, which automatically infers dynamic properties of computer systems,

---

[1] See http://semantik.informatik.uni-oldenburg.de/~jass/.

[2] See http://icontract2.org/.

[3] See http://www.aechmea.de/html/german/Information01_e.htm.

has been very successful these last years to automatically verify complex properties of real-time, safety critical, embedded systems.

We already had plenty of experience in Debugging and Verification of Declarative Languages where, by using Abstract Interpretation techniques, we could develop effective semantic-based tools [8,4,7,6,5,2]. The nice feature of this approach is that it can discover bugs even in absence of symptoms. Moreover it does not need a complete system to work, since we can (must) use, in place of missing components, their specification to diagnose existing parts.

This could be the case also for most systems providing UML diagrams with OCL specifications. However, given the level of complexity of such systems, it can easily be the case that the UML diagram *in itself* is not consistent. This would render the use of (complex), either static or dynamic, code diagnosis tools completely pointless. Hence it is important to have a tool to statically check the consistency of an UML model to achieve a good design *even before* the implementation starts. It can help further debugging stages and it is important in itself for Model Validation. This is even more important in Model Driven Architecture (MDA) approaches where new diagrams and code are automatically synthesized from the initial model: all the constructed artifacts would inherit the initial inconsistency. These considerations lead us to propose the conceptual framework described on the following.

The paper is structured as follows: in Section 2 we introduce some concepts about assertions in UML with OCL. In Section 3 we present our Conceptual Framework with an example to show how it works. Then in Section 4 we discuss about its applicability for development of software verification tools.

2 Assertions in the Software Engineering Practice

2.1 Design by Contract

In this section we will look how some of the concepts introduced above can be transferred to software systems in practice. Design by Contract (DbC) [21] is inspired by formal approaches embodied in specification languages such as Z and VDM. Bertrand Meyer has coined the concept of DbC to denote a software development style which (1) emphasizes the importance of formal specifications, (2) interleaves them with actual code, and (3) makes these contracts executable. DbC is a systematic method of assertion usage and interpretation introduced as a standard feature of the Eiffel language [20]. Without it, no trial would have ever been made to provide a similar mechanism in other languages and, by no means, would we have discussion papers like this and the ones mentioned in the references.

Software contracts have been invented to capture mutual obligations and benefits among classes, as they are e.g. needed in design patterns, where each of the involved classes is expected to exhibit a “proper” behavior [14,16]. A software contract is the specification of the behavior of a class and its associated methods. The contract outlines the responsibilities of both the caller and the method being called. Failure to meet any of the responsibilities stated in the contract results in a break of the contract itself, and indicates the existence of a bug somewhere in the design, in the implementation, in both of them, or - one must not forget this possibility in earlier project phases - in the assertions themselves. Software contracts can be completely specified through the use of preconditions, postconditions, and class invariants in object-oriented software. DbC views software construction as based on contracts between clients (callers) and suppliers (routines). Each party expects some benefits from the contract, and accepts some obligations in return. As in human affairs, the contract document spells out these mutual benefits and obligations and protects both the client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope. The DbC paradigm is as follow:

> The client's obligation is to call a method only in a program state where both the class invariant and the method's precondition hold. The method, in return, guarantees that the work specified in the postcondition has been done, and the class invariant is still respected.

A precondition violation is a manifestation of an error in the client, while a postcondition failure is a manifestation of a bug in the (implementation of the) supplier, which does not fulfill its promise (Note: The phrase "An assertion fails" in real life means just the opposite: the assertion did its job well, because it has found a bug). For this reason, in order to call a method, the client should verify only its preconditions. If the preconditions are satisfied, it should take for grant the postcondition after the termination of the method execution. The supplier, vice versa, should check the postconditions in order to guarantee its part of the contract, but under no circumstances shall the body of the method ever test for its preconditions. Under the *Non Redundancy Principle* [21], hence, the DbC encourage the developer to "check less and get more". DbC is, in this respect, the opposite of defensive programming, which recommends to protect every software module by as many checks as possible. This may result in redundancy and makes it also difficult to precisely assign responsibilities among modules.

2.2 UML and OCL

In the last few years, much effort has been spent to make the UML language more precise. Since its beginning, UML was conceived as a standard graphical language suitable to support the development of object-oriented systems. A clear intent in the UML design was the unification of the previous modeling languages, which all provided different notations for the same concepts. The standardization process was made by the Object Management Group (OMG), involving both the industry and the academia worlds. The results of this process was a relatively stable language, with an informal semantics. This level of definition was sufficient for sketching analysis and design models. However, when the model needed to be elaborated by automated tools for validation and verification purposes, the lack of a more formal foundation was immediately recognized. Because UML focused primarily on the diagrammatic elements and gave meaning to those elements through English text, a constraint language was added to the specification, in order to provide a more precise definition of the UML meta-model. That language was the Object Constraint Language (OCL) [30], initially developed in 1995 at IBM. OCL allows the integration of both well-formedness rules and assertions (i.e., preconditions, postconditions, invariants) in UML models. The former are useful to validate especially the syntax of a UML model, whereas the latter can be exploited to verify the conceptual constraints.

Preconditions and postconditions provide a mechanism to specify the properties required before and after the execution of an operation, respectively. They do not specify how that operation internally works. The recent development of version 2 for both OCL [22] and UML [23] is a breakthrough in order to completely define the semantics of a method in an object-oriented system. In these last versions, it is possible to define a behavior specification in OCL for any query operation (an operation without side-effects).

Following [26], now we summarize the relevant concepts about UML diagrams and the OCL specification language. For the sake of simplicity, here we present just a summary of the most important results.

In this work we use OCL as specification language to define software contracts such as method preconditions and postconditions, class invariants, and assertions in general. Hence we now define an object model $\mathcal{M}$ that contains the UML elements relevant for this task. Because preconditions, postconditions and invariants are defined typically for class diagram elements (i.e., class attributes and methods), we consider for the moment only the static structure of a UML model. A (static) object model $\mathcal{M}$ can be represented by the following tuple:

$$\mathcal{M} = \langle \mathit{CLASS}, \mathit{ATT}, \mathit{OP}, \mathit{ASSOC}, \preceq, \mathit{associates}, \mathit{roles}, \mathit{multiplicities} \rangle$$

where CLASS is a set of UML classes, ATT is a set of attributes, OP is a set of operations, ASSOC is a set of associations, $\preceq$ is a generalization hierarchy over classes, and *associations*, *roles*, and *multiplicities* are functions that give for each $as \in \mathit{ASSOC}$ its dedicated classes, classes' role names, and multiplicities, respectively (see [24] for complete definitions).

For an object model $\mathcal{M}$ providing a set of types $T_{\mathcal{M}}$, a relation $\leq$ on types reflecting the type hierarchy, and a set of operations $\Omega_{\mathcal{M}}$, the definition of OCL expressions is based upon the signature:

$$\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$$

According to [26], by using this signature we can define the OCL expressions syntax in the following way. Let $\mathtt{Var} = \{\mathtt{Var}_t\}_{t \in T_M}$ be a family of variable sets where each variable set is indexed by a type t. An expression over the signature $\mathtt{Expr}_{\mathcal{M}}$ is given by a set $\mathtt{Expr} = \{\mathtt{Expr}_t\}_{t \in T_M}$ and a function $\mathtt{free} : \mathtt{Expr} \rightarrow \mathcal{F}(\mathtt{Var})$ defined as follow.

- If $v \in \mathtt{Var}_t$ then $v \in \mathtt{Expr}_t$ and $\mathtt{free}(v) := \{v\}$.

- If $v \in \texttt{Var}_{t_1}, e_1 \in \texttt{Expr}_{t_1}, e_2 \in \texttt{Expr}_{t_2}$ then $\texttt{let } \mathbf{v = e_1 \text{ in } e_2} \in \texttt{Expr}_{t_2}$ and $\texttt{free}(\texttt{let } v = e_1 \, in \, e_2) := \texttt{free}(v) - \{v\}$.
- If $\omega : t_1 \text{ x } ... \text{ x } t_n \rightarrow t \in \Omega_{\mathcal{M}}$ and $e_i \in \texttt{Expr}_{t_i}$ for all $i = 1, ..., n$ then $\omega(\mathbf{e_1}, ..., \mathbf{e_n}) \in \texttt{Expr}_t$ and $\texttt{free}(\omega(e_1, ..., e_n)) := \texttt{free}(e_1) \cup ... \cup \texttt{free}(e_n)$.
- If $e_1 \in \texttt{Expr}_{Boolean}$ and $e_2, e_3 \in \texttt{Expr}_t$ then $\texttt{if } \mathbf{e_1} \texttt{ then } \mathbf{e_2} \texttt{ else } \mathbf{e_3} \texttt{ endif} \in \texttt{Expr}_t$ and $\texttt{free}(\text{ if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ endif}) := \texttt{free}(e_1) \cup \texttt{free}(e_2) \cup \texttt{free}(e_3)$.
- If $e \in \texttt{Expr}_t$ and $t' \leq t$ or $t \leq t'$ then $(\mathbf{e} \textsf{ asType } \mathbf{t'}) \in \texttt{Expr}_{t'}$,$(\mathbf{e} \texttt{ isType } \mathbf{t'}) \in \texttt{Expr}_{Boolean}$, $(\mathbf{e} \texttt{ isKindOf } \mathbf{t'}) \in \texttt{Expr}_{Boolean}$ and $\texttt{free}((e \textsf{asType} t')) := \texttt{free}(e)$, $\texttt{free}((e \texttt{ isTypeOf } t')) := \texttt{free}(e), \texttt{free}((e \texttt{ isKindOf } t')) := \texttt{free}(e)$.
- If $\mathbf{e_1} \rightarrow \mathbf{iterate}(\mathbf{v_1}; \mathbf{v_2} = \mathbf{e_2} | \mathbf{e_3}) \in \texttt{Expr}_{t_2}$ and $\texttt{free}(e_1 \rightarrow iterate(v_1; v_2 = e_2 | e_3)) := (\texttt{free}(e_1) \cup \texttt{free}(e_2) \cup \texttt{free}(e_3)) - \{v_1, v_2\}$.

In order to properly address the subtyping relation, an expression of type t' is also an expression of a more general type t. Hence, for all $t' \leq t$, if $e \in \texttt{Expr}_{t'}$ then $e \in \texttt{Expr}_t$.

Using the syntax defined above, we can start to write assertions in OCL, embedding them in a UML model, as we will show in Example 1.

3 A Conceptual Framework for Static Verification of Dynamic Diagrams Consistency

The expressive power of object-oriented paradigm makes it better suited for development of large software systems than the traditional imperative paradigm. However, the statically checks enforced by e.g. C++ or Java compilers test for such syntactic and typing restrictions only that guarantee the lack of runtime type errors. This is the contracting and specification level that has been used for too many years in the past by most software developers. Obviously, this is not enough to prevent surprising and often disastrous behavior of programs. In other words, the checks done by compilers are only part of what is needed to reason about the behavior (i.e., the semantics) of software.

Software contracts are a necessary prerequisite for being able to introduce a notion of correctness: if you do not state what your program should do, you are lacking the norm to which to compare what your program does in reality. In defining class correctness we follow [21], p. 370:

Definition 1 ([21]). *A class C is* correct *with respect to its specification if*

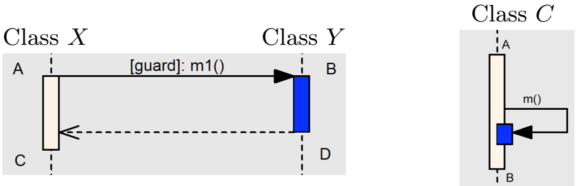
- *For any set of valid arguments* $e_1, \ldots, e_n$ *to a creation procedure p:*

$$\{Default_C \wedge Pre_{\texttt{p}} [\boldsymbol{x}/\boldsymbol{e}]\} \; p \; \{Post_{\texttt{p}} [\boldsymbol{x}/\boldsymbol{e}] \wedge Inv_{\texttt{C}}\}$$

- *For every public method m and any set of valid arguments* $e_1, \ldots, e_n$*:*

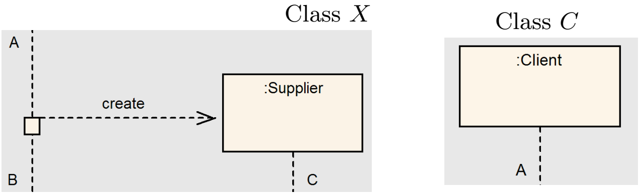
$$\{Pre_{\texttt{m}} [\boldsymbol{x}/\boldsymbol{e}] \wedge Inv_{\texttt{C}}\} \; m \; \{Post_{\texttt{m}} [\boldsymbol{x}/\boldsymbol{e}] \wedge Inv_{\texttt{C}}\}$$

where $Default_C$ *denotes the assertion expressing that the attributes of C have the default values of their type.*



(a) (Guarded) Method Call

(b) Self Method Call



(c) Create Object

(d) Object Life Line

Figure 1. Sequence Diagrams Basic Building Blocks

This notion clearly states what has to happen when we call a method in a state which satisfies $Pre_{\mathtt{m}}[\boldsymbol{x}/\boldsymbol{e}] \wedge Inv_{\mathtt{C}}$, but what happens when this does not hold? As already said, failure to meet any of the responsibilities stated in the contract results in a break of the contract, and indicates the existence of a bug somewhere in the design or implementation of the software or in the assertions themselves. Due to the size of most systems, the latter chance is not so unlikely. In this paper we want to focus on this situation, proposing our conceptual framework. In particular we want to check dynamic diagrams (and in particular Sequence Diagrams) against static diagrams and OCL specifications. In other words, the idea is to consider Class Diagrams and OCL specifications as a kind of meta-specification and all the dynamic diagrams as meta-code which has to conform the specification.

Thus we aim to guarantee that, by following the control flow on the diagram, the state is strong enough to satisfy the entry precondition of methods calls. For the sake of simplicity, we further restrict our attention to Sequence Diagrams which does neither involve concurrence nor timing constraints. This would require to define a much more complex verification method due to the complexity of considering more than one control flow at a time. We believe that, even with this restrictions, we have nevertheless a good level of generality to cover most of the existing software systems.

We will define our verification method by structural induction on the (graphical) syntax of Sequence Diagrams. Thus, in order to proceed, we need to specify in a formal way the graphical syntax of Sequence Diagrams; mainly formalize the way to compose (connect) basic diagrams to obtain bigger ones. Since messages

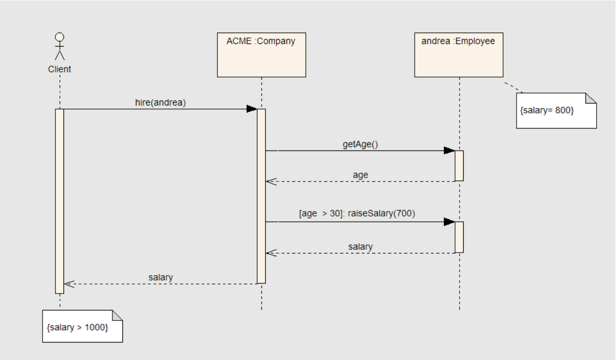


Figure 2. An Example of Sequence Diagram

(calls) involves at most two objects at a time, we consider graphical blocks that refer to the lifetime of at most 2 objects at the same time.

Let us start, for the sake of simplicity, with the basic diagrams of Figure 1. They have entry and exit points which are graphically connected to exits and entries of other blocks. We introduce a function link that, given an entry point of a block, returns the exit point of the block to which the former is connected, and vice versa.

Most important than this, blocks can be nested. Inside the colored parts of blocks of type 1(a) and 1(b) we can plug blocks of type 1(b) or the left side of blocks of type 1(a) and 1(c). We can also plug any arbitrary sequential composition of the latter. We can trivially extend function link to take this kind of connections into account. The following example surely clarifies better than many words.

Example 1 (Decomposition of Sequence Diagrams in Blocks).

The sequence diagram of Figure 2 is decomposed in blocks according to our schema as in Figure 3. The whole diagram is composed of 2 blocks β_1, β_2 of type 1(d) connected to a outer block β_3 of type 1(a) which inside contains the sequential composition of two other blocks β_4, β_5, both of them of type 1(a). Thus function link in this case is defined as

$$\begin{array}{lll} \mathrm{link}(B) = L & \mathrm{link}(N) = B & \mathrm{link}(C) = M \\ \mathrm{link}(D) = O & \mathrm{link}(E) = P & \mathrm{link}(G) = Q \end{array}$$

while the blue boxes in the diagram indicate the block division. (Note: these boxes are not part of the UML syntax.)

In UML Sequence Diagrams, especially version 2, there are also several diagrammatic elements which are naturally composed with inner blocks. This is the case of all fragments, for example those in Figures 4(a) and 4(b), where we define a new block by inserting blocks inside the “blank holes”.

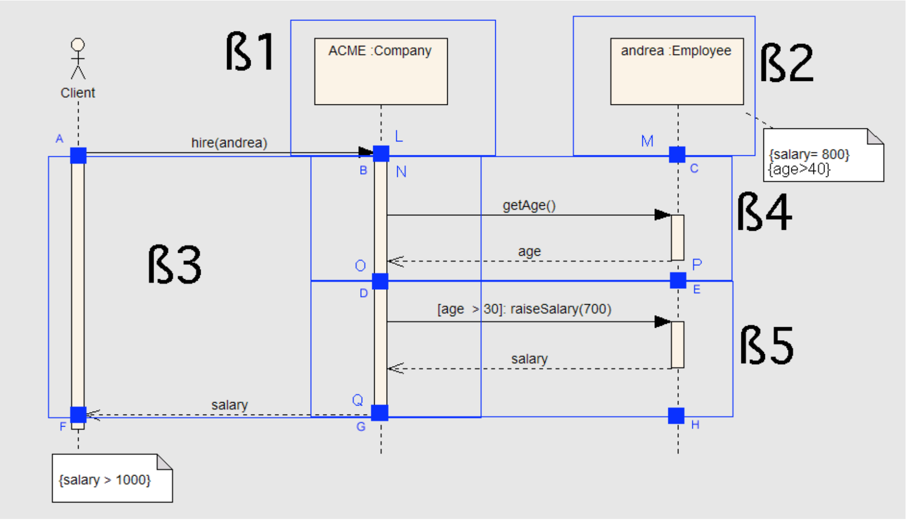


Figure 3. Decomposition of Sequence Diagrams in Building Blocks

We can handle Interaction Use Fragments simply by properly extending the link function. We can handle Gates simply by glueing two corresponding diagrams along the corresponding connection points.

For economy of space, we do not explicitly show all other possible diagrammatic elements as they can be treated analogously.

3.1 The Static Verification Method

We can now define our verification method by structural induction on the graphical syntax of Sequence Diagrams. The idea we follow here is first to introduce formula variables for all points of the blocks, then collect equalities between formula variables of the linked points and then add all the implications that must hold within the formula variables inside the various blocks according to their semantics. The implications that do not hold show manifestly an inconsistency of the sequence diagram.

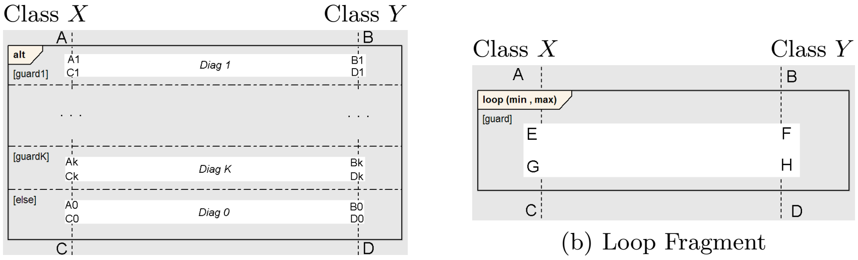
Let now present the various possibilities. We assume now that methods are called with actual arguments $e_1, \ldots, e_n$ (denoted by $\boldsymbol{e}$) and that its formal parameters are $x_1, \ldots, x_n$ (denoted by $\boldsymbol{x}$).

(Guarded) Method Call (Figure 1(a)) We need to impose that

$$\begin{aligned} \Phi_C &= \mathrm{result}(\Phi_A) \wedge \mathit{Post}_{\mathtt{m1}}\,[\boldsymbol{x}/\boldsymbol{e}] \qquad & \Phi_A &= \Phi_{\mathrm{link}(A)} \\ \Phi_D &= \Phi_B \wedge \mathit{Post}_{\mathtt{m1}}\,[\boldsymbol{x}/\boldsymbol{e}] & \Phi_B &= \Phi_{\mathrm{link}(B)} \end{aligned}$$

and check that

$$\Phi_A \wedge \mathit{guard} \wedge \Phi_B \implies \mathit{Pre}_{\mathtt{m1}}\,[\boldsymbol{x}/\boldsymbol{e}] \tag{3.1}$$



(a) Alternative Fragment

(b) Loop Fragment

Figure 4. Sequence Diagrams Fragments

$$\Phi_D \Longrightarrow Inv_{\mathtt{Y}} \qquad\qquad \Phi_C \Longrightarrow Inv_{\mathtt{X}} \tag{3.2}$$

where result(Φ_A) denotes the formula Φ_A modified (if it is the case) by inserting the result of method `m1` in the container specified by the call in class X.

Equation (3.1) prescribes that in order to call method `m1` the states of caller and callee, under the guard condition, have to be strong enough to guarantee that the precondition of the method holds. Equations (3.2) prescribe that the states reached by the caller and that by the callee do not invalidate the corresponding class invariant.

The Unguarded Method Call is just a particular case with $guard := True$.

Self Method Call (Figure 1(b)) Analogously to the previous case, we need to impose that

$$\Phi_B = \mathrm{result}(\Phi_A) \wedge Post_{\mathtt{m}}\,[\boldsymbol{x}/\boldsymbol{e}] \qquad\qquad \Phi_A = \Phi_{\mathrm{link}(A)}$$

and check that

$$\Phi_A \wedge guard \Longrightarrow Pre_{\mathtt{m}}\,[\boldsymbol{x}/\boldsymbol{e}] \qquad\qquad \Phi_B \Longrightarrow Inv_{\mathtt{C}}$$

Create Object (Figure 1(c)) We need to impose that

$$\Phi_B = \Phi_A \qquad\qquad \Phi_A = \Phi_{\mathrm{link}(A)} \qquad\qquad \Phi_C = Default_X$$

and check that $\Phi_C \Longrightarrow Inv_{\mathtt{X}}$.

Object Life Line (Figure 1(d)) We need to impose that $\Phi_A = Inv_{\mathtt{C}}$.

Alternative Fragment (Figure 4(a)) We need to impose, for all $0 \leq i \leq k$, that

$$\begin{aligned}
\Phi_A &= \Phi_{\mathrm{link}(A)} & \Phi_{A_i} &= \Phi_A \wedge guard_i \\
\Phi_B &= \Phi_{\mathrm{link}(B)} & \Phi_{B_i} &= \Phi_B \wedge guard_i \\
\Phi_C &= \bigvee_{0 \leq i \leq k} (guard_i \wedge \Phi_{C_i}) & \Phi_D &= \bigvee_{0 \leq i \leq k} (guard_i \wedge \Phi_{D_i}),
\end{aligned}$$

and check that $\Phi_D \Longrightarrow Inv_{\mathtt{Y}}$, $\Phi_C \Longrightarrow Inv_{\mathtt{X}}$, where

$$guard_0 := \neg \bigvee_{1 \leq i \leq k} guard_i$$

Note that the Alternative Method Call, as well as the Option and Break Fragments, are just a special case of this one.

Loop (Figure 4(b)) The loop fragment can be handled with special care. If we would have formulas within the body of loop which depend, even indirectly, upon each loop iteration then we would need to use a universal quantifier. Thus, to limit the expressive power of the underlying logic, we chose to treat just loop bodies which do not depend on iterations. With this assumption, we simply need to impose that

$$\begin{array}{lll} \Phi_A = \Phi_{\mathrm{link}(A)} & \Phi_E = \Phi_A \wedge guard & \Phi_C = \Phi_G \\ \Phi_B = \Phi_{\mathrm{link}(B)} & \Phi_F = \Phi_B \wedge guard & \Phi_D = \Phi_H \end{array}$$

The Fragments which inherently require concurrency, like Parallel, Critical Regions, Weak Sequential (when it does not boil down to Strict Sequentiality), cannot be considered without a concurrent semantics and thus fall out of our current scope.

Up to now we cannot consider too either the Assertion and Negative Fragments, as they would require to use an universal quantification over diagrams.

3.2 The Verification Method at work

Now we provide a complete example in order to show how our method can be applied in practice. We start to describe the static description of a software system, building the class diagram shown in Figure 5. In this diagram, the *Company* and *Employee* classes are defined. In particular, *Employee* has the following attributes: *age* of type *Integer*, name of type *String*, and *salary* of type *Double*. Similarly, the attributes of class *Company* are *location* and *name* (both of type *String*). *Employee* has two methods: *getAge*, which takes no arguments and returns an *Integer* value (the age), and *raiseSalary* which takes a *Double* and return a *Double* (the raised salary). *Company* has two methods: *fire* and *hire*, both of which takes an object of type *Employee* as argument. The method *hire* returns a *Double* (the salary of the hired employee). If the employee to be hired has an age greater than 30 years, the *hire* method call *raiseSalary* in order to increase the current employee's salary of 700 units. This scenario is depicted in the sequence diagram shown in Figure 3. Then we add the following OCL software contracts specifications for both Employee and Company classes. For the class Employee:

```
context Employee
  inv: (self.age >= 18)

context Employee::getAge() : Integer
```

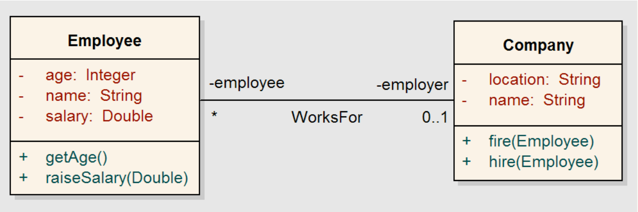


Figure 5. Example Class Diagram

```
  pre:  true
  post: (result = self.age)

context Employee::raiseSalary(amount : Double) : Double
  pre:  true
  post: (self.salary = (self.salary@pre + amount))
  post: (result = self.salary)
```

For the class Company:

```
context Company
  inv: self.employee->size() == self.employee->asSet()->size()

context Company::hire(p : Employee)
  pre  hirePre1: p.isDefined
  pre  hirePre2: self.employee->excludes(p)
  post hirePost: self.employee->includes(p)

context Company::fire(p : Employee)
  pre  firePre: self.employee->includes(p)
  post firePost: self.employee->excludes(p)
```

Now we show what we obtain with our method on the sequence diagram of Figure 3. The equalities on formula variables are:

$$\Phi_B = \Phi_N = \Phi_L = Inv_{Company} =$$
$$(Company.employee -> size() \equiv Company.employee -> asSet() -> size())$$
$$\Phi_C = \Phi_M = (Andrea.age \geq 40 \wedge salary \equiv 800)$$
$$\Phi_D = \Phi_O = (Inv_{Company} \wedge Result \equiv Andrea.age)$$
$$\Phi_E = \Phi_P = (Andrea.age \geq 40 \wedge salary \equiv 800 \wedge Result \equiv Andrea.age)$$
$$\Phi_Q = (Inv_{Company} \wedge Result \equiv 1500)$$
$$\Phi_H = (Andrea.age \geq 40 \wedge salary \equiv 1500 \wedge Andrea.salary \equiv 1500 \wedge Result \equiv 1500)$$
$$\Phi_A = (Company.isDefined \wedge Andrea.isDefined)$$
$$\Phi_F = (salary \equiv 1500 \wedge Result \equiv 1500)$$
$$\Phi_G = (Inv_{Company} \wedge Company.employee -> includes(Andrea))$$

While the implications that we have to check are

$$\begin{gathered}
\mathit{Inv}_{\mathit{Company}} \wedge \mathit{Andrea.age} \geq 40 \wedge \mathit{salary} \equiv 800 \Longrightarrow \mathit{True} \\
\mathit{Andrea.age} \geq 40 \wedge \mathit{salary} \equiv 800 \wedge \mathit{Result} \equiv \mathit{Andrea.age} \Longrightarrow \mathit{Andrea.age} \geq 18 \\
\mathit{Inv}_{\mathit{Company}} \wedge \mathit{Result} \equiv \mathit{Andrea.age} \Longrightarrow \mathit{Inv}_{\mathit{Company}} \\
\mathit{Inv}_{\mathit{Company}} \wedge \mathit{Andrea.age} \geq 30 \wedge \mathit{Andrea.age} \geq 40 \wedge \mathit{salary} \equiv 800 \Longrightarrow \mathit{True} \\
\mathit{Inv}_{\mathit{Company}} \wedge \mathit{Result} \equiv 1500 \Longrightarrow \mathit{Inv}_{\mathit{Company}} \\
\mathit{Andrea.age} \geq 40 \wedge \mathit{Andrea.salary} \equiv 1500 \Longrightarrow \mathit{Andrea.age} \geq 18 \\
\begin{aligned} &Company.isDefined \wedge Andrea.isDefined \wedge \mathit{Inv}_{\mathit{Company}} \Longrightarrow \\ &\quad Andrea.isDefined \wedge \mathit{Company.employee} -> \mathit{excludes}(\mathit{Andrea}) \end{aligned} \qquad \text{(i)} \\
\mathit{Inv}_{\mathit{Company}} \wedge \mathit{Company.employee} -> \mathit{includes}(\mathit{Andrea}) \Longrightarrow \mathit{Inv}_{\mathit{Company}}
\end{gathered}$$

All implications can be verified except of (i). Actually looking at (i) we discover that there is nothing in the diagram which specifies that *Andrea* is not already an hired employee. If we add in the diagram an initial constraint specifying that $\mathit{Company.employee} -> \mathit{excludes}(\mathit{Andrea})$ then we can prove the new (i) and then the diagram becomes consistent. This example suggests that in practical situations the assertions to be added in order to reach consistency can be quite easily derived by just inspecting the failing formula.

4 Applicability of the conceptual method

As already stated, the conceptual method that we have just presented is only a first step in a much more ambitious direction. Clearly in its generality it cannot be implemented because automatic proof of the verification formulas is undecidable. We think that even in this case the dimension of the state space generated is so large that it cannot be explored explicitly by model-checking techniques nor reasonably covered by testing.

However there are two possible nowadays well explored directions which we can follow from now on. One is that of using some proof assistant, like Coq[4]. The Coq tool is a formal proof management system: a proof done with Coq is mechanically checked by the machine. This direction of research is quite fertile in the literature. Several tools are being built on top of Coq, for object-oriented software verification purposes. For example Krakatoa[5] is a Java code certification tool that uses Coq to verify the soundness of implementations with regards to the specifications and Caduceus[6] is a verification tool for C programs.

However even computer-aided formal proofs tend to be humanly demanding and economically costly. An alternative is to use Abstract Interpretation Techniques where an abstraction of the semantics of the programs is automatically computed. This leaves out all information about reachable states which is not

---

[4] See http://coq.inria.fr/.

[5] See http://krakatoa.lri.fr/.

[6] See http://why.lri.fr/caduceus/.

strictly necessary for the proof. Of course if the abstraction is too precise, the computation cost are too high (resource exhaustion) and if it is too rough, nothing can be proved (false alarm). Although the best abstraction does exist, it is not computable, and so, must be found experimentally.

There has been a lot of research on these topics with promising results. Indeed recently we find tools based on Abstract Interpretation like Astree[7] [3] that can be used with great success for verification purposes of large C software systems.

5 Related Approaches and Future Works

In the literature we find also other completely different approaches to Model Consistency Verification, like the one of [29] where the information specified in class and Statechart Diagrams has been explicitly integrated into Sequence Diagrams. With these enriched diagrams, designers can *hopefully* identify gaps and contradictions in the specifications. In the future we would like to follow a similar idea and extend our method in order to collect state formulas from the Statechart Diagrams and inject them in the formula schema. This would *automatize* the identification of this kind of inconsistencies, without even cluttering the Sequence Diagrams.

The USE Tool [25] and the related works [15][27] represent another approach to achieve similar goals. These works consider validation by generating snapshots as prototypical instances of a model and comparing them against the specified model. In this way, snapshots provide immediate feedback and can be visualized using UML object diagrams. However the snapshots are defined by the user using a snapshot specification language (for object creation and operation calls). Eventually the sequence diagrams are automatically generated from the specified snapshot. Conversely, we prefer to use the standard UML notation for describing interactions, instead of a proprietary textual language, more suited to describe constraints than pictures, without mention the fact that modeling the system dynamics with diagrams is a standard practice in (object-oriented) software development.

There are also many approaches based on the ideas of software contracts and proof obligation generation. For example, the Java Modeling Language (JML) [17] is a formal behavioral interface specification language for Java. As such it allows one to specify both the syntactic interface of Java code and its intended behavior. However the JML approach is tightly coupled with the implementation programming language (in this case Java), whereas our framework is language independent. Moreover, our semantic-based verification approach can be applied even before the implementation starts. On the proof obligation side, the B formal method [1] provides a formal notation based on set theory and supported by automated tools, allowing specification, refinement, and proof. The B specification language come from Z and is not integrated neither in UML, nor in tool which support UML. There are works such as [18] which propose transformation

---

[7] See http://www.astree.ens.fr/.

rules in order to map OCL constraints into B formal expressions, but they are motivated by the lack of direct, automatic, static, and semantic-based verification tools in the UML/OCL arena. The conceptual framework presented here is a preliminary work toward the development of such tools.

References

1. J. R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 2005.
2. M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation – 12th International Workshop, LOPSTR 2002, Revised Selected Papers*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16, Berlin, 2003. Springer-Verlag.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03), June 7–14, San Diego, California, USA*, pages 196–207, New York, NY, USA, 2003. ACM Press.
4. M. Comini. VeriPolyTypes: a tool for Verification of Logic Programs with respect to Type Specifications. In M. Falaschi, editor, *Proceedings of 11th International Workshop on Functional and (constraint) Logic Programming*, number UDMI/18/2002/RR in Research Reports, pages 233–236, Udine, Italy, 2002. Dipartimento di Matematica e Informatica, Università di Udine.
5. M. Comini, R. Gori, and G. Levi. Assertion based Inductive Verification Methods for Logic Programs. In A. K. Seda, editor, *Proceedings of MFCSIT'2000*, volume 40 of *Electronic Notes in Theoretical Computer Science*, pages 1–18, North Holland, 2001. Elsevier Science Publishers. Available at URL: `http://www.elsevier.nl/locate/entcs/volume40.html`.
6. M. Comini, R. Gori, and G. Levi. How to Transform an Analyzer into a Verifier. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming and Automated Reasoning. Proceedings of the 8th International Conference (LPAR'01)*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 595–609, Berlin, 2001. Springer-Verlag.
7. M. Comini, R. Gori, and G. Levi. Logic programs as specifications in the inductive verification of logic programs. In A. Dovier, M. C. Meo, and A. Omicini, editors, *Declarative Programming – Selected Papers from AGP 2000*, volume 48 of *Electronic Notes in Theoretical Computer Science*, pages 1–16, North Holland, 2001. Elsevier Science Publishers. Available at URL: `http://www.elsevier.nl/locate/entcs/volume48.html`.
8. M. Comini, R. Gori, G. Levi, and P. Volpe. Abstract Interpretation based Verification of Logic Programs. *Science of Computer Programming*, 49(1–3):89–123, 2003.
9. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 1-2:47–103, 2002.
10. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Los Angeles, California, January 17–19*, pages 238–252, New York, NY, USA, 1977. ACM Press.

11. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, San Antonio, Texas, January 29–31*, pages 269–282, New York, NY, USA, 1979. ACM Press.
12. P. Cousot and R. Cousot. 'A la Floyd' induction principles for proving inevitability properties of programs. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 277–312. Cambridge University Press, Cambridge, UK, 1985.
13. P. Cousot and R. Cousot. A language independent proof of the soundness and completeness of generalized Hoare logic. *Information and Computation*, 80(2):165–191, 1989.
14. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
15. Martin Gogolla, Jörn Bohling, and Mark Richters. Validating uml and ocl models in use by automatic snapshot generation. *Software and System Modeling*, 4(4):386–398, 2005.
16. J.-M. Jézéquel, M. Train, and C. Mingins. *Design Pattern and Contracts*. Addison-Wesley, Reading, MA, 2000.
17. G. Leavens and Y. Cheon. Design by contract with jml, 2003.
18. R. Marcano and N. Levy. Transformation rules of OCL constraints into B formal expressions. Technical report, Universit de Versailles, 2002.
19. B. Meyer. Applying "Design by Contract". *Computer: Innovative Technology for Computer Professionals*, 25(10):40–51, 1992.
20. B. Meyer. *Eiffel – the Language*. Prentice-Hall, Englewood Cliffs, NJ, 1992.
21. B. Meyer. *Object-Oriented Software Construction, 2/E*. Prentice-Hall, Englewood Cliffs, NJ, 1997.
22. Object Management Group. *UML 2.0 OCL Specification*. Document – ptc/05-06-06 (OCL FTF report – convenience document).
23. Object Management Group. *UML 2.0 Superstructure Specification, v2.0*. Document – formal/05-07-04 (UML Superstructure Specification, v2.0).
24. M. Richters. A precise approach to validating uml models and ocl constraints. Technical report, University of Bremen, 2001. draft.
25. M. Richters. The use tool: A uml-based specification environment. 2001.
26. M. Richters and M. Gogolla. OCL: Syntax, Semantics, and Tools. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 42–68. Springer-Verlag, Berlin, 2002.
27. Mark Richters and Martin Gogolla. Validating uml models and ocl constraints. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*, volume 1939 of *Lecture Notes in Computer Science*, pages 265–277. Springer, 2000.
28. H. Toth. On theory and practice of Assertion Based Software Development. *Journal of Object Technology*, 4(2):109–130, 2005.
29. A. Tsiolakis. Integrating Model Information in UML Sequence Diagrams. In L. Baresi, M. Pezzè, and G. Taentzer, editors, *Proc. of the Satellite Workshops of the 28th ICALP (GT-VMT 2001)*, volume 50 of *Electronic Notes in Theoretical Computer Science*, pages 268–276, North Holland, 2001. Elsevier Science Publishers.
30. J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, Reading, MA, 2003.