Checking 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 UML is nowadays a de-facto standard for design and development of (object-oriented) software. With version 2.0 UML has achieved a more precise formal semantics. The same happened to OCL, a specification language which is an integral part of UML that allows to embed software contracts in the model.
In this work we propose an approach for a static verification of consistency of UML models which relies on OCL constraints.
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 drawbacks. For example, it cannot generally guarantee that a constraint will never be violated, unless an infinite number of tests is performed. Also the generation of just a *significative* finite subset 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 other hand, static approaches based on model checking suffer of the state explosion problem and thus cannot scale to real system sizes.
In this paper we propose a static verification technique that, by using OCL constraints together with class diagrams, certifies that the dynamic part of the model is satisfied. This encompasses the weaknesses of the other mentioned methods as it does not require users to build test scenarios and also performs the verification of *all* system properties at the same time.

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 [21]). The basic foundations have been laid by Bertrand Meyer with his concept of *Design by Contract* (DbC) as realized in the Eiffel language (see [16,15]). This approach

has then rapidly spread to other languages, for instance there emerged lot of support for assertions for Java (e.g. Jass, *etc.*) and C++ (e.g. the iContract tool, *etc.*). Mostly important 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.

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 by mistake side effects). 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 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 real software system sizes. 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 [9,10,11,12,8,2]. 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, 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 [7,3,6,5,4,1]. The nice of this approach is that it can discover bugs even in absence of symptoms. Moreover it does not

need of 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 real 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 why now we propose a conceptual framework for static verification of UML model consistency.

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) [17] is inspired by formal approaches embodied in specification languages such as Z, VDM, etc. Bertrand Meyer has coined the concept of DbC to denote a software development style which (1) emphasizes the importance of formal specifications, and (2) interleaves them with actual code. DbC is a systematic method of assertion usage and interpretation introduced as a standard feature of the Eiffel language [16]. 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 [13,14]. 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, 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* [17], 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 years, many efforts 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 UML meta-model. That language was the Object Constraint Language (OCL) [22], 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 [18] and UML [19] 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 [20], now we summarize the relevant concepts about UML diagrams, the OCL specification language and the action semantics. 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:

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

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 $as \in ASSOC$ its dedicated classes, classes' role names, and multiplicities, respectively.

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:

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

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

– If $\mathbf{e_1} \rightarrow \mathbf{iterate}(\mathbf{v_1}; \mathbf{v_2} = \mathbf{e_2}|\mathbf{e_3}) \in \mathtt{Expr}_{t_2}$ and $\mathtt{free}(e_1 \rightarrow \mathit{iterate}(v_1; v_2 = e_2|e_3)) := (\mathtt{free}(e_1) \cup \mathtt{free}(e_2) \cup \mathtt{free}(e_3)) - \{v_1, v_2\}$.

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 $e \in \mathtt{Expr}_{t'}$ then $e \in \mathtt{Expr}_t$.

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 2.

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 [17], p. 370:

Definition 1 ([17]). *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:*

$$\{\mathit{Default}_C \wedge \mathit{Pre}_{\mathtt{p}}\,[\boldsymbol{x}/\boldsymbol{e}]\}\; p\; \{\mathit{Post}_{\mathtt{p}}\,[\boldsymbol{x}/\boldsymbol{e}] \wedge \mathit{Inv}_{\mathtt{C}}\}$$

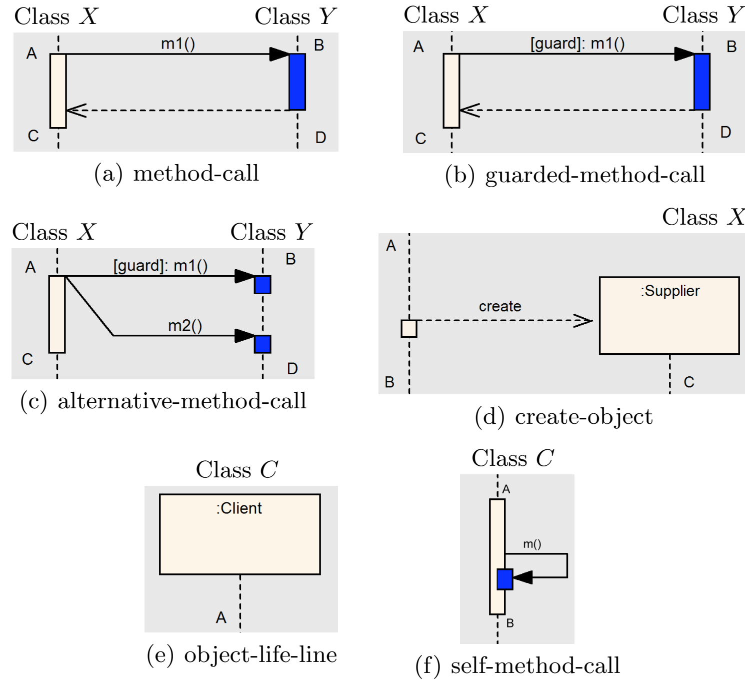
– *For every public method m and any set of valid arguments $e_1, \ldots, e_n$:*

$$\{\mathit{Pre}_{\mathtt{m}}\,[\boldsymbol{x}/\boldsymbol{e}] \wedge \mathit{Inv}_{\mathtt{C}}\}\; m\; \{\mathit{Post}_{\mathtt{m}}\,[\boldsymbol{x}/\boldsymbol{e}] \wedge \mathit{Inv}_{\mathtt{C}}\}$$

where $\mathit{Default}_C$ denotes the assertion expressing that the attributes of C have the default values of their type.

What happens, for example, when a diagram specifies a call to a method when $\mathit{Pre}_{\mathtt{m}}\,[\boldsymbol{x}/\boldsymbol{e}] \wedge \mathit{Inv}_{\mathtt{C}}$ 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 real systems the latter chance is not so unlikely. In this paper we want to focus on this choice and propose a conceptual framework to reason about consistency of a UML diagram. In particular we want



(a) method-call (b) guarded-method-call (c) alternative-method-call (d) create-object (e) object-life-line (f) self-method-call

Figure 1. Sequence Diagrams Building Blocks

to check dynamic 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. For the moment we restrict our attention to sequence diagrams.

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.

We will define our verification method by structural induction on the (graphical) syntax of sequence diagrams. Since for some diagram elements there is not a precise semantics, we consider now only the ones which (as far as we are aware of) are the most relevant in practice and have a precise meaning. We believe these are enough to maintain a good level of generality and to be useful for practical cases.

In order to proceed we need to specify in a formal way the graphical syntax of sequence diagrams; mainly how we can compose (connect) basic diagrams to obtain bigger ones. All graphical blocks (we consider) refer to the lifetime of at most 2 objects at the same time. Hence they fall in one of the taxonomy shown in 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

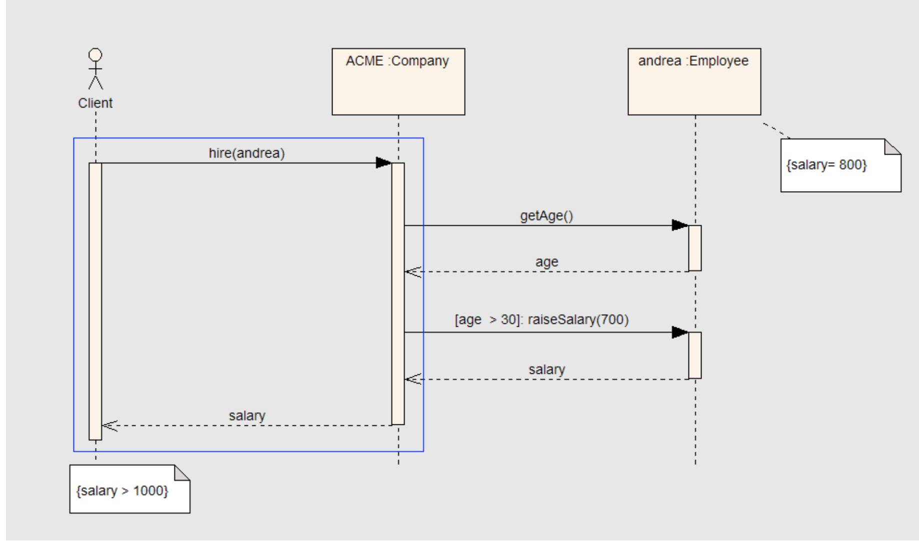


Figure 2. An Example of Sequence Diagram

entry point of a block, returns the exit point of the block to which the former is connected.

Most important than this, blocks can be nested. Inside the colored parts of blocks of type 1(a), 1(b), 1(c) and 1(f) we can plug blocks of type 1(f) or the left side of blocks of type 1(a), 1(b), 1(c) and 1(d). 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.

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(e) 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

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

while the blue boxes in the diagram indicate the block division.

We can now define our verification method by structural induction on the graphical syntax of sequence diagrams. The idea we follow here is to introduce formula variables for all points of the blocks, then we collect equalities between formula variables of the linked points and then we 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 inconsis-

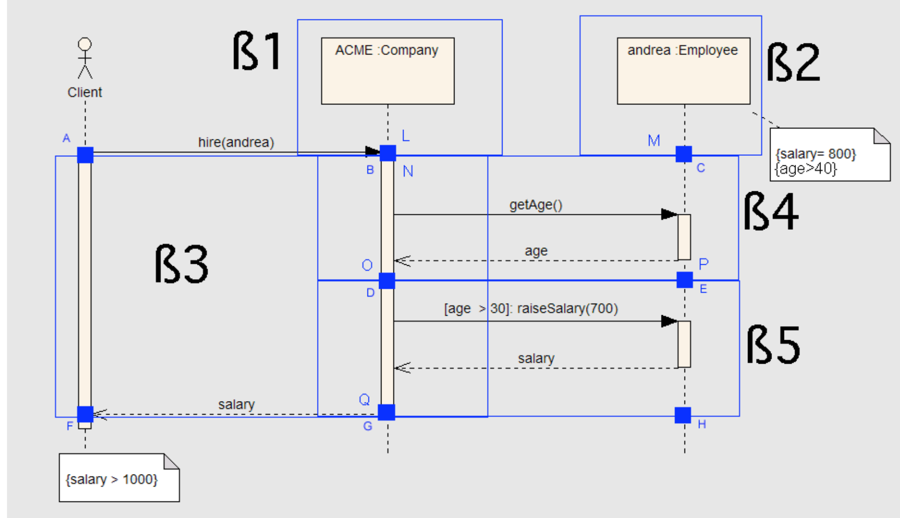


Figure 3. Decomposition of Sequence Diagrams in Building Blocks

tency of the sequence diagram, which can exploit a problem in the diagram or in the OCL specification of the method or of the class invariants involved.

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(b)) We need to impose that

$$\Phi_C = result(\Phi_A) \wedge Post_{\mathtt{m1}}\,[\boldsymbol{x}/\boldsymbol{e}] \qquad \Phi_A = \Phi_{link(A)}$$
$$\Phi_D = \Phi_B \wedge Post_{\mathtt{m1}}\,[\boldsymbol{x}/\boldsymbol{e}] \qquad \Phi_B = \Phi_{link(B)}$$

and check that

$$\Phi_A \wedge guard \wedge \Phi_B \implies Pre_{\mathtt{m1}}\,[\boldsymbol{x}/\boldsymbol{e}] \tag{3.1}$$
$$\Phi_D \implies Inv_{\mathtt{Y}} \tag{3.2}$$
$$\Phi_C \implies Inv_{\mathtt{X}} \tag{3.3}$$

where $result(\Phi_A)$ 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) and (3.3) prescribe that the states reached by the caller and that by the callee do not invalidate the corresponding class invariant.

(Unguarded) Method Call (Figure 1(a)) This is analogous to the previous with $guard := True$.

Conditional Method Call (Figure 1(c)) We need to impose that

$$\begin{aligned}
&\Phi_C = result(\Phi_A) \wedge ((guard \wedge Post_{\mathtt{m1}}[\boldsymbol{x}/\boldsymbol{e}]) \vee (\neg guard \wedge Post_{\mathtt{m2}}[\boldsymbol{x}/\boldsymbol{e}]))\\
&\Phi_A = \Phi_{link(A)}\\
&\Phi_D = (guard \wedge \Phi_B \wedge Post_{\mathtt{m1}}[\boldsymbol{x}/\boldsymbol{e}]) \vee (\neg guard \wedge \Phi_B \wedge Post_{\mathtt{m2}}[\boldsymbol{x}/\boldsymbol{e}])\\
&\Phi_B = \Phi_{link(B)}
\end{aligned}$$

and check that

$$\begin{aligned}
&\Phi_A \wedge guard \wedge \Phi_B \Longrightarrow Pre_{\mathtt{m1}}[\boldsymbol{x}/\boldsymbol{e}]\\
&\Phi_A \wedge \neg guard \wedge \Phi_B \Longrightarrow Pre_{\mathtt{m2}}[\boldsymbol{x}/\boldsymbol{e}]\\
&\Phi_D \Longrightarrow Inv_{\mathtt{Y}}\\
&\Phi_C \Longrightarrow Inv_{\mathtt{X}}
\end{aligned}$$

Self Method Call (Figure 1(f)) We need to impose that

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

and check that

$$\begin{aligned}
&\Phi_A \wedge guard \Longrightarrow Pre_{\mathtt{m}}[\boldsymbol{x}/\boldsymbol{e}]\\
&\Phi_B \Longrightarrow Inv_{\mathtt{C}}
\end{aligned}$$

Object Life Line (Figure 1(e)) We need to impose that

$$\Phi_A = Inv_{\mathtt{C}}$$

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

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

and check that

$$\Phi_C \Longrightarrow Inv_{\mathtt{X}}$$

Example 2 (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 4. 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

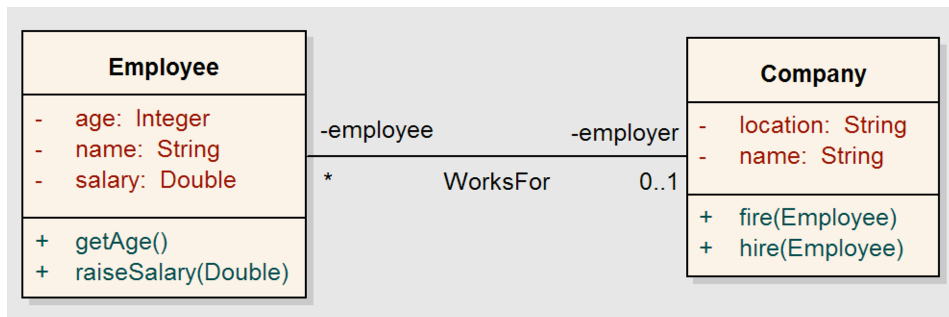


Figure 4. Example Class Diagram

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 now embed in the class diagram the software contracts specifications for both Employee and Company classes. In particular, we should define the classes invariants and the precondition - postconditions for each public method defined. We provide here part of this specification for the class Employee:

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

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

The specification of the class Company is as follows:

```
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 about the sequence diagram of Figure 3. The equalities on formula variables are:

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

While the implications that we have to check are

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

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 $Company.employee -> excludes(Andrea)$ then we can prove the new (i).

This example suggests that in practical situations the assertions to be added in order to reach consistency can be quite easy by just inspecting the failing formula.

4 Applicability of the conceptual method

As already stated, the conceptual method that we have just presented is just 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 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 [?]. 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 [?] is a Java code certification tool that uses Coq to verify the soundness of implementations with regards to the specifications and Caduceus [?] is a verification tool for C programs.

However even computer-aided formal proofs tend to be humanely 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 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 and recently we find tools based on Abstract Interpretation like Astree [2] that can be used with great success for verification purposes of large C software systems.

5 Conclusions

Many approaches for model validation and verification rely on generation of suitable code which dynamically checks the validity of OCL constraints. This approach has several drawbacks. For example, it cannot generally guarantee that a constraint will never be violated, unless an infinite number of tests is performed. Also the generation of just a *significative* finite subset 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 other hand, static approaches based on model checking suffer of the state explosion problem and thus cannot scale to real system sizes. Moreover they are also inherently limited to verification of a single specific property of the system at a time.

In this paper we presented an approach for a static verification of consistency of UML models. By using OCL constraints together with class diagrams, it certifies that the dynamic part of the model is satisfied. We hope to have convinced the reader that, to some extent, this encompasses the weaknesses of the other mentioned methods as it does not require users to build test scenarios and also performs the verification of *all* system properties at the same time.

Since the method is not necessarily effective, we suggested two possible directions of research to render it effective (without loosing its nice properties).

References

1. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 1-2:47–103, 2002.
9. 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.
10. 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.
11. 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.

12. 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.
13. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
14. J.-M. Jézéquel, M. Train, and C. Mingins. *Design Pattern and Contracts*. Addison-Wesley, Reading, MA, 2000.
15. B. Meyer. Applying “Design by Contract”. *Computer: Innovative Technology for Computer Professionals*, 25(10):40–51, 1992.
16. B. Meyer. *Eiffel – the Language*. Prentice-Hall, Englewood Cliffs, NJ, 1992.
17. B. Meyer. *Object-Oriented Software Construction, 2/E*. Prentice-Hall, Englewood Cliffs, NJ, 1997.
18. Object Management Group. *UML 2.0 OCL Specification*. Document – ptc/05-06-06 (OCL FTF report – convenience document).
19. Object Management Group. *UML 2.0 Superstructure Specification, v2.0*. Document – formal/05-07-04 (UML Superstructure Specification, v2.0).
20. 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.
21. H. Toth. On theory and practice of Assertion Based Software Development. *Journal of Object Technology*, 4(2):109–130, 2005.
22. J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, Reading, MA, 2003.