Model Checking of Extended OCL Constraints on UML Models in SOCLe[⋆]

John Mullins[1,⋆⋆] and Raveca Oarga[2]

[1] INRIA Rhône-Alpes, Domaine Scientifique de la Doua, Bât. Léonard de Vinci, 21, av. Jean Capelle - 69621 Villeurbanne Cedex. FRANCE.
[2] École Polytechnique de Montréal, Campus de l'U. de Montréal, Pavillon Mackay-Lassonde, 2500, Chemin de Polytechnique - Montréal (Qc) CANADA, H3T 1J4.

Abstract. We present the first tool that offers dynamic verification of extended OCL constraints on UML models. It translates a UML model into an Abstract State Machine (ASM) which is transformed by an ASM simulator into an abstract structure called *UML-valued OO TransitionSystem* ($OOTS_{UML}$). The *Extended Object Constraints Language* (EOCL) is interpreted on computation trees of this $OOTS_{UML}$ allowing for the statement of both OCL expressions modelling the system and OO primitives binding it to UML on the one hand, and safety or liveness constraints on the computation trees of the UML/OCL model on the other hand. An *on-the-fly* model checking algorithm, which provides the capability to work, at any time, on as small a possible subset of states as necessary, has been integrated into the tool.

1 Introduction

1.1 Motivation

Why use UML/OCL In recent years, the Unified Modeling Language (UML) has been accepted as a *de facto* standard for object-oriented software design. The UML notation supports designers by allowing them to express both structural and behavioral aspects of their design, through class diagrams and statechart diagrams respectively. Based on mathematical logic, the Object Constraint Language (OCL) is a notation embedded in UML allowing constraint specifications such as well-formedness conditions (e.g. in the definition of UML itself), and contracts between parts of the modeled system (e.g. class invariants or the pre- and post-condition methods).
Why use formal methods in UML/OCL Furthermore, used alongside formal method-based tools, UML/OCL also offers a unique opportunity for developing complex or critical software systems with high quality standards in an industrial context. Such systems require a high level guarantee that they can cope with their specifications from end to end of the development cycle.

[⋆] The SOCLe project is sponsored by Defence Research and Development Canada (DRDC) (Government of Canada).
[⋆⋆] Corresponding author. On leave from École Polytechnique de Montréal. Supported by an NSERC individual research grant (Government of Canada).

1.2 Related work

In this section we describe various proposed tools and promising verification frameworks that integrate UML/OCL in some way. These are divided into the following categories related to the depth of integration. First, tools performing UML model checking of logic where OCL is not embedded. Second, proposals to embed or extend OCL in a clean and systematic way, to take liveness properties into account, which have model checking in mind but where verification issues are not extensively discussed. Finally, UML tools for OCL constraints that support something other than model checking as a validation technique. (Due to space limitations, these are discussed in Appendix A-1).

Model checking of UML without OCL All these tools are based on translation of UML models into the modeling language of some model checking tool. While some of them opt explicitly for the specification language of the model checker itself to specify properties e.g.([1, 2, 3], others define methods based on the diagrammatic capabilities of UML to specify properties before verification [4, 5, 6].

The tool proposed in [2] maps the static part of a UML model, including OCL expressions on the object diagrams, onto an ASM and offers static verification including syntactic correctness according to the well-formedness constraints of the UML metamodel, and coherence of the object diagram with respect to the class diagram. However, OCL expressions are not integrated in the UML semantics, and are not evaluated as the model evolves. The statecharts modeling the behavior, are mapped onto SMV. Similarly, in [1] a subset of UML statechart diagrams is translated into Promela, the modeling language of the SPIN model checker. Specifications are then expressed in LTL (Linear Temporal Logic), the specification language of SPIN, for checking. UMC (UML *on-the-fly* Model Checking, [3]) is an environment that integrates the JACK model checker into UML. UMC also translates UML statecharts restricted to signals (no method calls or returns), into labeled transition systems, the modeling language of JACK, while properties are specified in ACTL, the specification language for JACK.

In [4], the authors propose an architecture for UML model checking based on a translation of UML models into IF, a language for the modeling and verification of hierarchical extended communicating automata. In IF, properties are specified by means of automata called *observer automata*, that are expressive enough to specify safety properties. Furthermore, an extension of UML, (*observers classes*), has been proposed to verify safety properties, which would then be translated along with the UML model into IF. The resulting IF automata are then translated into the specification language of equivalence-checkers like EVALUATOR. In [5], the authors propose vUML. This tool translates a restricted form of UML statechart diagrams into Promela. To express properties, statechart diagram annotations called *stereotypes* are used. The annotated statechart diagram is then translated into LTL before the model checking is done with SPIN. In HUGO [6], a restricted form of UML statechart diagrams is translated into Promela. *Collaboration diagrams* are used to specify properties. These are more expressive than vUML stereotypes. They allow some sequence patterns between objects and statecharts events to be described, but are not expressive enough to specify more complex properties. Collaboration diagrams are then translated into a Büchi automaton and SPIN solves *on-the-fly* the emptiness problem for the automata resulting from the synchronization of the Promela automata with the Büchi automaton.

Toward model checking OCL and beyond There are also many logical extensions of OCL with *modalities* [7, 8, 9] or translations of OCL into modal logic [10].

In [10], an object-based version of CTL called BOTL is used. BOTL, in contrast to EOCL does not extend OCL by temporal operators. Instead, it translates a fragment of OCL into BOTL. This means that temporal extensions of OCL could be translated into BOTL as well, but such extensions are not provided. Hence, verification issues for these extensions cannot be addressed. The purpose of BOTL is model checking of existing OCL. A strength of this work is that it provides a clear and precise object-based operational model which we reuse in $OOTS_{UML}$ with minor modifications together with extensions to cope with inheritance.

In [8] $\mathcal{O}_\mu(\texttt{OCL})$ is presented. Thus extends OCL with temporal constructs using observational μ-calculus, a two-level temporal logic in which temporal features at the higher level interact with the domain specific logic OCL at the lower level. Even though $\mathcal{O}_\mu(\texttt{OCL})$ was clearly designed with verification in mind, verification issues have not been extensively discussed in the paper. The strength of this work is that it provides a unified framework to design new logics combining the cleanly dynamic power of the μ-calculus with the static expressiveness of OCL (which we reuse in EOCL with minor modifications to cope with $OOTS_{UML}$ and CTL). In [7], the authors present an OCL extension, also based on CTL. This extension concerns system behavior modeled with statecharts, but evolution of attributes is not considered. In [9], an extension of OCL with elements of a bounded linear temporal logic is proposed. The semantics of this extension is given with respect to sequences of states representing the UML model history. However, the authors do not discuss how to compute these sequences from the model's behavioral diagrams.

SOCLe: model checking OCL on UML Starting from a clean framework drawn from the above second category proposals and taking advantage of lessons learned from the above first category proposals, we present the first UML model checker of EOCL.

1.3 Content of the paper

The work presented in this paper only addresses tool-related issues: The specification of an extended Object Constraint Language (EOCL) (Sec. 2) i.e. its syntax (Sec. 2.2), and operational semantics (Sec. 2.3) in terms of a transition system extended with state labels denoting UML-typed values ($OOTS_{UML}$) (Sec. 2.1), together with some illustrations of how this language might be used in practice to support a wider range of constraints on OO systems (Sec. 2.4); the sketch of the basic principles of the static (Sec. 3.1) and dynamic (Sec. 3.2) semantics of the modeling language UML on an Abstract State Machine (ASM); and a short overview of the SOCLe tool by itself (Sec. 4). A quick demonstration of the tool is given in Appendix A-2 while Sec. 5 provides the conclusion.

Out of the scope of the paper Due to space limitations, the *on-the-fly* model checking algorithm of EOCL on $OOTS_{UML}$ that is implemented in the tool is not presented here. The reader is referred to [11] for a systematic presentation of this algorithm, but in summary, it is a version of Vergauven and Lewi's *on-the-fly* linear time CTL model checking algorithm [12] extended to cope with $OOTS_{UML}$ and EOCL. The ASM semantics of UML itself are presented only to the extent necessary for understanding the way a $OOTS_{UML}$is uniformly generated from an UML/OCL model. For further details on Abstract State Machines, the reader is referred to [13] and [14] for a formal presentation of the ASM semantics of UML as implemented in the tool.

2 Extended OCL

We are now going to design logic that concentrates on the essential features of an object-oriented system (Sec. 2.2). Accordingly, we will define the semantics of this logic (Sec. 3) using a model called *UML-valued Object-Oriented Transition System* ($OOTS_{UML}$), which will be as simple as possible (Sec. 2.1) i.e. containing only the features of UML semantics addressable by the logic and nothing more. The degree of parallelism or the manner of method invocation, for example, need not be parts of $OOTS_{UML}$.

2.1 The abstract operational model

Let us start with some notations that we will use in the paper:

– Σ_c, a finite set of *class names* ranged over by c;
– Σ_m, a finite set of *method names* ranged over by m;
– $\mathcal{V}$, a finite set of variables;
– $\preceq_h$, a partial order over Σ_c called *inheritance relation*;
– $\preceq_o$, a partial order over $\Sigma_c \times \Sigma_m$ called *overriding relation* compatible with $\preceq_h$:
($\preceq_h$-compatibility) , if $(c, m) \preceq_o (c', m')$ then $m = m'$ and $c \preceq_h c'$
(i.e. an instance of method may only override a homonym in a superclass);
– $\mathcal{N}$, a countable indexing set ranged over by $i, j, \ldots$;
– $\mathcal{C} = \Sigma_c \times \mathcal{N}$, the set of *instances of classes*;
– $\mathcal{M} = \mathcal{C} \times \Sigma_m \times \mathcal{N}$, the set of *instances of methods*;
– $\mathcal{E} = \mathcal{C} + \mathcal{M}$ the set of *UML entities*;

As a first approximation, an $OOTS_{UML}$ can be seen as a transition system whose states are labeled with UML-typed values. A set Type of basic UML types is also defined:

$$\tau \in \mathtt{Type} = \mathtt{Void} \mid \mathtt{Int} \mid \mathtt{Bool} \mid \mathtt{Obj}^c \mid \mathtt{Met}^{c,m} \mid \mathtt{L}(\tau) \qquad (1)$$

where types Void, Int and Bool are defined in the usual way and, for every $c \in \Sigma_c$ and $m \in \Sigma_m$,

$$\begin{aligned} Val^{\mathtt{Obj}^c} &= \{c' : c' \preceq_h c\} \times \mathcal{N} \\ Val^{\mathtt{Met}^{c,m}} &= Val^{\mathtt{Obj}^c} \times \{m' : m' \preceq_o m\} \times \mathcal{N} \end{aligned}$$

are respectively the set of all objects of the class c and the set of all instances of the method m of objects of the class c. Finally, $\mathtt{L}(\tau)$ is the type of lists of type τ, with element $[]$ (the empty list) and $h :: w$ (for the list having the element h as head and the list w as tail). We will denote by Val, the universe of values i.e. $Val = \cup_{\tau \in \mathtt{Type}} Val^{\tau}$, and by $Val_{\perp}$ its extension with the undefined value ($\perp$). Finally, letting the symbol $\rightharpoonup$

denote a partial function, we define the *class signature* function $C \times M$ in the following way:

$$C \times M : \Sigma_c \rightharpoonup [\mathcal{V} \rightharpoonup \mathsf{Type}] \times [\Sigma_m \rightharpoonup [[\mathcal{V} \rightharpoonup \mathsf{Type}] \rightharpoonup \mathsf{Type}]]$$

which associates to a class c, the declaration $C(c)$ of its attributes, and the declaration $M(c)$ of its methods that is, for each method of the class, the declaration of its formal parameters and the type of its return parameter [3].

Definition 1. *A* `UML`-valued Object-Oriented Transition System *($OOTS_{UML}$) is a structure $\mathcal{OT} = \langle \mathcal{S}, \mathcal{R}, s_0 \rangle$ such that:*

– *$\mathcal{S}$ is a set of states. To each state $s \in \mathcal{S}$, are associated functions ρ_s, σ_s, γ_s and h_s such that:*
  - *$\rho_s : \mathcal{V} \rightharpoonup Val$, a valuation of attributes and method parameters in s;*
  - *$\sigma_s : \mathcal{C} \rightharpoonup [\mathcal{V} \rightharpoonup Val]$, a valuation of objects active in s consistent with $C \times M$ and ρ_s ($\mathcal{C}_s$ will denote the domain of σ_s);*
  - *$\gamma_s : \mathcal{M} \rightharpoonup [[\mathcal{V} \rightharpoonup Val] \rightharpoonup Val_\perp]$, a valuation of instances of methods active in s consistent with $C \times M$ and ρ_s ($\mathcal{M}_s$ will denote the domain of γ_s and $\gamma_s(m).f$, the value in s of the parameter f in the instance of method m);*
  - *$h_s : \mathcal{M} \rightharpoonup S \times \{\circ, \bullet\}$, a* history *which associates to each instance of method active in s, a pebble and the state where this instance is or has been called. If s is the last state for the instance before being returned, the pebble is $\bullet$ and otherwise, it is $\circ$. The history will provide more particularly a concise and elegant way to define the* `OCL` *operator $@\mathbf{pre}$. It has to be consistent with $\preceq_h$ and $\preceq_o$ i.e.:*

    ($\preceq_h$-consistency) *If $(c, i, m, j) \in \mathcal{M}_s$ then $m \in C(c)$ or ($m \in C(c')$ and $c \preceq_h c'$) (i.e. an instance of a method can be inherited);*

    ($\preceq_o$-consistency) *If $(c, i, m, j) \in \mathcal{M}_s$ and $(c, m) \preceq_o (c', m')$ then*

    $$\forall_{i,j \in \mathcal{N}} (c, i, m', j) \notin \mathcal{M}_s$$

    (i.e. the instance of an overriding method inhibits any instance of overridden ones);

– *$\mathcal{R} \subseteq S \times S$ is a transition relation*

– *$s_0 \in S$ is the initial state.*

A *computation* or *run* r in an $OOTS_{UML}$ $\mathcal{OT}$ is an infinite sequence of states $r = s_0 s_1 s_2 \cdots$ such that $(s_i, h_i, s_{i+1}, h_{i+1}) \in \mathcal{R}$, for each i. We denote by $r[i]$, the $(i+1)$-th element, s_i, of the path, and by $\mathcal{R}un(\mathcal{OT})$ the set of all computation paths in $\mathcal{OT}$. We denote by $\mathcal{R}un_s(\mathcal{OT})$ the subset of $\mathcal{R}un(\mathcal{OT})$ that comprises the computation paths starting from $s \in \mathcal{S}$.

[3] Let functions $f_1 : X \to Y_1$ and $f_2 : X \to Y_2$. The function $f_1 \times f_2 : X \to Y_1 \times Y_2$ is the function defined by $f_1 \times f_2(x) = (f_1(x), f_2(x))$.

2.2 Extended OCL syntax

We propose an extension of OCL (EOCL) working on $OOTS_{UML}$. EOCL is an extension of OCL with CTL temporal operators and some first-order features. It is two-level logic: intuitively, the upper level is CTL extended with quantifiers (the set of *temporal formulae* F_{exp}), and the lower level is a significant fragment of OCL expressions as defined in [15] (the set of *state formulae* P_{exp}). In order to get a clean separation of OCL expressions from purely temporal properties, we restrict OCL expressions to appearances within temporal operators or as atomic formulae of CTL. EOCL is largely inspired by BOTL [10] but is based on an instantiation of the temporal extension framework proposed in [8], and takes into account inheritance and overriding in its semantics. The EOCL syntax is given in Fig. 1. The set of types of OCL expressions is the same

$$\begin{aligned} e(\in P_{exp}) &::= x \mid v \mid e.\mathsf{a} \mid \omega(e_1, \ldots, e_n) \mid e_1 \longrightarrow \mathbf{iterate}\ \{x_1\ ;\ x_2 = e_2 \mid e_3\ \} \mid e\ ) \mid \\ &\quad\ e\ @\mathbf{pre} \mid e.\mathbf{owner} \mid \mathbf{act}(e) \\ \varphi(\in F_{exp}) &::= e \mid \neg\phi \mid \phi \wedge \psi \mid \forall z \vdash \tau : \phi \mid \mathbf{EX}\phi \mid \mathbf{E}[\phi \mathbf{U} \psi] \mid \mathbf{A}[\phi \mathbf{U} \psi] \end{aligned}$$

Fig. 1. EOCL syntax

one as the UML type set defined in Eq. 1. We write $e \vdash \tau$ to denote that expression $e \in P_{exp}$ has the type τ. We refer the reader to [11] for a complete definition of the typing function $\vdash$. The rest of this section is devoted to an informal description of the meaning of state and temporal formulae. We postpone to Sec. 2.3 the formal description of operational semantics based on $OOTS_{UML}$.
State formulae P_{exp}

- x is a variable in $\mathcal{V}$. These include $self$, a special variable in OCL referring to the current context, fields of objects, parameters of methods and local variables;
- v is a value in $Val_\perp$;
- $e.f$ is a *field/parameter navigation*. If e is an object (resp. a list of objects), then f is a field (resp. a list of fields). If e is a method (resp. a list of methods), then f is a parameter (resp. a list of parameters);
- $\omega(e_1, \ldots e_n)$ stands for the application of any *n*-ary operator on booleans, integers or lists.
- the $\mathbf{iterate}$ construct is the OCL main collection operator; It lets variable x_1 iterate through values of the collection denoted by e_1, stores successive values of e_3 in variable x_2 (which first evaluates to e_2), and returns the final value of x_2. The $\mathbf{iterate}$ construct is quite expressive and is used to encode additional collection operators (size, forall, exists, filter), that are also supported by SOCLe.
- $@\mathbf{pre}$ is a typical OCL operator. It refers to the value of a property at the method call, and may be applied in a postcondition at the method return;
- $\mathbf{act}(e)$ signifies that the object or method instance e is currently active. An object becomes active when it is created and becomes inactive when it dies, whereas a method becomes active when it is invoked (pushed onto a calling stack) and becomes inactive after it has returned a value (is popped from the calling stack).
- $e.\mathbf{owner}$ denotes the object executing the method e.

Temporal formulae F_{exp} A formula is built inductively from boolean state formulae ($e \in P_{exp}$ and $e \vdash \mathsf{Bool}$), classical propositional logic operators ($\neg, \wedge$, etc.), CTL temporal operators (AX, EU, etc.), and type domain quantifiers. The temporal operators have the following intuitive meaning:

– $\mathbf{EX}\phi$ holds in s if there is a state next to s such that the formula ϕ holds;
– $\mathbf{E}[\phi\mathbf{U}\psi]$ holds in s if there is a path starting from s such that ϕ holds until ψ holds;
– $\mathbf{A}[\phi\mathbf{U}\psi]$ holds in s if for every path starting from s, ϕ holds until ψ holds;
– $\forall z \vdash \tau : \phi$ holds in s if ϕ holds for all occurrences z of type $\tau \in \mathtt{Type}$;

The other usual auxilliary operators are obtained by combining these basic operators. It has to be noted that type domains being generally infinite, quantifier scopes are also so.

2.3 Extended OCL semantics

Let $\mathcal{OT} = \langle \mathcal{S}, \mathcal{R}, s_0 \rangle$, an $OOTS_{UML}$. The semantics of state formulae is defined by the function $[\![\_]\!] : P_{exp} \to [\mathcal{S} \to Val_\perp]$ defined as follows:

$$\begin{array}{lllll}
[\![v]\!]_s & = v & & \\
[\![x]\!]_s & = \rho_s(x) & & \\
[\![\omega(e_1, \ldots e_n)]\!]_s & = \omega([\![e_1]\!]_s \ldots [\![e_n]\!]_s) & & \\
[\![e.f]\!]_s & = \sigma_s((c,i))(f) & \text{if} & [\![e]\!]_s = (c,i) \\
 & = \gamma_s((c,i,m,j)).f & \text{if} & [\![e]\!]_s = (c,i,m,j) \\
[\![e.\mathbf{owner}]\!]_s & = (c,i), & \text{where} & [\![e]\!]_s = (c,i,m,j) \\
[\![\mathbf{act}(e)]\!]_s & = \mathsf{True} & \text{iff} & [\![e]\!]_s \in \mathcal{C}_s + \mathcal{M}_s
\end{array}$$

$$[\![e_1 \!\longrightarrow\! \mathbf{iterate}\{x_1; x_2 = e_2 \mid e_3\}]\!]_s$$
$$= [\![\mathbf{for}\ x_1 \in [\![e_1]\!]_s\ \mathbf{do}\ x_2 := e_3]\!]_{\rho_s[x_2 \mapsto [\![e_2]\!]_s]}$$
where
$$[\![\mathbf{for}\ x_1 \in []\ \mathbf{do}\ x_2 := e]\!]_s = [\![x_2]\!]_s$$
$$[\![\mathbf{for}\ x_1 \in h :: w\ \mathbf{do}\ x_2 := e]\!]_s = [\![\mathbf{for}\ x_1 \in w\ \mathbf{do}\ x_2 := e]\!]_{\rho_s[x_2 \mapsto [\![e]\!]_{\rho_s[x_1 \mapsto h]}]}$$

and $\rho_s[x \mapsto e]$ stands for the state obtained from s by evaluating x to e in ρ_s. Finally for any instance of method (c,i,m,j):

$$[\![e@\mathbf{pre}]\!]_s = \begin{cases} [\![e]\!]_{s'} & \text{if } (c,i,m,j) \in dom(\gamma_s) \text{and } h_s(c,i,m,j) = (s', \bullet) \\ \perp & \text{otherwise} \end{cases}$$

The semantics of temporal formulae (Fig. 2) is given by the relation $\models \subseteq \mathcal{S} \times F_{exp}$

2.4 Applying EOCL

Constraints are conditions which have to be fulfilled by the system. An OCL constraint is defined as being in a context. We denote by C_{exp}, the set of constraints defined by the following:

$$
\begin{array}{lcl}
s \models e & \iff & [\![e]\!]_s = \mathsf{True} \\
s \models \neg\phi_1 & \iff & s \not\models \phi_1 \\
s \models \phi_1 \wedge \phi_2 & \iff & (s \models \phi_1) \text{ and } (s \models \phi_2) \\
s \models \forall z \vdash \tau : \phi_1 & \iff & s \models \phi_1[z \mapsto v] \text{ for all } v \in Val^\tau \\
s \models EX\phi_1 & \iff & \exists_{r \in \mathcal{R}un_s(\mathcal{OT})} r[1] \models \phi_1 \\
s \models E[\phi_1 U \phi_2] & \iff & \exists_{r \in \mathcal{R}un_s(\mathcal{OT})} \\
 & & \exists_{j \geq 0} r[j] \models \phi_2 \wedge \forall_{0 \leq k < j} r[k] \models \phi_1 \\
s \models A[\phi_1 U \phi_2] & \iff & \forall_{r \in \mathcal{R}un_s(\mathcal{OT})} \\
 & & \exists_{j \geq 0} r[j] \models \phi_2 \wedge \forall_{0 \leq k < j} r[k] \models \phi_1
\end{array}
$$

Fig. 2. Semantics of temporal formulae

$$\kappa(\in C_{exp}) ::= \mathbf{context}\ C\ \mathbf{inv}\ e \ \mid\ \mathbf{context}\ C :: M\ \mathbf{pre}\ e_1\ \mathbf{post}\ e_2$$

where $C \in \Sigma_c$ is the context of an invariant, $M \in \Sigma_m$ is the context of a pre/postcondition and e, e_1, e_2 are boolean `OCL` expressions. Below, we illustrate how an `OCL` constraint has its counterpart in `EOCL`.

Invariant An invariant is a condition which has to be fulfilled by the system whenever an instance of the context, or of a class inherited from the context is active, and no method of *self* is executing. Since the $OOTS_{UML}$ semantics of `EOCL` takes into account inheritance, this can be expressed by the following constraint:

$\mathbf{context}\ C\ \mathbf{inv}\ e\ \equiv$

$$\textbf{AG}[\forall z \vdash \mathsf{Obj}^C : \textbf{act}(z) : ((\forall m_1 \in z.M_1 : \ldots \forall m_n \in z.M_k) : (\neg\textbf{act}(m_1) \wedge \ldots \neg\textbf{act}(m_n)) \Rightarrow e]$$

where

- $\forall m \in z.M : e$ stands for the formula $\forall m \vdash Met^{C,M} : (z.\textbf{owner} = m) \Rightarrow e$
- z is an active object of the class C;
- $\{M_1, \ldots M_k\}$ is the set of the methods of the class C;
- $\{m_1, \ldots m_n\}$ is the set of instances of the methods of the class C.

Pre/postcondition A pre/postcondition is verified if for each instance of M of the class C, the post-condition holds when M is returned whenever the pre-condition held when M was called. This can be expressed by the following constraint:

$\mathbf{context}\ C\ ::\ M\ \mathbf{pre}\ e_1\ \mathbf{post}\ e_2\ \equiv$

$$\forall z \vdash \mathsf{Obj}^C : \textbf{act}(z) : \forall m \in z.M : \textbf{AG}[call(m) \Rightarrow \textbf{AX}[\textbf{AG}[return(m)] \Rightarrow e_2]]$$

where

- $call(m)$ stands for the formula $\neg act(m) \wedge \textbf{EX}[act(m) \wedge e_1]$;
- $return(m)$ stands for the formula $act(m) \wedge \textbf{AX}[\neg act(m)]$;
- z is an object of the class C;
- m is an occurrence of the method M of the object z.

Extended OCL constraints EOCL allows the expression of liveness properties. For instance a template **after/eventually** could stand for the following property: whenever e_1 is verified during the life of any instance of C then eventually e_2 will also be verified during its life. This could be expressed as the extended constraint:

$$\textbf{context}\ C\ \ \textbf{after}\ e_1\ \textbf{eventually}\ e_2\ \equiv$$
$$\textbf{AG}[e_1 \Rightarrow \textbf{A}[\mathsf{True}\ \textbf{U}\ e_2]]$$

3 ASM semantics of UML

Why use ASM to define UML While designing semantics of logic requires as simple a model as possible, modeling UML by contrast, requires formalism that, like ASM, has already proved to be a simple and uniform fashion of modeling the operational semantics of models as complex as programming languages. ASM will allow a rich, succinct and understandable operational semantics of UML to be written. In this section, we present UML and its ASM semantics. The UML semantics are presented only to the extent necessary for understanding the way an $OOTS_{UML}$ is uniformly generated from an UML/OCL model. The reader is referred to [13, 14] for a more formal presentation of the ASM semantics of UML and the integration of OCL into this semantics.

An ASM state is a collection of sorts, and a set of enumerated functions for these sorts. ASM evolution is specified by a transition rule built from predicates, control subrules and update sub-rules. Predicates are evaluated according to the current interpretation of the ASM state enumerated functions. Control rules supporting non-determinism choose a set of update rules to be applied. Update rules modify the interpretation of the current ASM state functions, yielding successor states.

Basic model elements, such as class or method names, are mapped to sorts. More complex elements, such as method declarations and statechart transitions, are translated into enumerated functions. The object diagram is mapped to a specific subset of these functions, and represents the initial configuration of the UML model (Sec. 3.1). From a configuration, successor configurations are computed by evaluating an ASM rule that captures the dynamic semantics of UML models (Sec. 3.2). Edges are labeled with statechart transitions fired as the UML model evolves.

3.1 Static semantics of UML

The UML models supported by the tool must contain exactly one class diagram, one statechart diagram for each class, and one object diagram. In this section we illustrate the main features of the static semantics of these three diagrams through the modeling of a simple object-oriented component acting as a small memory cell.

Class diagram Fig. 3 presents the supported features of the class diagram. Class $Cell$ models a simple memory cell with assignment, retrieval and incrementation. Class $BackupCell$ models an extended memory cell with a restore functionality. Notice how class $Client$ is tagged with the $thread$ stereotype. As a result, a calling stack will be associated with all instances of this class.

The first step to create the ASM specification is to map class, method and field names to the following ASM sorts (note that an association is mapped to a field of the owner

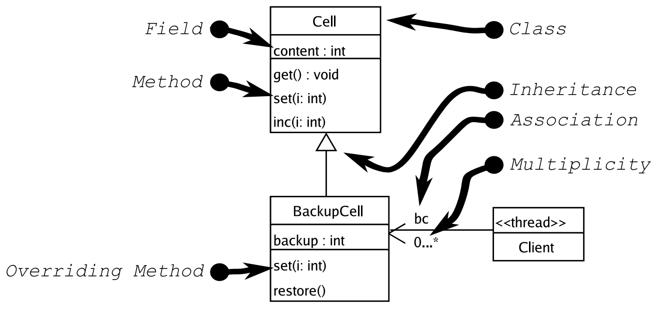


Fig. 3. Example of a class diagram

class), which implement states of the abstract model defined in Sec.2.1. More particularly, $heap$ implements σ while $stack$ implements γ.

$$\begin{aligned}&\textbf{sort } ClassName = \{Cell, BackupCell, Client\}\\ &\textbf{sort } FieldName = \{content, backup, bc\}\\ &\textbf{sort } MethName = \{set, get, inc, restore\}\end{aligned} \tag{2}$$

Remaining information, like the inheritance relation $\preceq_h$ and overriding relation $\preceq_o$, is then extracted, and additional functions are implemented. Here are some partially enumerated examples:

$$\begin{aligned}&\textbf{fun } \preceq_h \;\;= BackupCell \mapsto Cell, Cell \mapsto Cell, \ldots\\ &\textbf{fun } \preceq_o \;\;= Cell/set \mapsto Cell/set, BackupCell/set \mapsto Cell/set, \ldots\\ &\textbf{fun } lookup = BackupCell, inc \mapsto Cell, BackupCell, set \mapsto BackupCell, \ldots\end{aligned} \tag{3}$$

These functions are then used to define the important $lookup$ function indicating whether or not refined or inherited behavior will be executed following a method call.

Statechart diagrams Similarly, statechart diagrams are mapped to ASM sorts and functions. Fig. 4 shows supported features for this diagram. Notice how functionalities of the memory cell are modeled by sub-states specifying the behavior of a method. Method inc, for example, is modeled in three steps: transition ct_3 retrieves the current value of field $content$ by calling method get; transition ct_4 increments that current value by calling method set; finally, transition ct_5 waits for method set to return and terminates method inc.

The control flow of a statechart is specified by states and transitions. The basic condition for a transition to be fired is that its source state be active. The basic response to firing a transition is the activation of its target state. In the case of a composite state, the initial states it encompasses are also activated. This control flow of statecharts is inspired by Harel's statecharts [16] and is statically elaborated and stored in ASM functions. The compiler determines, for example, which states are activated and which deactivated when firing a transition:

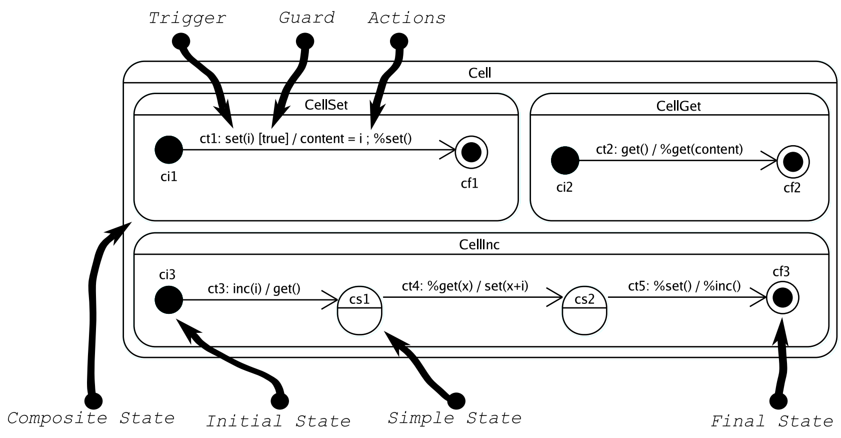


Fig. 4. Example of a statechart diagram

$$\begin{array}{ll}\textbf{fun}\ \ act & = ct_1 \mapsto \{\}, ct_3 \mapsto \{cs_1\}, \ldots \\ \textbf{fun}\ deact & = ct_1 \mapsto \{CellSet\}, ct_3 \mapsto \{ci_3\}, \ldots\end{array} \tag{4}$$

In addition, transitions are labeled with a trigger, a guard and a list of actions. Triggers refer to signals (atomic events), method calls or method returns. For example, the actions of a transition labeled with trigger inc, will model that method's instructions. Guards are boolean OCL expressions. The syntax of OCL expressions used on the UML models is given by Fig. 5, and their semantics are expressed in Sec. 2.2.

$$e(\in E_{exp}) ::= v \mid x \mid \omega(e_1, \ldots, e_n) \mid e\ .\ f \mid e_1 \longrightarrow \textbf{iterate}\ \{x_1\ ;\ x_2 = e_2 \mid e_3\ \}$$

Fig. 5. OCL expressions syntax

The tool supports the following actions: method call/return, field assignment, object creation/deletion, and signal emission. Actions are specified in part by OCL expressions, which enable the designer to model high-level behavior by using non-determinism. In a method call action, for instance, an OCL expression specifies a collection of possible receiver objects, from which the actual receiver is chosen non-deterministically.

Finally, statechart compiling includes a fair amount of static verification: $i)$ statecharts are inspected to ensure they are well-formed, $ii)$ OCL expressions are type-checked to ensure that guards are boolean expressions, that parameters of method calls are well-typed, etc. $iii)$ triggers and actions are analyzed to ensure consistency with the class diagram methods and field declarations.

Object diagram An object diagram is mapped to ASM sorts and functions that hold the UML model configuration. Fig. 6 shows such a diagram with all features covered by the tool. It models a simple configuration in which a client accesses two memory cells.

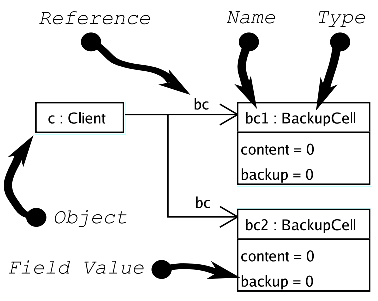


Fig. 6. Example of an object diagram

ASM functions as, $class$, $heap$ and $stack$ hold active states, object types, field environments, and calling stacks, (one for each thread - in this case only object c is a thread), respectively.

$$\begin{aligned} &\textbf{fun}\ \ as = bc_1 \mapsto \{ci_1, ci_2, \ldots\}, c \mapsto \{cli_1\}, \ldots \\ &\textbf{fun}\ class = bc_1 \mapsto BackupCell, c \mapsto Client, \ldots \\ &\textbf{fun}\ heap = bc_1, content \mapsto 0, bc_1, backup \mapsto 0, \ldots \\ &\textbf{fun}\ stack = c \mapsto \langle (run, \emptyset, \perp, c) \rangle \end{aligned} \quad (5)$$

Note how the calling stack of object c contains method run in the initial configuration to ensure that the thread is active.

3.2 Dynamic semantics of UML

The ASM transition rule that captures a UML model's dynamic semantics is structured roughly as follows: $i)$ choose the current thread, $ii)$ select the current object and current statechart, $iii)$ choose one of the enabled transitions and $iv)$ fire the transition. It sketches some of the ASM rules that are used to implement the transition relation of the abstract operational model defined in Sec. 2.1.

Sub-rules $i)$ and $iii)$ use a non-deterministic choice to model thread-level and statechart-level concurrency. Sub-rules $i)$ models a simple thread scheduler. The current object $\tilde{o}$ is selected from an active thread's calling stack, i.e. $\tilde{o}$ would be executing a method. Sub-rules $iii)$ computes the transition interleaving of a statechart's concurrent regions.

In sub-rule $ii)$, the current statechart is either the statechart of the current object's class or the statechart of one of its superclasses if inherited behavior is to be executed (this is decided according to the $lookup$ function of Eq. 3). This mechanism captures behavioral inheritance, an important feature of object-orientation.

Sub-rule $iii)$ dynamically determines whether a transition is enabled. The basic condition that the source state is active, is checked against the current value of function as (Eq. 5). Moreover, a transition is enabled if $a)$ its trigger corresponds to an active event, $b)$ its guard is satisfied, and $c)$ all of its actions can be fired. An assignment action, for example, will not be fired if it violates the multiplicity requirement (see Fig. 3).

In sub-rule $iv)$, the selected transition $\tilde{t}$ is fired. The basic effect of deactivating and activating states is captured by updating function as using the statically elaborated functions act and $deact$ (Eq. 4): $as(\tilde{o}) := (as(\tilde{o}) \setminus deact(\tilde{t})) \cup act(\tilde{t})$.

Then, the transition's action list is iterated and every action is fired. Objects are created by using the sort extensions mechanism of the ASM formalism, and by updating function $heap$ (Eq. 5) accordingly. If an object creation action a of the form "$\mathbf{new}\ f$" is fired by the current object $\tilde{o}$, for example, the following ASM sub-rule updates the UML model configuration:

$$\begin{array}{l} \mathbf{extend}\ Objects\ \mathbf{with}\ x\ \mathbf{d}o \\ \quad heap(\tilde{o}, f) := x :: heap(\tilde{o}, f) \end{array} \tag{6}$$

The assignment action uses the OCL expression evaluation function and updates function $heap$ (Eq. 5) accordingly. For example, if an assignment action a of the form "$content\ := \ self.content\ +\ i$" is fired by the current object $\tilde{o}$, the following ASM sub-rule updates the UML model configuration: $heap(\tilde{o}, content)\ := \ [\![a.e]\!]_\rho$. This sub-rule uses function $[\![\ ]\!]_\rho$ to evaluate $a.e$, the OCL expression of the assignment action (in this case "*self*$.content\ +\ i$"), and updates function $heap$ (Eq. 5). Function $[\![\ ]\!]_\rho$ evaluates an OCL expression by recursively evaluating its sub-expressions relative to the current UML configuration and a variable assignment ρ. The environment always maps the variable *self* to the current object $\tilde{o}$. In this case it also maps variable i to the value of the formal parameter of method inc as indicated on the calling stack. As the function is external, it uses ASM functions to access the current UML model configuration, but is not enumerated in the ASM state.

4 The tool SOCLe

The SOCLe tool is divided into three main modules: $i)$ an XmiToAsm compiler, $ii)$ a specialized ASM interpreter and $iii)$ an *on-the-fly* EOCL model checker. This architecture is depicted in Fig. 7. UML models are expressed in the XML Metadata Interchange format, which is supported by most UML CASE tools.

The verification process has two phases: $i)$ the UML model is translated into an ASM specification according to its UML model static semantics, and $ii)$ an execution graph implementing $OOTS_{UML}$ is generated from the ASM specification while OCL constraints are verified *on-the-fly* by this execution graph. The model checker implements a version of the Vergauven and Levi's *on-the-fly* linear time CTL model checking algorithm [12], that is extended to cope with $OOTS_{UML}$ and EOCL, thus improving an earlier version of SOCLe presented in [17], which is based on a naive approach to verification of OCL extended with fixed points to express temporal contracts.

The tool also includes a graphical user interface embedded into ArgoUML, a customizable open-source UML CASE tool developed by Tigris[4]. It allows the designer to visualize verification results and inspect the model's execution graph. A short demonstration of the tool is given in Appendix A-2. It compares the performance of the *on-the-fly* approach in Fig. 8 with the naive approach in Fig. 9, in the verification of the

[4] http://argouml.tigris.org/

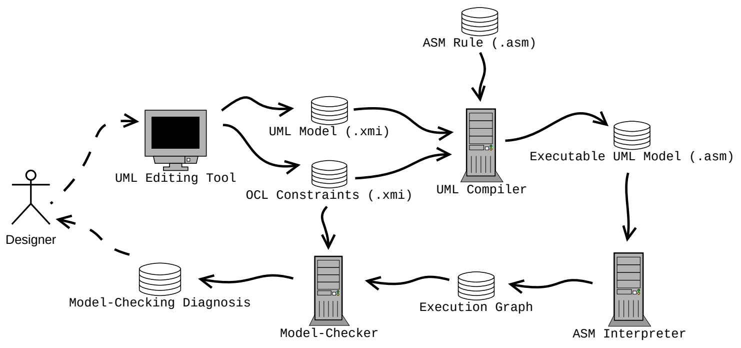


Fig. 7. Tool architecture

following invariant stating informally that the backup value of the attribute $backup$ is always smaller than or equal to $content$:

context: `BackupCell`
inv: *self*.`Cell`$/content <=$ *self*.`BackupCell`$/backup$

Finally, it should be noted that an extensive case-study on a simplified caveat-separation system has been carried out for Defence Research and Development Canada (DRDC) - Valcartier, as an illustrative example of possible application of SOCLe in the design of secure software. The full case-study has been reported in [18].

5 Conclusion and future work

In this paper we have presented the main issues related to SOCLe, an EOCL model-checker of UML models. Firstly, an extension of OCL interpreted into an OO Transition System with UML-type values, in which inheritance and overriding are considered as possibly leading to richer interpretations of extended OCL constraints, where these OO features will have to be taken into account in practice for such things as invariance verification. Secondly, illustrations of how the ASM based semantics of UML models capture complex features of UML such as concurrency, inheritance, overriding, and object creation; and integrate an OCL expression evaluation function that is defined relative to the UML configuration, thus generating an implementation of $OOTS_{UML}$ which includes EOCL models. Finally, the architecture of the tool itself is presented.

We have not extensively discussed the scheme for transformation from ASM to $OOTS_{UML}$ in this paper. Considerable further work is needed to prove the correctness of this transformation. Alongside this, we may directly extract an $OOTS_{UML}$ that represents a UML model saved as XML. Also, abstraction and symbolic techniques to check whether or not the $OOTS_{UML}$ satisfies an EOCL formula have to be developed. Because EOCL incorporates OCL as its lower-level logic, abstraction and symbolic techniques used to check CTL or real-time CTL formulae could be extended to EOCL formulae in a quite simple way.

References

[1] Latella, D., Majzik, I., Massink, M.: Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. The International Journal of Formal Methods **11** (1999) 637–664

[2] Shen, W., Compton, K., Huggins, J.K.: A tool for supporting UML static and dynamic model checking. In: 26th IEEE International Computer Software and Applications Conference (COMPSAC), Oxford, England, IEEE Computer Society (2002) 147–152

[3] Gnesi, S., Mazzanti, F.: On the fly model checking of communication UML state machine. In: Second ACIS International Conference on Software Engineering Research, Management and Applications (SERA2004). (2004)

[4] Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: Tools and applications II: The if tool. In Corradini, F., Bernardo, M., eds.: Proceedings of SFM'04. Volume 3185 of LNCS., Bertinoro, Italy, Springer (2004)

[5] Lilius, J., Paltor, I.P.: vUML: a tool for verifying UML models. In: IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE Computer Society (1999) 255–258

[6] Schäfer, T., Knapp, A., Merz, S.: Model Checking UML State Machines and Collaborations. In: CAV 2001 Workshop on Software Model Checking. Volume 55 (3) of ENTCS. (2001)

[7] Flake, S., Mueller, W.: An OCL extension for real-time constraints. In Clark, T., Warmer, J., eds.: Object Modeling with the OCL: The Rationale behind the Object Constraint Language. Springer (2002) 150–171

[8] Bradfield, J., Filipe, J.K., Stevens, P.: Enriching OCL using observational mu-calculus. In Kutsche, R.D., Weber, H., eds.: Fundamental Approaches to Software Engineering, Fifth International Conference, FASE 2002. Volume 2306 of LNCS., Springer (2002) 203–217

[9] Ziemann, P., Gogolla, M.: An OCL extension for formulating temporal constraints. Technical Report 1/03, Universität Bremen (2003)

[10] Distefano, D., Katoen, J.P., Rensink, R.: On a temporal logic for object-based systems. In Smith, S.F., Talcott, C.L., eds.: Formal Methods for Open Object-Based Distributed Systems IV - Proc. FMOODS'2000, Kluwer Academic Publishers (2000)

[11] Oarga, R.: On-the-fly verification of extended OCL constraints over UML models. Master's thesis, École Polytechnique de Montréal, Université de Montréal (2005) (*In French*).

[12] B. Vergauwen, J. Lewi: A Linear Local Model Checking Algorithm for CTL. In E. Best, ed.: 4th International Conference on Concurrency Theory (CONCUR'93). Volume 715 of LNCS., Hildesheim, Germany, Springer-Verlag (1993) 447–461

[13] Cavarra, A., Riccobene, E., Scandurra, P.: Mapping UML into abstract state machines: a framework to simulate UML. Studia Informatica Universalis. Volume 3(3) (2004) 367–398

[14] Bergeron, M.: An ASM semantics for UML/OCL. Master's thesis, École Polytechnique de Montréal, Université de Montréal (2004)

[15] OMG: Response to the UML 2.0 OCL RfP (ad/2000-09-03). Technical Report ad/2002-05-09 (2002)

[16] Harel, D., Naamad, A.: The STATEMATE Semantics of Statecharts. ACM Transactions on Software Engineering and Methodology **5** (1996) 293–333

[17] Azambre, D., Bergeron, M., Mullins, J.: Validating UML and OCL models in SOCLe by simulation and model checking. In J. Lilius et al., ed.: Proc. of MOMPES'05, 2nd International Workshop on Model Based Methodologies for Pervasive and Embedded Software. Number 39 in General Publications, TUCS (2005) 67–76

[18] Painchaud, F., Azambre, D., Bergeron, M., Mullins, J., Oarga, R.: Socle: Integrated design of software applications and security. In: Proceedings of The Tenth International Command and Control Research and Technology Symposium (ICCRTS 2005). (2005)

A-1 UML and OCL tools for objectives other than model checking

Some tools support OCL expressions and constraints, but with different objectives in mind (e.g. [1, 2, 3]).

The KeY tool [1] integrates deductive verification techniques within UML/OCL. It translates OCL constraints into dynamic logic for Java CARD, a proper subset of Java for smart-card applications and embedded systems (to specify proof obligations), and provides a state-of-the-art theorem prover to perform verification. The USE tool [2] offers the evaluation of OCL expressions and constraints on manually constructed object models and sequence diagrams. The OCLE tool [3] offers validation of OCL well formedness, profile and methodological rules, defined at the meta-model level on UML models, and i.e., static semantic validation of OCL constraints on UML models.

References

[1] Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Technical Report 2003-05, Department of Computing Science, Chalmers University of Technology and Göteborg University (2003)

[2] Gogolla, M., Richters, M., Bohling, J.: Tool Support for Validating UML and OCL Models Through Automatic Snapshot Generation. In: SAICSIT '03: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology, South African Institute for Computer Scientists and Information Technologists (2003) 248–257

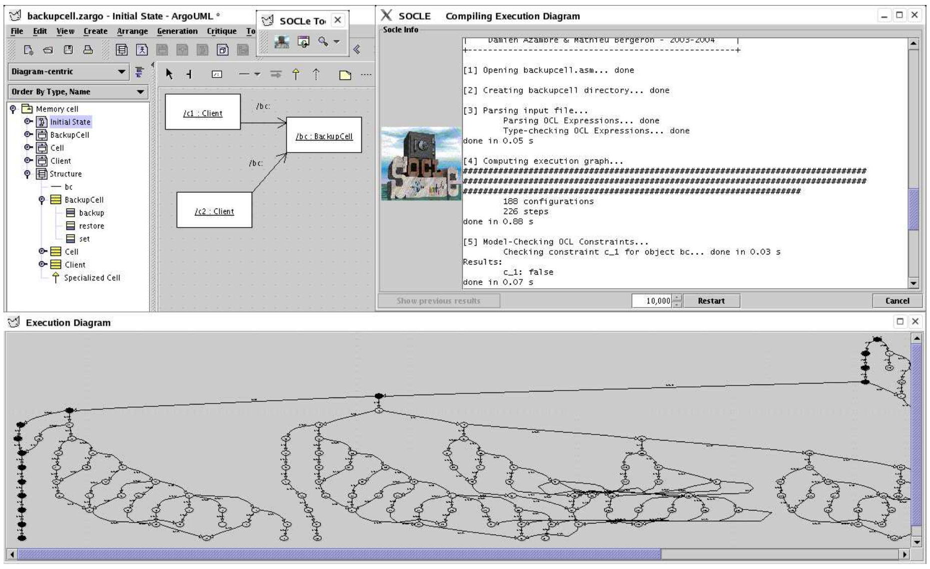
[3] Chiorean, D., Pasca, M., Carcu, A., Botiza, C., Moldovan, S.: Ensuring UML Models Consistency Using The OCL Environment. In: UML 2003 - OCL Workshop. (2003)

A-2 A quick tool demonstration

As an illustration of the *on-the-fly* approach compared to the naive one, consider the verification of the following invariant stating informally, that the backup value of the attribute $backup$ is always smaller than or equal to $content$:

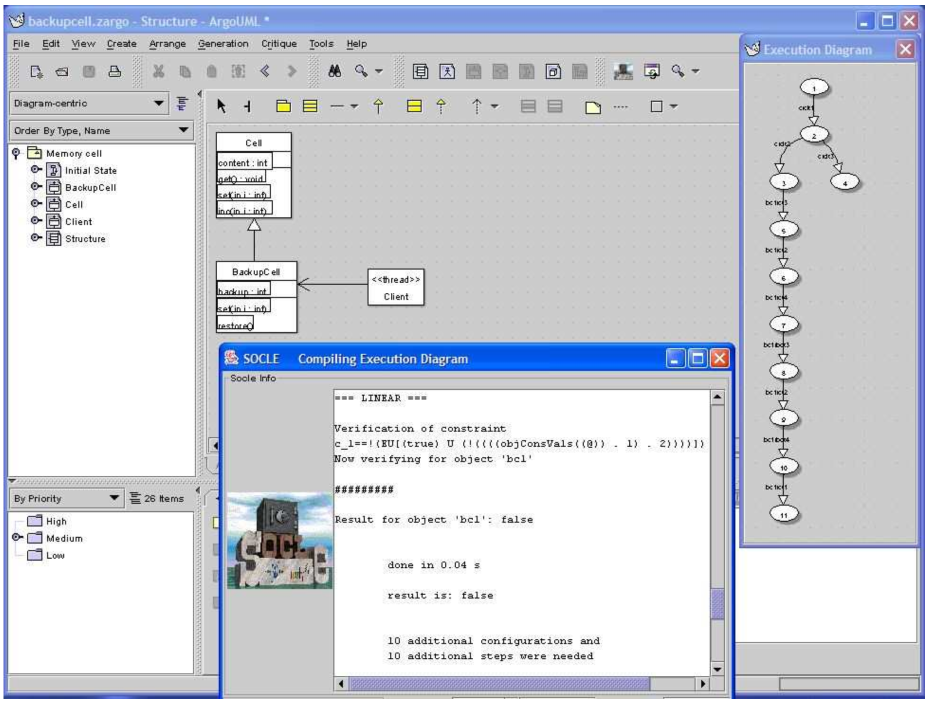
context: BackupCell
inv: *self*.Cell/$content <=$ *self*.BackupCell/$backup$

Fig. 8 and 9 provide illustrations of the screen-shots obtained using both approaches for verifying this invariant. With the *on-the-fly* EOCL verification algorithm, a counter-example is found after exploration of only eleven states of the model, while the naive fixed point OCL extension verification algorithm requires exploration of the full state space of the model (4630 states).



```
|     Damien Azambre & Mathieu Bergeron - 2003-2004     |
+-------------------------------------------------------+

[1] Opening backupcell.asm... done

[2] Creating backupcell directory... done

[3] Parsing input file...
        Parsing OCL Expressions... done
        Type-checking OCL Expressions... done
done in 0.05 s

[4] Computing execution graph...
#################################################################################
#################################################################################
####################################################################
        188 configurations
        226 steps
done in 0.88 s

[5] Model-Checking OCL Constraints...
        Checking constraint c_1 for object bc... done in 0.03 s
Results:
        c_1: false
done in 0.07 s
```

Fig. 8. The SOCLe tool diagnosis - μ-calculus



```
=== LINEAR ===

Verification of constraint
c_1==!(EU[(true) U (!((((objConsVals((@)) . 1) . 2))))])
Now verifying for object 'bc1'

#########

Result for object 'bc1': false


        done in 0.04 s

        result is: false


        10 additional configurations and
        10 additional steps were needed
```

Fig. 9. The SOCLe tool diagnosis - EOCL -logic