Towards model checking OCL

Dino Distefano, Joost-Pieter Katoen and Arend Rensink

Department of Computer Science, University of Twente

P.O. Box 217, 7500 AE Enschede, The Netherlands

E-mail: {ddino, katoen, rensink}@cs.utwente.nl

Abstract

This paper presents a logic, called BOTL (Object-Based Temporal Logic), that facilitates the specification of dynamic and static properties of object-based systems. The logic is based on the branching temporal logic CTL and the Object Constraint Language (OCL). Eventually, the aim is to do model checking. The formal semantics of BOTL is defined in terms of a general operational model that is aimed to be applicable to a wide range of object-oriented languages. A mapping of a large fragment of OCL onto BOTL is defined, thus providing a formal semantics to OCL and, at the same time, permitting model checking of OCL constraints.

Keywords: Object Constraint Language (OCL), formal verification, object-based system, property specification, temporal logic.

1 Introduction

Due to the ever increasing complexity of systems, attempts to assess their correctness by engineering "rules of thumb" do not work: they easily lead to wrong conclusions and may cause costly redesigns. Instead, a systematic and rigorous method for checking systems correctness is needed. For the specification and verification of reactive systems, the use of temporal logics has been thoroughly investigated. The availability of software tools that support the automatic verification of systems with respect to logical formulae has become popular and very successful. This applies in particular to the model checking approach [7, 8]. For object-oriented systems, however, such automated verification techniques have received scant attention.

In our project we aim at applying the model checking approach to object-oriented systems. As a first step, this paper presents a temporal logic, referred to as BOTL, that is suited for specifying *static* and *dynamic properties* of object-based systems. The dynamic properties are related to the behavior of the system when time evolves, while the static properties refer to the relations between syntactical entities such as classes. The logic is an object-based extension of the branching temporal logic CTL [6], a formalism for which efficient model checking algorithms and tools do exist. The object-based ingredients in our logic are largely inspired by the Object Constraint Language (OCL) [17, 22, 23]. The precise relationship with OCL is defined by means of a mapping of a large fragment of OCL including, amongst others, invariants, pre- and postconditions, navigations and iterations onto BOTL.

The semantics of the logic is defined in terms of a *general* operational model that is aimed to be applicable to a rather wide range of object-oriented programming languages. The operational model is a Kripke structure, in which states are equipped with information concerning the status of objects and method invocations. The mapping of BOTL onto these Kripke structures is defined in a formal, rigorous way [9]. We believe that such formal approach is indispensable for the construction of reliable software tools such as model checkers. Besides, the semantics of BOTL together with the aforementioned translation of OCL provides a *formal semantics* of OCL. (Alternative formalizations of OCL have been considered in [10, 12, 19].)

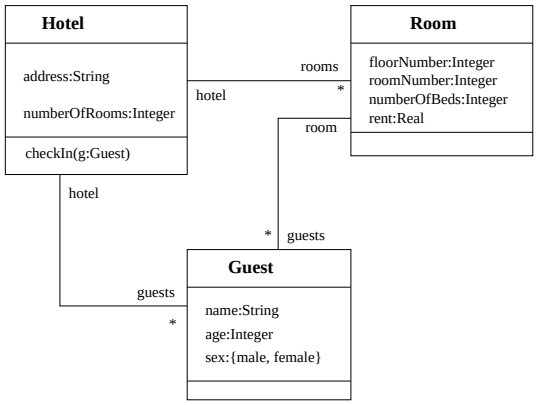

Figure 1: The Hotel Class Diagram

In this paper we confine ourselves to *object-based* systems, i.e., object-oriented systems in which inheritance and subtyping are not considered.

Classes and their associations can be described by UML *class diagrams*. An example is depicted in Figure 1, adopted from [23].

Constraints over UML class diagrams can be described in OCL [17, 22, 23]. We recall that OCL allows to specify two types of constraints:

- *Invariants*, i.e. statements that should be valid at any point in the computation.

$$\begin{array}{l}\mathsf{context}\ \text{Hotel}\ \ \mathsf{invariant}\\ rooms.guests = guests\end{array} \tag{1}$$

  This invariant states that the collection of guests in the rooms of the hotel should be consistent with the collection of guests maintained at the hotel. Clearly, this statement is not valid in any state of the system as, for instance, its validity cannot be guaranteed while executing a method that changes the number of guests (like including a guest in or out).

- *Pre-* and *postconditions* i.e., statements about the start and end of a method execution. Pre- and postconditions have a method and a class as context. For instance, in

$$\begin{array}{l}\mathsf{context}\ \text{Hotel} :: \text{checkIn}(g : \text{Guest})\\ \mathsf{pre}\ \ \mathsf{not}\ guests{\rightarrow}includes(g)\\ \mathsf{post}\ guests{\rightarrow}size = (guests@\mathsf{pre}{\rightarrow}size){+}1\ \mathsf{and}\ guests{\rightarrow}includes(g)\end{array} \tag{2}$$

  the precondition states that the person to be checked in is not a current guest of the hotel, while the postcondition states that after checking him in, the number of guests has increased by one and the new guest is one of the current guests.

Related work. Logics for reasoning about object-oriented systems have mainly based on Hoare-style logics that concentrate on verifying pre- and postconditions and/or invariants [1, 4, 14, 18]. Temporal logics for object-oriented systems have been previously defined by, amongst others, [3, 15, 21]. A modal logic for the object calculus is presented in [2]. Verification techniques based on other techniques have been proposed in [13]. To our knowledge, this paper presents the first attempt towards embedding OCL in a temporal logic setting.

2 The definition of BOTL

In the following, we assume a set VNAME of *variable names*; a set MNAME of *method names*, ranged over by M; and a set of *class names* CNAME, ranged over by C.

2.1 Data types and values

BOTL expressions rely on a language TYPE of data types, defined by the following grammar:

$$\tau(\in \textsc{Type}) ::= \mathsf{void} \mid \mathsf{nat} \mid \mathsf{bool} \mid \tau\ \mathsf{list} \mid C\ \mathsf{ref} \mid C.M\ \mathsf{ref}$$

where $C \in$ CNAME and $M \in$ MNAME are arbitrary. The types have the following intuitions:

- void is the unit type; it only has the trivial value $()$.
- nat is the type of natural numbers.
- bool is the type of boolean values tt (true) and ff (false).
- $\tau\ \mathsf{list}$ denotes the type of lists of τ, with elements $[]$ (the empty list) and $h :: w$ (for the list with head element h and tail w).
- $C\ \mathsf{ref}$ denotes the type of objects of class C.
- $C.M\ \mathsf{ref}$ denotes the type of method occurrences (discussed in more detail below) of the method M of class C.

Let us specify the data values of these types more precisely. Among others we will use (references to) *objects* and *events* as data values; the latter correspond to *method occurrences*, i.e., invocations of a given method of a given object. For this purpose, we introduce the following sets (for all $C \in$ CNAME and $M \in$ MNAME):

$$\begin{aligned} \textsc{Oid}^C &= \{C\} \times \mathbb{N} \\ \textsc{Evt}^{C,M} &= \textsc{Oid}^C \times \{M\} \times \mathbb{N}\ . \end{aligned}$$

Thus, object identities $\xi \in \textsc{Oid}^C$ correspond simply to numbered instances of the class C, whereas events $(\xi, M, j) \in \textsc{Evt}^{C,M}$ are numbered instances of the method name M, together with an explicit association to the object $\xi \in \textsc{Oid}^C$ executing the method.

The combined universe of values will be denoted VAL; the set of values of a given type $\tau \in$ TYPE is denoted $\textsc{Val}^\tau$. There exists a large number of standard boolean, arithmetic and list operations over these values, which we will use when convenient, without introducing them formally.

Finally, there is a special element $\bot \notin \textsc{Val}$ that is used to model the "undefined" value: we write $\textsc{Val}_\bot = \textsc{Val} \cup \{\bot\}$, etc. All operations are extended to $\bot$ by requiring them to be *strict* (meaning that if any operand equals $\bot$, the entire expression equals $\bot$). For instance, for lists we have $\bot :: w = \bot$ and $h :: \bot = \bot$.

2.2 Syntax of BOTL

The syntax of BOTL is built up from two kinds of terms: static expressions (for a large part inspired by OCL) and temporal formulae (largely taken from CTL). We also use a set of *logical variables* LOGVAR.

$$\begin{aligned} e(\in S_{exp}) \quad ::= \quad & z \mid e.a \mid e.\mathsf{owner} \mid e.\mathsf{return} \mid \mathsf{act}(e) \mid \omega(e, \ldots, e) \\ & \mid \mathsf{with}\ z_1 \in e\ \mathsf{from}\ z_2 := e\ \mathsf{do}\ z_2 := e \\ \phi(\in T_{exp}) \quad ::= \quad & e \mid \neg\phi \mid \phi \vee \phi \mid \forall z \in \tau : \phi \mid \mathsf{EX}\phi \mid \mathsf{E}[\phi \mathsf{U} \phi] \mid \mathsf{A}[\phi \mathsf{U} \phi] \end{aligned}$$

where $\tau \in$ TYPE, $a \in$ VNAME and $z \in$ LOGVAR. Apart from this context-free grammar, we implicitly rely on a context-sensitive *type system*, with type judgments of the form $e \in \tau$, to ensure type correctness of the expressions; its definition is outside the scope of this paper. We give an informal explanation of the BOTL constructs.

Expressions

- $z \in \textsc{LogVar}$ is a variable, bound to a value elsewhere in the expression or formula;
- $e.a$ stands for *attribute/parameter navigation*. The sub-expression e provides a reference to an object with an attribute named a or to a method occurrence with a formal parameter named a; the navigation expression denotes the value of that attribute/parameter. Navigation is extended naturally to the case where e is a *list* of references; the result of $e.a$ is then the list of $\_.a$-navigations from the elements of e.
- $e.\mathsf{owner}$ denotes the object executing the method e.
- $e.\mathsf{return}$ denotes the return value of the method denoted by e (in case the method has indeed returned a value, otherwise the result of the expression is undefined; see below).
- $\mathsf{act}(e)$ expresses that the object or method occurrence denoted by e is currently *active*. An object becomes active when it is created and remains active ever thereafter, whereas a method becomes active when it is invoked and becomes inactive again after it has returned a value.
- $\omega(e_1, \ldots, e_n)$ $(n \geqslant 0)$ denotes an application of the n-ary operator ω. Thus, ω is a syntactic counterpart to the actual boolean, arithmetic and list operations defined over our value domain. Possible values for ω include at least a conditional expression ("if-then-else") as well as an (overloaded) equality test $=^\tau$ for all $\tau \in \textsc{Type}$ (where the index τ is usually omitted).
- The with-from-do expression is inspired by the *iterate* feature of OCL. The expression binds logical variables and can therefore not be seen as an ordinary operator. Informally, $\mathsf{with}\ z_1 \in e_1\ \mathsf{from}\ z_2 := e_2\ \mathsf{do}\ z_2 := e_3$ has the following semantics: first, z_2 is initialized to e_2; then e_3 is computed repeatedly and its result is assigned to z_2 while z_1 successively takes as its value an element of the sequence e_1. A large group of OCL queries can be reduced to *iterate* expressions (and therefore to with-from-do expressions) [22].

Temporal expressions A temporal expression ϕ is built by the application of classical first order logic operators ($\neg$, $\vee$ etc.) and CTL temporal operators (AX, U, etc.); see [6]. The basic predicates are given by boolean expressions in S_{exp}. The temporal operators have the following intuition:

- $\mathsf{EX}\phi$ expresses that there is a next state in which the formula ϕ holds.
- $\mathsf{E}[\phi\mathsf{U}\psi]$ expresses that there exists a path starting from the current state along which ψ holds at a given state, and ϕ holds in every state before. The special case where ϕ equals tt (true) thus stands for the property that there is a reachable state where ψ holds; this is sometimes denoted $\mathsf{EF}\psi$ ("potentially eventually ψ"). The dual of that is denoted $\mathsf{AG}\psi$ ("invariantly ψ").
- $\mathsf{A}[\phi\mathsf{U}\psi]$ expresses that along *every* path starting from the current state, ψ holds at a given state and ϕ holds in every state before. Again, if ϕ equals tt we get the special case $\mathsf{AF}\psi$ ("ψ is inevitable") and its dual, $\mathsf{EG}\psi$ ("potentially always ψ").

Finally, we have universal (and, by duality, existential) quantification: $\forall z \in \tau\colon \phi$ expresses that the formula ϕ must hold for all instances z of the type τ. Note that $\textsc{Val}^\tau$ is infinite for most $\tau \in \textsc{Type}$, making model checking of universally quantified formulae impossible. When applying model checking to BOTL, therefore, we will have to restrict quantification to bounded cases; for instance, all *active* objects or all integers *smaller than* a given upper bound. For the purpose of this paper, however, we need not make such restrictions.

In examples, we often omit the type τ when it is clear from the context. Moreover, apart from the usual abbreviations such as $\forall z \neq e\colon \phi$ for $\forall z\colon (z \neq e) \Rightarrow \phi$, we also use

- $\forall z \in \mathsf{act}(\tau)\colon \phi$ for $\forall z \in \tau\colon \mathsf{act}(z) \Rightarrow \phi$, to quantify over all *active* objects or methods in τ;
- $\forall z \in e.M\ \mathsf{ref}\colon \phi$ (where $e \in C\ \mathsf{ref}$) for $\forall z \in C.M\ \mathsf{ref}\colon (z.\mathsf{owner} = e) \Rightarrow \phi$, to quantify over all method occurrences of a given object.

Example 2.1. Consider the OCL invariant (1). In BOTL, the same property would be expressed by

$$\begin{array}{c}\mathsf{AG}[\forall z \in \mathsf{act}(\text{Hotel ref}) : (\forall m \in z.checkIn\ \mathsf{ref} : \neg\mathsf{act}(m)) \Rightarrow \\ sort(flat(z.rooms.guests)) = sort(z.guests)].\end{array} \tag{3}$$

The function *flat* flattens nested lists; we need it because $z.rooms.guests$ is a list of lists, whereas $z.guests$ is a simple list. Note that the condition $\neg\mathsf{act}(m)$ on the occurrence m of the method *checkIn* is essential: during the execution of a *checkIn*, it is not possible to guarantee the validity of the invariant. As another example, consider the following OCL invariant:

$$\begin{array}{l}\mathsf{context}\ \text{Guest}\ \ \mathsf{invariant} \\ age \geq 18\end{array}$$

In BOTL, this will be expressed by: $\mathsf{AG}[\forall z \in \mathsf{act}(\text{Guest ref}) : z.age \geq 18]$.

2.3 The underlying operational model

In the design of our logic we have concentrated on the *essential* features of an object-based system. By this we mean that the logic can only address features, such as object attributes, that are likely to be available in any reasonable behavioral model of an object system. Accordingly, we define the semantics of BOTL using an operational model that is as "poor" as possible, i.e., includes those features addressable by the logic but no more than those. We do not go into the question how such a model is to be generated. Any richer kind of model can be abstracted to a BOTL model; thus, hopefully, the logic can be used to express properties of behavior models generated by a wide range of formalisms.

Our models are Kripke structures, i.e., tuples $\mathcal{M} = (Conf, \rightarrow)$ where *Conf* is the set of configurations (or states) over which $\rightarrow \subseteq Conf \times Conf$ defines a transition relation. On one side *Conf* describes the currently active objects: for each active object ξ, it denotes the local state of ξ, i.e. it records the values of the attributes of ξ. On the other side, *Conf* describes the currently active method occurrences. A possible instance of configuration is depicted in Figure 2. In this example, there is only one occurrence of the method *checkIn*, and it involves object g_6 (notice the dashed lines denoting that the operation is still not complete) that will be assigned to room r_3.

BOTL has a formal semantics defined on top of the operational model $\mathcal{M}$. The definition of this semantics is described in [9] and it is outside the scope of this paper.

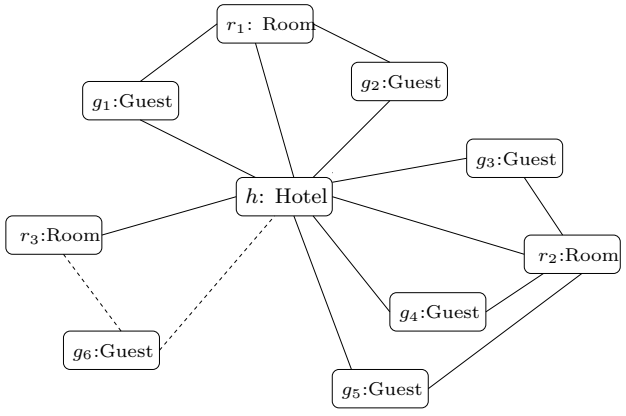

Figure 2: An instance of the Hotel Class Diagram

3 Translating OCL in BOTL

In this section we will give a summary of the translation of OCL into BOTL and investigate differences as well as relations between them. First note that BOTL is not primarily intended to be the exact formal counterpart of OCL. In defining BOTL we were concerned with some issues derived mostly from our aim to do model checking of object-oriented programs. On the other hand, mapping OCL into BOTL provides several benefit: firstly it gives a formal semantics to OCL, therefore our logic can be seen as one of the many "opinions" on how to give a sound foundation to OCL. Second, as shown in Figure 3, it allows model checking OCL constraints. Finally, the translation provides us with a feeling above the expressiveness of BOTL.

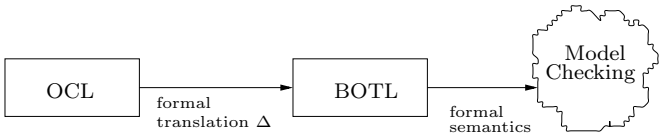

Figure 3: Model checking OCL.

3.1 OCL syntax

The set of OCL expressions is given by the following grammar

$$
\begin{array}{rcl}
(\chi \in) C_{OCL} & ::= & \mathsf{context}\ C\ \mathsf{invariant}\ e \mid \mathsf{context}\ C :: M(\vec{p})\ \mathsf{pre}\ e\ \mathsf{post}\ e \\
(e \in) S_{OCL} & ::= & \mathsf{self} \mid z \mid \mathsf{result} \mid e@\mathsf{pre} \mid e.a \mid \omega(e, \ldots, e) \\
 & & \mid e.\omega(e, \ldots, e) \mid e{\rightarrow}\omega(e, \ldots, e) \mid e{\rightarrow}\mathsf{iterate}(z_1; z_2 = e \mid e)
\end{array}
$$

As for BOTL we assume that OCL terms are type correct (with, however some differences in the possible types; see below). At the top level, a *constraint* χ can either be an invariant or a pre/postcondition (see Section 1).

Many of the expressions $e \in S_{OCL}$ have their direct counterpart in BOTL.

- self refers to the context object of the class C.
- z represents either an attribute of the context object, or a formal parameter of the context method, or a logical variable.
- result refers to the value returned by the context method. @pre is a suffix that refers to the value of its operand at the time of the method invocation. These two operators can be used only in postconditions (see below).
- $e.a$ and $\omega(e_1, \ldots, e_n)$ the same as in BOTL.
- $e.\omega(e_1, \ldots, e_n)$ represents an operator on basic types that is applied on $e, e_1, \ldots, e_n$. If the expression e is a collection (i.e. a set, bag or list), we have the special case $e{\rightarrow}\omega(e_1, \ldots, e_n)$.
- $e_1{\rightarrow}\mathsf{iterate}(z_1; z_2 = e_2 \mid e_3)$ has the same meaning as with $z_1 \in e_1$ from $z_2 := e_2$ do $z_2 := e_3$. The difference is only in the type that can be returned, namely sets and bags (see Section 3.2).

Particular OCL features not included in the previous syntax are expressions of the kind $M(e, \ldots, e)$ and $e.M(e, \ldots, e)$ where M is a so-called *query method*; i.e., M is a method which returns a value without side effects. Nevertheless, also constraints where query methods appear can be translated, in terms of another OCL expression that does not contain them but that describes the function implemented by query method[1].

[1]Provided the function is not defined recursively.

3.2 Translation issues

Before proceeding with a summary of the formal translation of OCL into BOTL, let us give the intuition, sometimes in a rather informal way, of the solutions to the issues involved. The detailed translation is presented in [9]. In particular, we define a function δ from S_{OCL} to S_{exp} and a function Δ from C_{OCL} to T_{exp}.

Data types One of the differences between BOTL and OCL is their type system: rather than arbitrary lists, OCL allows sets, bags and lists of primitive data values; i.e., *nested* lists are not included. There are two reasons why in BOTL we consider only arbitrary lists. On one side, lists have enough expressive power to represent sets and bags; on the other side, using only lists, we avoid the problem of *nondeterminism*[2].

Therefore apart from strings, reals and enumerations which are absent in BOTL but could be added without any change in the logic, we define a mechanism to translate OCL operations on sets and bags. We provide a way to shows in which sense this mechanism is faithful.

Invariants The key issue for the translation of $\mathsf{context}\ C\ \mathsf{invariant}\ e$, concerns the states in which the invariant expression e has to hold. In particular we have to assure that none of the methods of C is active. In fact, during the execution of methods, there can be some intermediate configurations in which e does not hold (see Example 3.1).

The translation has the typical prefix AG. Let $y \in \textsc{LogVar}$ and let $\{M_1, \ldots, M_k\}$ being the methods of the class C. We define:

$$\begin{aligned}&\Delta(\mathsf{context}\ C\ \mathsf{invariant}\ e) = \\ &\quad \mathsf{AG}[\forall z \in \mathsf{act}(C\ \mathsf{ref}) : \forall m_1 \in z.M_1\ \mathsf{ref} : \ldots : \forall m_k \in z.M_k\ \mathsf{ref} : \\ &\qquad\qquad (\neg\mathsf{act}(m_1) \wedge \ldots \wedge \neg\mathsf{act}(m_k)) \Rightarrow \delta_{z,y,[]}(e)].\end{aligned}$$

Pre/postconditions The translation of pre/postconditions is more involved. In particular, the OCL operator @pre has to be handled in a special way as it forces us to consider two different moments in time, viz. the start and end of a method invocation. We use the following strategy. Consider the following constraint: $\mathsf{context}\ C :: M(\vec{p})\ \mathsf{pre}\ e_{\mathsf{pre}}\ \mathsf{post}\ e_{\mathsf{post}}$. By definition, $e@\mathsf{pre}$ subexpressions occur a finite number of times, say $n \geq 0$, only in e_{post}. We first enumerate all the occurrences of $e@\mathsf{pre}$ subexpressions in e_{post}. We write $e@_{\mathsf{i}}\mathsf{pre}$ for $1 \leq i \leq n$. Then when we translate e_{post}, we substitute terms $e@_{\mathsf{i}}\mathsf{pre}$ by new fresh logical variables u_i for $1 \leq i \leq n$. The value of the variables u_i is bound to the appropriate value in the translation of e_{pre}: we "add" to the translated precondition $\delta(e_{\mathsf{pre}})$ a binding term $u_i = \delta(e)$ for all u_i and $e@_{\mathsf{i}}\mathsf{pre}$. Thus, the variables u_i are associated to the value of e in $e@_{\mathsf{i}}\mathsf{pre}$ at the beginning of the method execution, and therefore can be used instead of $e@_{\mathsf{i}}\mathsf{pre}$ in the postcondition.

Formally, consider the OCL expression $\mathsf{context}\ C :: M(\vec{p})\ \mathsf{pre}\ e_{\mathsf{pre}}\ \mathsf{post}\ e_{\mathsf{post}}$. The *extended* translated precondition e^+_{pre} is given by

$$e^+_{\mathsf{pre}} \triangleq \delta(e_{\mathsf{pre}}) \wedge \bigwedge_{e@_{\mathsf{i}}\mathsf{pre} \in e_{\mathsf{post}}} (u_i = \delta(e))$$

where u_i for $1 \leq i \leq n$ are fresh logical variables.

Here the symbol $\in$ means "occurs syntactically in". Now we can map OCL pre/postconditions into BOTL.

$$\begin{aligned}&\Delta(\mathsf{context}\ C :: M(\vec{p})\ \mathsf{pre}\ e_{\mathsf{pre}}\ \mathsf{post}\ e_{\mathsf{post}}) = \\ &\quad \forall u_1 \in \tau_1, \ldots, u_n \in \tau_n : \forall z \in \mathsf{act}(C\ \mathsf{ref}) : \forall m \in z.M\ \mathsf{ref} : \\ &\qquad \mathsf{AG}[(e^+_{\mathsf{pre}} \wedge \neg\mathsf{act}(m)) \Rightarrow \\ &\qquad\qquad \mathsf{AX}[\mathsf{act}(m) \Rightarrow \mathsf{A}[\mathsf{act}(m)\mathsf{U}(\mathsf{term}(m) \wedge \delta(e_{\mathsf{post}}))]]\end{aligned}$$

[2]This problem is manifest in in OCL. In fact it suffers of nondeterminism in the iterate expression and in the flattening of collections.

where $\mathsf{term}(m) \equiv \mathsf{act}(m) \wedge \mathsf{EX}[\neg\mathsf{act}(m)]$.

The expressions e^+_{pre} and e_{post} are embedded in a kind of "template" scheme. Intuitively, a pre/postcondition holds if and only if for all invocations m of M executed by an object of the class C we have that: if the (extended) precondition holds at the moment of the method call, then the postcondition holds when the method execution terminates. This must be true for all active objects of C and all possible executions of the method M. In other words a pre/postcondition is actually an invariant on method calls.

Example 3.1. Suppose we want to translate pre/postcondition (2) in Section 1. Again, let us call the precondition e_{pre} and the postcondition e_{post}. Consider two logical variables z and m. The former will be instantiated with an object of class Hotel and the latter with an occurrence of the method checkIn. Applying δ to e_{pre} yields: $\delta(\mathsf{not}\ \ guests{\rightarrow}includes(g)) = \neg\overline{includes}(z.guests, m.g)$, where $\overline{includes}$ is a BOTL operation that, given a list w and an element l, returns tt if and only if the element l belongs to w. The extended precondition becomes:

$$e^+_{\mathsf{pre}} \quad \equiv \quad \neg\overline{includes}(z.guests, m.g) \wedge u_1 = z.guests$$

The translation of the postcondition is:

$$\delta(e_{\mathsf{post}}) \quad = \quad (\overline{size}(z.guests) = \overline{size}(u_1) + 1 \wedge \overline{includes}(z.guests, m.g)).$$

The translation of (2) now yields:

$$\begin{aligned}\forall u_1 \in \text{Guest } \mathsf{ref\ list} : \forall z \in \mathsf{act}(\text{Hotel } \mathsf{ref}) : \forall m \in z.checkIn\ \mathsf{ref} : \mathsf{AG}[(e^+_{\mathsf{pre}} \wedge \neg\mathsf{act}(m)) \\ \Rightarrow \mathsf{AX}[\mathsf{act}(m) \Rightarrow \mathsf{A}[\mathsf{act}(m)\mathsf{U}(\mathsf{term}(m) \wedge \delta(e_{\mathsf{post}}))].\end{aligned}$$

Figure 4 describes the configurations of the transition system during the execution of the method *checkIn* and how the validity of pre/postcondition changes. In configuration 1 object g_1 does not belong to the guests of h. The set of method calls is empty. In this state $\neg\mathsf{act}(checkIn)$ and the precondition e_{pre} are valid. In configuration 2, the method is active and, as a first step, g_1 is inserted among z guests. Thus e_{pre} does not hold anymore. However, from this state e_{post} becomes valid. In configuration 3, g_1 is assigned to room r and the method execution ends. Finally in configuration 4, checkIn is not active anymore, and the postcondition e_{post} still holds. Notice how in this example it becomes clear why the invariant (1) does not hold during the execution of *checkIn*.

References

[1] M. Abadi and K.R.M. Leino. A logic of object-oriented programs. In *Theory and Practice of Software Development (TAPSOFT)*, LNCS 1214, pp. 682–696, 1997.

[2] D.S. Andersen, L.H. Pedersen, H. Hüttel and J. Kleist. Objects, types and modal logics. In *Foundations of Object-Oriented Languages (FOOL)*, 1997.

[3] J.-P. Bahsoun, R. El-Baida, and H.-O. Yar. Decision procedure for temporal logic of concurrent objects. In *EuroPar'99*, LNCS 1685, pp. 1344–1352, Springer, 1999.

[4] F.S. de Boer. A proof system for the parallel object-oriented language POOL. In *Automata, Languages, and Programming (ICALP)*, LNCS 443, pp. 572–585, 1990.

[5] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.

[6] E.M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs*, LNCS 131, pp. 52–71, Springer, 1981.

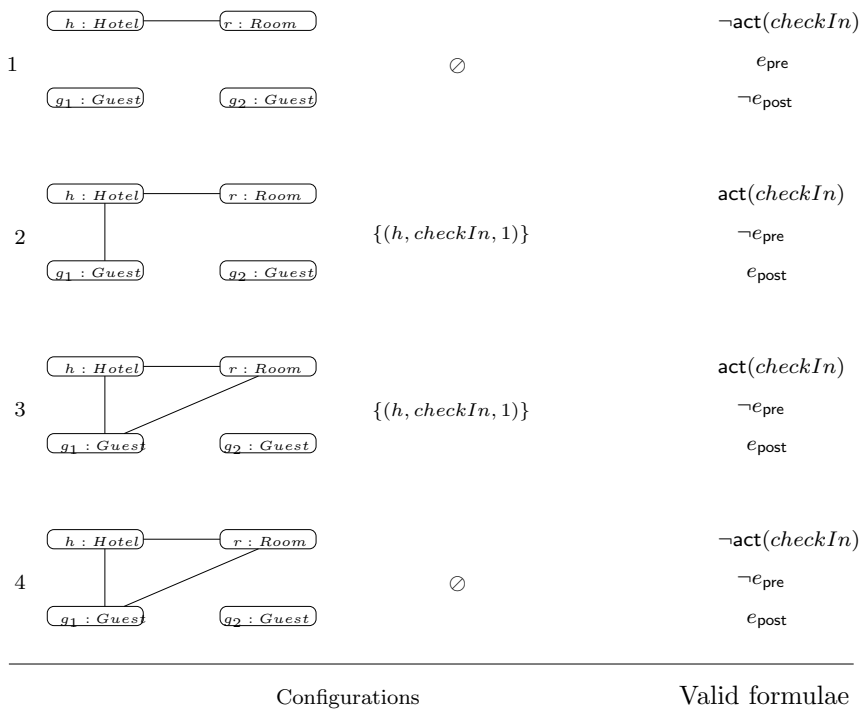

Figure 4: Configurations during the execution of checkIn(g_1).

[7] E.M. Clarke and R. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[8] E.M. Clarke, O. Grumberg and D. Peled. *Model Checking*. MIT Press, 1999.

[9] D. Distefano, J.-P. Katoen, and A. Rensink. On a temporal logic for object-based systems. Technical report CTIT 00-06, University of Twente, 2000.

[10] M. Gogolla and M. Richters. On constraints and queries in UML. In *The Unified Modeling Language – Technical Aspects and Applications*, Physica-Verlag, 1998.

[11] A. Hamie, F. Civello, J. Howse, S. Kent, and R. Mitchell. Reflections on the Object Constraint Language. In *The Unified Modeling Language (UML)*, LNCS, pp. 137–145, Springer, 1998.

[12] A. Hamie, J. Howse, and S. Kent. Interpreting the Object Constraint Language. In *Asia Pacific Software Engineering Conference*, pp. 288–295. IEEE CS Press, 1998.

[13] S.J. Hodges and C. B. Jones. Non-interference properties of a concurrent object-based language: Proofs based on an operational semantics. In *Object Orientation with Parallelism and Persistence*, pp. 1–22, Kluwer, 1996.

[14] K. Huizing and R. Kuiper and SOOP. Verification of object-oriented programs using class invariants. In *Fundamental Approaches to Software Eng. (FASE)*, LNCS, Springer 2000 (to appear).

[15] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL – a language for object-oriented specification of information systems. *ACM Trans. on Inf. Sys.*, 14(2):175–211, 1996.

[16] L. Mandel and M. V. Cengarle. On the expressive power of the Object Constraint Language OCL. Technical report, Forschungsinstitut für angewandte Software-Technologie (FAST e.V.), 1999.

[17] Rational Software Corporation. *Object Constraint Language Specification, version 1.1*, 1997. (available from `www.rational.com/uml`).

[18] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In *Programming Concepts and Methods (PROCOMET)*, pp. 404–424, Kluwer, 1998.

[19] M. Richters and M. Gogolla. On formalizing the UML object constraint language OCL. In *Conceptual Modeling (ER'98)*, LNCS 1507, pp. 449–464, Springer, 1998.

[20] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Guide*. Addison-Wesley, 1998.

[21] A. Sernadas, C. Sernadas, and J.F. Costa. Object specification logic. *J. of Logic and Computation*, 5(5):603–630, 1995.

[22] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[23] J. Warmer and A. Kleppe. OCL: The constraint language of the UML. *J. of Obj.-Or. Progr.*, 12(1):10–13&28, 1999.