On a Temporal Logic for Object-Based Systems

DINO DISTEFANO, JOOST-PIETER KATOEN AND AREND RENSINK

Faculty 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), an optional part of the UML standard for expressing static properties over class diagrams. 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.

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

1 Introduction

Due to the ever increasing complexity of forthcoming 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 their 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 behaviour 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) [16, 21, 22], an optional part of the Universal Modelling Language (UML) [5, 19] standard which allows expressing static properties over class diagrams in a textual way. The precise relationship with OCL is defined by means of a mapping of a large fragment of OCL 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. 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. This approach resolves several ambiguities and unclearities in OCL that have been recently reported [10]. (Alternative formalisations of OCL have been considered in [9, 11, 18].) Our proposal covers a rather large fragment of OCL including, amonst others, invariants, pre- and postconditions, navigations and iterations.

Object-based systems. In this paper we confine ourselves to *object-based* systems, i.e., object-oriented systems in which inheritance and subtyping are not (yet) considered. Object-based systems are composed by *objects*. An object contains internal data that can only be accessed from the outside by invoking one of the object's methods. Objects run concurrently and communicate by means of message passing; i.e., an object that invokes a method (of another object) has to wait until the method has returned its result. Objects are dynamic and can be created in arbitrary numbers during the computation. On the static level, the corresponding notion is that of a class. A *class* is a template for the creation of its instances, i.e., its objects, and specifies the behaviour of the objects by describing their state (in terms of so-called attributes) and methods.

Class diagrams. Classes and their associations are described by *class diagrams*, a variant of entity-relationship diagrams. A class diagram describes the attributes (with their type) and the methods (with their formal parameters) of a class. An example class diagram in UML notation is depicted in Figure 1, adopted from [22]: boxes represent classes and interconnecting lines denote associations. Each direction of an association has a multiplicity and an optional name. For instance, a Hotel has a (possibly zero) number of *rooms* and *guests*. Note that class diagrams only address the data aspects of the system, not its dynamic (i.e., process) aspects. The latter aspects are described by other diagrams such as UML statecharts. Associations can be traversed — this is referred to as *navigation* — to refer to attributes and methods of a (collection of) object(s) in the system, e.g., for object h of class Hotel, the expression $(h.guests).name$ refers to the collection of names of the guests of h. Navigations are parsed from left to right.

Object Constraint Language. *Constraints* over UML class diagrams can be described in OCL [16, 21, 22], an optional part of the UML standard. Two prominent constraints in OCL are invariants (statements that should be valid at any point in the computation), and pre- and postconditions (statements about the start and end of a method execution). The invariant

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

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

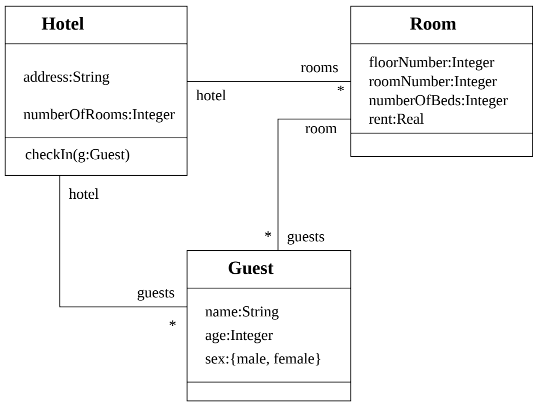


Figure 1: The Hotel Class Diagram

have a method and a class as context. For instance, in

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

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. The @pre-operator refers to the number of guests before the check-in. The standard OCL operation *size* determines the number of elements of a collection.

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, 13, 17]. Temporal logics for object-oriented systems have been previously defined by, amongst others, [3, 14, 20]. A modal logic for the object calculus is presented in [2]. Verification techniques based on other techniques have been proposed in [12]. To our knowledge, this paper presents the first attempt towards embedding OCL in a temporal logic setting.

Organisation of the paper. Section 2 introduces the syntax and semantics of BOTL. Section 3 presents the translation of OCL into BOTL. Throughout the paper simple examples illustrate the use of the logic. Section 4 presents some conclusions and discusses future work.

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 $\textsc{Type}$ 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 $C \in \textsc{CName}$ and $M \in \textsc{MName}$ 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). For the sake of readability, we will often write lists as comma-separated sequences enclosed by square brackets; e.g., $1 :: 2 :: []$ is written $[1, 2]$, whereas $[[1], 2]$ denotes $(1 :: []) :: 2 :: []$.
- $C\ \mathsf{ref}$ 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 $C \in \textsc{CName}$ and $M \in \textsc{MName}$):

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

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. We also use $\textsc{OId} = \bigcup_C \textsc{OId}^C$, ranged over by ξ, and $\textsc{Evt} = \bigcup_C \bigcup_M \textsc{Evt}^{C,M}$, ranged over by μ.

The combined universe of values will be denoted $\textsc{Val}$; the set of values of a given type $\tau \in \textsc{Type}$ is denoted $\textsc{Val}^\tau$. We define:

$$\begin{aligned} \textsc{Val}^{\mathsf{void}} &= \{()\} \\ \textsc{Val}^{\mathsf{nat}} &= \mathbb{N} \\ \textsc{Val}^{\mathsf{bool}} &= \{\mathsf{ff}, \mathsf{tt}\} \\ \textsc{Val}^{\tau\ \mathsf{list}} &= \{[]\} \cup \{h :: w \mid h \in \textsc{Val}^\tau, w \in \textsc{Val}^{\tau\ \mathsf{list}}\} \\ \textsc{Val}^{C\ \mathsf{ref}} &= \{\mathsf{null}\} \cup \textsc{OId}^C \\ \textsc{Val}^{C.M\ \mathsf{ref}} &= \textsc{Evt}^{C,M}\ . \end{aligned}$$

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

2.2 Syntax of BOTL

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

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

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 judgements 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

- $z \in$ 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. This is made more precise in the semantic model; see below.
- $\omega(e_1,\ldots,e_n)$ $(n \geqslant 0)$ 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 $\tau \in$ Type (where the index τ is usually omitted). We will use $[\![\omega]\!]$ to indicate the underlying operation of which ω is the syntactic representation.
- The with-from-do expression is inspired by the *iterate* feature of OCL —which in turn resembles the *fold* operation of functional programming. 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 initialised 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. For instance, the expression

$$\mathsf{with}\ z_1 \in [1,2,3]\ \mathsf{from}\ z_2 := 0\ \mathsf{do}\ z_2 := z_1 + z_2$$

computes the sum of the elements of the list $[1,2,3]$ $(=6)$. A large group of OCL queries can be reduced to *iterate* expressions (and therefore to with-from-do expressions) [21].

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

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

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:

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

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

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 behavioural model of an object system. Accordingly, we will 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 behaviour models generated by a wide range of formalisms.

We first need to give the notions of classes, methods and variables more substance. Consider the following partial functions:

$$\begin{array}{rcl}\text{VDecl} & = & \text{VName} \rightharpoonup \text{Type} \\ \text{MDecl} & = & \text{MName} \rightharpoonup \text{VDecl} \times \text{Type} \\ \text{CDecl} & = & \text{CName} \rightharpoonup \text{VDecl} \times \text{MDecl}\end{array}$$

A variable declaration in VDecl is a partial function mapping variable names to the corresponding (image) types. MDecl does the same for method names, taking into account that these are actually functions with formal parameters and a return value. Finally, each $D \in$ CDecl is a class declaration mapping class names to the corresponding atribute and method declarations.

Let us assume the class declaration $D \in$ CDecl to be given. For any class $C \in \text{dom}(D)$, we denote $C.attrs$ ($\in$ VDecl) for its attribute declaration function, and $C.meths$ ($\in$ MDecl) for its method declaration function; thus, $D(C) = (C.attrs, C.meths)$. Furthermore, if the class C of a method M is clear from the context then we use $M.fpars$ ($\in$ VDecl) to denote the formal parameters of M and $M.retty$ ($\in$ Type) for the return type; hence $C.meths(M) = (M.fpars, M.retty)$.

Our models are Kripke structures, i.e., tuples $\mathcal{M}_D = (Conf, \rightarrow)$ where *Conf* is the set of configurations (or states) over which $\rightarrow \subseteq Conf \times Conf$ defines a transition relation. $D \in$ CDecl is the global class declaration, whereas the elements of *Conf* are pairs of the form (σ, γ) where:

- $\sigma \in \Sigma = \text{OId} \rightharpoonup \text{VName} \rightharpoonup \text{Val}$;
- $\gamma \in \Gamma = \text{Evt} \rightharpoonup (\text{VName} \rightharpoonup \text{Val}) \times \text{Val}_\perp$.

We discuss these briefly.

- σ describes the currently active objects: for each active object $\xi \in \text{dom}(\sigma)$, $\sigma(\xi)$ denotes the local state of ξ, i.e. it records the values of the attributes of ξ. σ has to be consistent with D in the sense that $\sigma(\xi) = \ell$ with $\xi \in \text{OId}^C$ implies $\text{dom}(\ell) = \text{dom}(C.attrs)$ and $\ell(a) \in \text{Val}^{C.attrs(a)}$ for all $a \in \text{dom}(\ell)$.

  σ is extended pointwise to *lists* of objects; thus $\sigma([])(a) = []$ and $\sigma(h :: w)(a) = \sigma(h)(a) :: \sigma(w)(a)$.

- γ describes the currently active method occurrences; namely, an event is active if it belongs to the domain of γ. The images of γ consist of a (partial) mapping of variable names to values, representing the valuation of the formal parameters of the method invocation, as well as the value returned by the method. The latter becomes defined only when the method has terminated; therefore the value can be $\bot$. γ also has to be consistent with D: if $\gamma(\mu) = (\ell, v)$ for a given method occurrence $\mu \in \textsc{Evt}^{C,M}$ then $\mathrm{dom}(\ell) = \mathrm{dom}(M.\mathit{fpars})$ and $\ell(p) \in \textsc{Val}^{M.\mathit{fpars}(p)}$ for all $p \in \mathrm{dom}(\ell)$, and $v \in \textsc{Val}_{\bot}^{M.\mathit{retty}}$.

- The transition relation $\rightarrow$ satisfies the following property regarding the termination of method invocations: if an active method occurrence μ becomes inactive then is has a well-definded return value (i.e., not $\bot$). Formally: if $(\sigma, \gamma) \rightarrow (\sigma', \gamma')$ then $\mu \in \mathrm{dom}(\gamma) \setminus \mathrm{dom}(\gamma') \Rightarrow \exists v \neq \bot\colon \gamma(\mu) = (\ell, v)$. Furthermore, we assume that *Conf* contains no terminated or deadlocked configurations; i.e., there is at least one outgoing transition from every element of *Conf*. (This property is imposed only for the sake of simplifying the definitions later on; it can be satisfied easily by adding a self-loop to every deadlocking configuration.)

2.4 Semantics of BOTL

We are now in a position to define the semantics of our logic. We assume the class declaration D to be fixed and given. Let $\Theta = \textsc{LogVar} \rightharpoonup \textsc{Val}$, ranged over by θ, be the set of maps that assign values to (some of) the logical variables. The semantics of expressions is given by the function $[\![\_]\!]\colon S_{exp} \rightarrow (\Sigma \times \Gamma \times \Theta) \rightarrow \textsc{Val}_{\bot}$. Let (σ, γ) be a configuration of $\mathcal{M}_D$.

$$
\begin{array}{lcll}
[\![z]\!]_{\sigma,\gamma,\theta} & = & \theta(z) & \\
[\![e.a]\!]_{\sigma,\gamma,\theta} & = & \ell(a) & \text{where } e \in C \ \mathsf{ref} \text{ and } \sigma([\![e]\!]_{\sigma,\gamma,\theta}) = \ell \\
 & & & \text{or } e \in C.M \ \mathsf{ref} \text{ and } \gamma([\![e]\!]_{\sigma,\gamma,\theta}) = (\ell, v) \\
 & = & \vec{\ell}(a) & \text{where } e \in C \ \mathsf{ref\ list} \text{ and } \sigma([\![e]\!]_{\sigma,\gamma,\theta}) = \vec{\ell} \\
 & & & \text{or } e \in C.M \ \mathsf{ref\ list} \text{ and } \gamma([\![e]\!]_{\sigma,\gamma,\theta}) = (\vec{\ell}, \vec{v}) \\
[\![e.\mathsf{owner}]\!]_{\sigma,\gamma,\theta} & = & \xi & \text{where } [\![e]\!]_{\sigma,\gamma,\theta} = (\xi, M, j) \\
[\![e.\mathsf{return}]\!]_{\sigma,\gamma,\theta} & = & v & \text{where } \gamma([\![e]\!]_{\sigma,\gamma,\theta}) = (\ell, v) \\
[\![\mathsf{act}(e)]\!]_{\sigma,\gamma,\theta} & = & \multicolumn{2}{l}{([\![e]\!]_{\sigma,\gamma,\theta} \in \mathrm{dom}(\sigma) \cup \mathrm{dom}(\gamma))} \\
[\![\omega(e_1, \ldots, e_n)]\!]_{\sigma,\gamma,\theta} & = & \multicolumn{2}{l}{[\![\omega]\!]([\![e_1]\!]_{\sigma,\gamma,\theta}, \ldots, [\![e_n]\!]_{\sigma,\gamma,\theta})} \\
\multicolumn{4}{l}{[\![\mathsf{with}\ z_1 \in e_1\ \mathsf{from}\ z_2 := e_2\ \mathsf{do}\ z_2 := e_3]\!]_{\sigma,\gamma,\theta}} \\
 & = & \multicolumn{2}{l}{[\![\mathsf{for}\ z_1 \in [\![e_1]\!]_{\sigma,\gamma,\theta}\ \mathsf{do}\ z_2 := e_3]\!]_{\sigma,\gamma,\theta\{[\![e_2]\!]_{\sigma,\gamma,\theta}/z_2\}}}
\end{array}
$$

$$
\begin{array}{ll}
\text{where} & [\![\mathsf{for}\ z_1 \in []\ \mathsf{do}\ z_2 := e]\!]_{\sigma,\gamma,\theta} \\
 & \qquad = [\![z_2]\!]_{\sigma,\gamma,\theta} \\
 & [\![\mathsf{for}\ z_1 \in h :: w\ \mathsf{do}\ z_2 := e]\!]_{\sigma,\gamma,\theta} \\
 & \qquad = [\![\mathsf{for}\ z_1 \in w\ \mathsf{do}\ z_2 := e]\!]_{\sigma,\gamma,\theta\{[\![e]\!]_{\sigma,\gamma,\theta\{h/z_1\}}/z_2\}}
\end{array}
$$

Given the discussion of the operational model, the semantics should be self-explanatory, with the possible exception of the "with-from-do"-expression. This is evaluated by means of the "for-do" meta-expression, which successively re-computes the "do"-expression for every value of z_1 out of the "for"-list.[1]

[1]For those familiar with functional programming: $\mathsf{with}\ z_1 \in e_1\ \mathsf{from}\ z_2 := e_2\ \mathsf{do}\ z_2 := e_3$ may alternatively be translated to *foldl* $[\![e_1]\!]_{\sigma,\gamma,\theta}\ [\![e_2]\!]_{\sigma,\gamma,\theta}\ \lambda\, v\, h. [\![e_3]\!]_{\sigma,\gamma,\theta\{h/z_1, v/z_2\}}$.

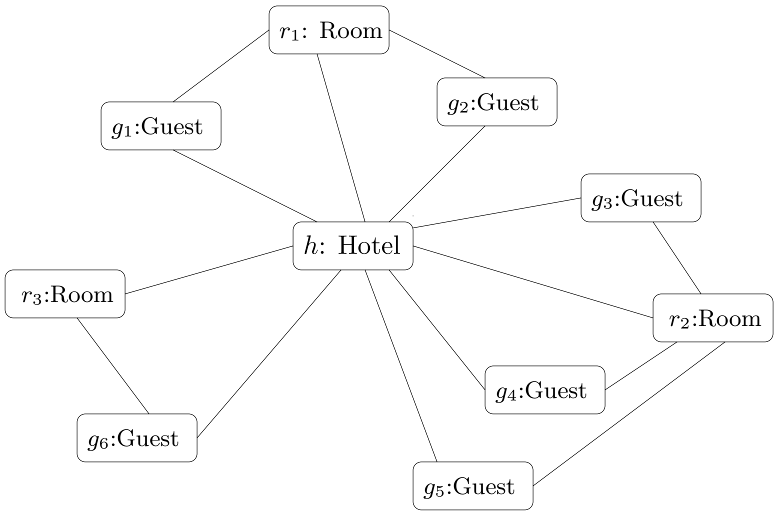


Figure 2: An instance of the Hotel Class Diagram

Example 2.2. Consider the object diagram in Figure 2, and suppose we want to compute $z.rooms.guests$ in the configuration (σ, γ) with variable interpretation $\theta\colon z \mapsto h$. Skipping some details, we obtain

$$\begin{aligned}
[\![z.rooms.guests]\!]_{\sigma,\gamma,\theta} &= \sigma([\![z.rooms]\!]_{\sigma,\gamma,\theta})(guests) \\
&= \sigma(\sigma([\![z]\!]_{\sigma,\gamma,\theta})(rooms))(guests) \\
&= \sigma(\sigma(h)(rooms))(guests) \\
&= \sigma([r_1, r_2, r_3])(guests) \\
&= [[g_1, g_2], [g_3, g_4, g_5], [g_6]] \ .
\end{aligned}$$

As expected the result is a list of lists.

The semantics of BOTL formulae is now straightforward. It is defined by a satisfaction relation between the model $\mathcal{M}_D$ defined by the transition system, a reference configuration (σ, γ), a valuation θ and a formula ϕ. To define it, we need an auxiliary definition of *paths* through a transition model. Some notation first: if $s \in A^\omega$ is an (infinite) sequence, we write $s[i]$ to denote the $(i+1)$-th element of s; hence $s = s[0]s[1]\cdots$. Given a model $\mathcal{M}_D = (\mathit{Conf}, \rightarrow)$, a *path* is an infinite sequence of configurations $\eta \in \mathit{Conf}^\omega$ such that $\eta[i] \rightarrow \eta[i+1]$ for all $i \geq 0$. The set of paths starting from $(\sigma, \gamma) \in \mathit{Conf}$ is defined by

$$P_{\mathcal{M}_D}(\sigma, \gamma) = \{\eta \in \mathit{Conf}^\omega \mid \eta[0] = (\sigma, \gamma), \forall i \geqslant 0\colon \eta[i] \rightarrow \eta[i+1]\} \ .$$

The semantics of temporal formulae is then given by a relation $\models \subseteq (\Sigma \times \Gamma \times \Theta) \times T_{exp}$.

Let (σ, γ) be a configuration of $\mathcal{M}_D$ and let $\theta \in \Theta$.

$$\begin{array}{lcl}
\sigma, \gamma, \theta \models e & \iff & [\![e]\!]_{\sigma,\gamma,\theta} = \mathsf{tt} \\
\sigma, \gamma, \theta \models \neg\phi & \iff & \neg(\sigma, \gamma, \theta \models \phi) \\
\sigma, \gamma, \theta \models \phi \vee \psi & \iff & (\sigma, \gamma, \theta \models \phi) \vee (\sigma, \gamma, \theta \models \psi) \\
\sigma, \gamma, \theta \models \forall z \in \tau : \phi & \iff & \sigma, \gamma, \theta\{v/z\} \models \phi \text{ for all } v \in \mathrm{V{\scriptsize AL}}^{\tau} \\
\sigma, \gamma, \theta \models \mathsf{EX}\phi & \iff & \exists \eta \in P_{\mathcal{M}_D}(\sigma, \gamma) \colon \eta[1], \theta \models \phi \\
\sigma, \gamma, \theta \models \mathsf{E}[\phi \mathsf{U} \psi] & \iff & \exists \eta \in P_{\mathcal{M}_D}(\sigma, \gamma) \colon \\
& & \exists j \geq 0 \colon \eta[j], \theta \models \psi \wedge \forall 0 \leq k < j \colon \eta[k], \theta \models \phi \\
\sigma, \gamma, \theta \models \mathsf{A}[\phi \mathsf{U} \psi] & \iff & \forall \eta \in P_{\mathcal{M}_D}(\sigma, \gamma) \colon \\
& & \exists j \geq 0 \colon \eta[j], \theta \models \psi \wedge \forall 0 \leq k < j \colon \eta[k], \theta \models \phi \ .
\end{array}$$

3 Translating OCL in BOTL

In this section we will give a 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, since OCL is not yet very "stable" in the sense that there are many proposals to improve it, see e.g. [10], our logic can be seen as one of the many "opinions" on how to give a sound foundation to OCL. At the same time, the translation provides us with a feeling above the expressiveness of BOTL.

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). The context of a constraint is a class C in case of invariant or a method $M \in \mathit{dom}(C.\mathit{meths})$ in case of pre/postconditions. The context can be referred to by the expression in the constraint. For instance, in an OCL navigation expression self.a, we describe a route starting from an object of the context class C.

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 $\mathsf{with}\ z_1 \in e_1\ \mathsf{from}\ z_2 := e_2\ \mathsf{do}\ z_2 := e_3$. 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[2]. Thus, as in other related works [11, 18], we do not treat query method explicitly.

3.2 Translation issues

Before proceeding with in the formal translation of OCL into BOTL, let us give the intuition, in a rather informal way, of the solutions to the issues involved.

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 *nondeterministic* behavior in the BOTL expression with-do-from.

In order to have a more rigorous comparison, let us define OCL types. Then we will show how to encode them using BOTL types. We omit strings, reals and enumerations which are absent in BOTL but could be added without problems.

$$\begin{array}{rcl} \rho & ::= & \mathsf{nat} \mid \mathsf{bool} \mid C\ \mathsf{ref} \\ \tau (\in \textsc{Type}_{OCL}) & ::= & \rho \mid \rho\ \mathsf{list} \mid \rho\ \mathsf{set} \mid \rho\ \mathsf{bag} \end{array}$$

$\rho\ \mathsf{set}$ includes sets of element of type ρ, while $\rho\ \mathsf{bag}$ are multisets whose elements have type ρ. The semantics of those sorts included in $\textsc{Type}$ is unchanged, while for the new types we have:

$$\begin{array}{rcl} \textsc{Val}^{\rho\ \mathsf{set}} & = & \mathcal{P}(\textsc{Val}^{\rho}) \\ \textsc{Val}^{\rho\ \mathsf{bag}} & = & \textsc{Val}^{\rho} \longrightarrow \mathbb{N} \end{array}$$

where $\mathcal{P}(\cdot)$ represents the set of all finite subsets. The set of values in OCL is:

$$\textsc{Val}_{OCL} = \bigcup_{\tau \in \textsc{Type}_{OCL}} \textsc{Val}^{\tau}.$$

[2]Provided the function is not defined recursively.

Now, let us discuss how we will translate OCL operations on sets and bags, say $e_1{\rightarrow}\omega(e_2, \ldots, e_n)$. For OCL types $\rho\,\mathsf{set}$ and $\rho\,\mathsf{bag}$, we define a function α_{set} and α_{bag} on BOTL values. These functions abstract from the order of the elements in a list and return a set or a bag. Formally $\alpha_{\mathsf{set}} : \textsc{Val} \longrightarrow \textsc{Val}_{OCL}$ is given by

$$\alpha_{\mathsf{set}}(v) = \begin{cases} \varnothing & \text{if } v = [] \\ \{h\} \cup \alpha_{\mathsf{set}}(w) & \text{if } v = h :: w \\ v & \text{otherwise.} \end{cases}$$

Using $\{\!|\cdot|\!\}$ as notation for bags and $\uplus$ for their union, $\alpha_{\mathsf{bag}} : \textsc{Val} \longrightarrow \textsc{Val}_{OCL}$ is given by

$$\alpha_{\mathsf{bag}}(v) = \begin{cases} \{\!||\!\} & \text{if } v = [] \\ \{\!|h|\!\} \uplus \alpha_{\mathsf{bag}}(w) & \text{if } v = h :: w \\ v & \text{otherwise.} \end{cases}$$

For each operation $e_1{\rightarrow}\omega(e_2, \ldots, e_n)$ on sets or bags, there exists a corresponding operation in BOTL, say $\bar{\omega}(e_1, e_2, \ldots, e_n)$, such that the diagram in Figure 3 commutes. This shows in

$$\begin{tikzcd} \textsc{Val}^n_{OCL} \arrow[r, "{[\![\omega]\!]}"] & \textsc{Val}_{OCL} \\ \textsc{Val}^n \arrow[u, "\alpha"] \arrow[r, "{[\![\bar{\omega}]\!]}"'] & \textsc{Val} \arrow[u, "\alpha"'] \end{tikzcd}$$

Figure 3: Commutative diagram.

which sense the translation of OCL into BOTL is faithful.

Example 3.1. Take e.g. the OCL expression $e_1{\rightarrow}union(e_2)$. The intended semantics $[\![union]\!]$ is the mathematical union on sets. In BOTL, there will be an appropriate operator with semantics $[\![\overline{union}]\!] : \textsc{Val}^{\tau\ \mathsf{list}} \times \textsc{Val}^{\tau\ \mathsf{list}} \longrightarrow \textsc{Val}^{\tau\ \mathsf{list}}$. According to the commutative diagram, we have that $\alpha_{\mathsf{set}}([\![\overline{union}(v_1, v_2)]\!]) = [\![union]\!](\alpha_{\mathsf{set}}(v_1), \alpha_{\mathsf{set}}(v_2))$. i.e., the result on lists is equal, up to abstracting from sets, to the corresponding union on sets. The operator $\overline{union}$ can be defined for instance as $\overline{union}(w_1, w_2) \triangleq concat(w_1, w_2)$ where w_1 and w_2 are lists.

Example 3.2. Consider now equality on sets in OCL: $e_1 = e_2$ where e_1 and e_2 have type set. The corresponding BOTL expression will have semantics $[\![\bar{=}]\!] : \textsc{Val}^{\tau\ \mathsf{list}} \times \textsc{Val}^{\tau\ \mathsf{list}} \longrightarrow \textsc{Val}^{\mathsf{bool}}$. The operator $\bar{=}$ is defined as follow:

$$\bar{=}(w_1, w_2) \triangleq EqList(sort(del\_duplicates(w_1)), sort(del\_duplicates(w_2))).$$

Apart from *del_duplicates*, the same argument applies to bags.

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 in $\text{dom}(C.meths)$ is active. In fact, during the execution of methods, there can be some intermediate configurations in which e does not hold. (see Example 3.3).

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@pre subexpressions occur a finite number of times, say $n \geq 0$, only in e_{post}. We first enumerate all the occurrences of e@pre subexpressions in e_{post}. We write $e@_{\mathsf{i}}\mathsf{pre}$ for $1 \leq i \leq n$. Then when we translate e_{post}, by means of the function δ that we will define in the next subsection, we substitute terms $e@_{\mathsf{i}}\mathsf{pre}$ with new fresh logical variables $u_i \in \tau_i$ 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. Note that the judgment $u_i \in \tau_i$ can be inferred by the type of e in $e@_{\mathsf{i}}\mathsf{pre}$.

3.3 Translating OCL expressions into BOTL

We will now define a syntactic mapping of OCL into BOTL. First we will give a partial function δ that translates S_{OCL}. Then by means of δ we will address the issues involved in the translation of C_{OCL}. The function δ takes three parameters: o, m, $\vec{p}$. Given a $\chi \in C_{OCL}$, the first parameter o represents a variable bound to an object of the context class C. In case of pre/postcondition, the value of parameter m is a method occurrence of the context method M and $\vec{p}$ is the list of formal parameters. In case of invariants, m has an arbitrary value whereas $\vec{p}$ is the empty list. The translation function $\delta : S_{OCL} \rightharpoonup (\textsc{LogVar} \times \textsc{LogVar} \times \textsc{VName}^*) \longrightarrow S_{exp}$ is given by

$$\delta_{o,m,\vec{p}}(\mathsf{self}) = o$$

$$\delta_{o,m,\vec{p}}(z) = \begin{cases} o.z & \text{if } o \in C\ \mathsf{ref} \text{ and } z \in \mathrm{dom}(C.attrs) \\ m.z & \text{if } z \in \vec{p} \\ z & \text{otherwise} \end{cases}$$

$$\delta_{o,m,\vec{p}}(\mathsf{result}) = m.\mathsf{return}$$

$$\delta_{o,m,\vec{p}}(e@_{\mathsf{i}}\mathsf{pre}) = u_i$$

$$\delta_{o,m,\vec{p}}(e.a) = \begin{cases} \mathit{flat}(\delta_{o,m,\vec{p}}(e).a) & \text{if } e \in C\ \mathsf{ref\ list} \text{ and } C.attrs(a) = \tau\ \mathsf{list} \\ \delta_{o,m,\vec{p}}(e).a & \text{otherwise} \end{cases}$$

$$\delta_{o,m,\vec{p}}(\omega(e_1, \ldots, e_n)) = \bar{\omega}(\delta_{o,m,\vec{p}}(e_1), \ldots, \delta_{o,m,\vec{p}}(e_n))$$

$$\delta_{o,m,\vec{p}}(e.\omega(e_1, \ldots, e_n)) = \bar{\omega}(\delta_{o,m,\vec{p}}(e), \delta_{o,m,\vec{p}}(e_1), \ldots, \delta_{o,m,\vec{p}}(e_n))$$

$$\delta_{o,m,\vec{p}}(e{\rightarrow}\omega(e_1, \ldots, e_n)) = \bar{\omega}(\delta_{o,m,\vec{p}}(e), \delta_{o,m,\vec{p}}(e_1), \ldots, \delta_{o,m,\vec{p}}(e_n))$$

$$\delta_{o,m,\vec{p}}(e_1{\rightarrow}\mathsf{iterate}(z_1; z_2 = e_2 \mid e_3)) = \\ \mathsf{with}\ z_1 \in \delta_{o,m,\vec{p}}(e_1)\ \mathsf{from}\ z_2 := \delta_{o,m,\vec{p}}(e_2)\ \mathsf{do}\ z_2 := \delta_{o,m,\vec{p}}(e_3).$$

The translation of S_{OCL} is straightforward for almost every operator.

- A variable z is prefixed by the context object if it is one of its attributes; it is prefixed by m if it is among m's formal parameters.

- As discussed in the previous section, in translating $e@\mathsf{pre}$, we assume an enumeration of their occurrences, say $e@_{\mathsf{i}}\mathsf{pre}$ for $1 \leq i \leq n$. Each numbered expression is then replaced by a fresh variable u_i.
- In case of attributes or navigation $e.a$ we apply the definition recursively on the prefix. If both e and a are lists then the resulting BOTL expression has to be flattened since the result would produce a nested list that is not admitted by OCL. This is done explicitly with the operation *flat*.
- The expressions $e{\rightarrow}\omega(e_1, \ldots, e_n)$ and $e.\omega(e_1, \ldots, e_n)$ are translated using the corresponding BOTL $(n+1)$-ary operation $\bar{\omega}$.

3.4 Translating OCL constraints into BOTL

In this section we will complete the translation of OCL into BOTL by defining a map $\Delta : C_{OCL} \longrightarrow T_{exp}$.

Invariants In case of an invariant, the translation has the typical prefix AG. The invariant must hold for all active objects of the class C when none of their methods is active. Let $y \in \textsc{LogVar}$ and $\mathrm{dom}(C.meths) = \{M_1, \ldots, M_k\}$. We define:

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

The reader is invited to check that the BOTL equivalent of OCL invariant (1) is indeed the expression (3) when the collection *guests* is a bag.

Pre/postconditions As discussed above, we augment the precondition with some extra information that is used to evaluate the postcondition.

Consider the OCL expression $\mathsf{context}\ C :: M(\vec{p})\ \mathsf{pre}\ e_{\mathsf{pre}}\ \mathsf{post}\ e_{\mathsf{post}}$. The *extended* translated precondition w.r.t. the object o and the method occurrence m, $e_{\mathsf{pre}}^{o,m,\vec{p}}$ is given by

$$e_{\mathsf{pre}}^{o,m,\vec{p}} \triangleq \delta_{o,m,\vec{p}}(e_{\mathsf{pre}}) \wedge \bigwedge_{e@_{\mathsf{i}}\mathsf{pre} \in e_{\mathsf{post}}} (u_i = \delta_{o,m,\vec{p}}(e))$$

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

Here the symbol $\in$ means "occurs syntactically in". Thus given a precondition e_{pre} we can build an extended precondition $e_{\mathsf{pre}}^{o,m,\vec{p}}$ using a new variable u_i for each subexpression $e@_{\mathsf{i}}\mathsf{pre}$ involved in the postcondition, which "freezes" the value of e while evaluating the precondition and can be used in stead in the postcondition. Now we are ready to map OCL pre/postconditions into BOTL.

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

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

The expressions $e_{\mathsf{pre}}^{z,m,\vec{p}}$ 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 3.3. 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:

$$\begin{aligned} \delta_{z,m,g}(\mathsf{not}\ \ \mathit{guests}{\rightarrow}\mathit{includes}(g)) &= \neg\delta_{z,m,g}(\mathit{guests}{\rightarrow}\mathit{includes}(g)) \\ &= \neg\overline{\mathit{includes}}(z.\mathit{guests}, m.g) \end{aligned}$$

where $\overline{\mathit{includes}}$ 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:

$$\begin{aligned} e_{\mathsf{pre}}^{z,m,g} &\equiv \neg\overline{\mathit{includes}}(z.\mathit{guests}, m.g) \wedge u_1 = \delta_{z,m,g}(\mathit{guests}) \\ &\equiv \neg\overline{\mathit{includes}}(z.\mathit{guests}, m.g) \wedge u_1 = z.\mathit{guests} \end{aligned}$$

After some calculations, the translation of the postcondition yields:

$$\begin{aligned} \delta_{z,m,g}(e_{\mathsf{post}}) &= \delta_{z,m,g}(\mathit{guests}{\rightarrow}\mathit{size}) = \delta_{z,m,g}(\mathit{guests}@\mathsf{pre}{\rightarrow}\mathit{size}) + 1 \\ &\quad \wedge\ \delta_{z,m,g}(\mathit{guests}{\rightarrow}\mathit{includes}(g)) \\ &= (\overline{\mathit{size}}(z.\mathit{guests}) = \overline{\mathit{size}}(u_1) + 1 \wedge \overline{\mathit{includes}}(z.\mathit{guests}, m.g). \end{aligned}$$

The translation of (2) now yields:

$$\begin{aligned} &\forall u_1 : \forall z \in \mathsf{act}(\text{Hotel}\ \mathsf{ref}) : \forall m \in z.\mathit{checkIn}\ \mathsf{ref} : \mathsf{AG}[(e_{\mathsf{pre}}^{z,m,g} \wedge \neg\mathsf{act}(m)) \\ &\qquad \Rightarrow \mathsf{AX}[\mathsf{act}(m) \Rightarrow \mathsf{A}[\mathsf{act}(m)\mathsf{U}(\mathsf{term}(m) \wedge \delta_{z,m,g}(e_{\mathsf{post}}))]. \end{aligned}$$

Figure 4 describes the configurations of the transition system during the execution of the method *checkIn* and how the validity of pre/postcondition changes. The second and the third column describe how the components σ and γ evolve w.r.t the configuration (first column). In configuration 1 object g_1 does not belong to the guests of h. The set of method calls is empty. In this state $\neg\mathsf{act}(\mathit{checkIn})$ 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*.

4 Concluding remarks

The temporal logic BOTL developed in this paper, facilitates the specification of static properties (similar to OCL) and dynamic properties (using CTL) of object-based systems. The syntax and semantics of the logic were formally defined, and a translation of OCL into BOTL has been presented, thus providing a formal semantics to a large subset of OCL. In the future we plan to extend our approach towards subtyping and inheritance, and to work towards an effective model checking approach for BOTL. The latter issue requires a treatment of the potentially infinite number of active objects and events (method invocations).

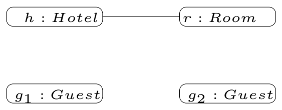
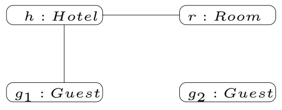
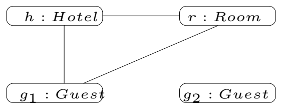
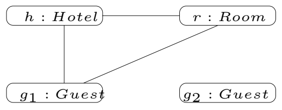
Conf	σ	γ	Valid formulae
1	 h : $Hotel$ r : $Room$ g_1 : $Guest$ g_2 : $Guest$	$\oslash$	$\neg\mathsf{act}(checkIn)$ e_{pre} $\neg e_{\mathsf{post}}$
2	 h : $Hotel$ r : $Room$ g_1 : $Guest$ g_2 : $Guest$	$(h, checkIn, 1) \mapsto (g \mapsto g_1, \perp)$	$\mathsf{act}(checkIn)$ $\neg e_{\mathsf{pre}}$ e_{post}
3	 h : $Hotel$ r : $Room$ g_1 : $Guest$ g_2 : $Guest$	$(h, checkIn, 1) \mapsto (g \mapsto g_1, ())$	$\mathsf{act}(checkIn)$ $\neg e_{\mathsf{pre}}$ e_{post}
4	 h : $Hotel$ r : $Room$ g_1 : $Guest$ g_2 : $Guest$	$\oslash$	$\neg\mathsf{act}(checkIn)$ $\neg e_{\mathsf{pre}}$ e_{post}

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

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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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).

[14] 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.

[15] 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.

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

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

[18] 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.

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

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

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

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