Proving modal and temporal properties of rewriting logic programs

Isabel Pita and Narciso Martí-Oliet
Depto. Sistemas Informáticos, Univ. Complutense Madrid, Spain

Abstract

Verification Logic for Rewriting Logic (VLRL) is a modal action logic appropriate for proving modal and temporal properties of concurrent systems specified in rewriting logic. This paper improves the initial definition of VLRL by making available two new action modalities that simplify the treatment of the contextual identity transitions. The main novelty of the VLRL logic is a topological modality associated with state constructors that allows us to reason about the structure of states, stating that the current state can be decomposed into regions satisfying certain properties. Then, on top of the modal logic, we define a temporal logic for reasoning about properties of the computations generated from rewrite theories. This is illustrated by means of several examples.

Keywords: Rewriting logic specifications, Maude, Modal and temporal logics, Verification logic for rewriting logic, Object-oriented systems.

1 Introduction

Rewriting logic is a logic for reasoning about the correctness of concurrent systems having states, and evolving by means of transitions [7]. It is a logic *of* change which can be used directly as a wide spectrum language supporting specification, rapid prototyping, and programming of concurrent systems [6]. Modal and temporal logics are, on the other hand, logics to talk *about* change in a more indirect and global manner [4, 9]. In our view these latter logics support a nonexecutable—as far as the system described is concerned—and more abstract level of specification above that of rewriting logic. Our approach envisions two different roles played by logics for concurrent systems: an *executable* role, played in our case by rewriting logic and making the programming itself declarative, and a *specification* role, played by an adequate logic for specification of concurrent systems, which is used to formulate properties about the system.

Once we have the programs writen in rewriting logic, the Verification Logic for Rewriting Logic (VLRL) [3] is used both to specify modal properties about the system, and to establish the relationship between properties given in a temporal logic and programs, so that the programs can be verified against the specification properties. VLRL takes programs as models and allows observation properties to be

derived directly or through inference rules that relate the program and the temporal specification logic (or logics). In this setting, given a program P and a specification S, verifying that P satisfies S consists in determining a set V of sentences in VLRL such that P is a model of V and V entails S.

In order to express the properties of the system, we make available attributes for making observations of the state of a system and action symbols to account for its elementary state changes. From those action symbols, we build action terms α associated to the transitions in the system. Then, the modalities $[\alpha]$ and $\langle\alpha\rangle$ are the usual modalities representing possibility and necessity, that capture the state transitions at the top level of the system. We improve the initial definition of VLRL by making available two new action modalities that simplify the treatment of the contextual identity transitions. Another novelty of the VLRL logic is a topological modality associated with state constructors that allows us to reason about the structure of states, stating that the current state can be decomposed into regions satisfying certain properties.

In Section 2 we present some basic notions of rewriting logic and the Maude language; Section 3 introduces VLRL as it is presented in [3], and in Section 4 we propose the new action modalities. Next, Section 5 introduces the temporal logic we shall use and the interface inference rules of the temporal logic with VLRL. Finally, in Section 6 we illustrate the use of the logic, in particular the action and topological modalities, and prove some temporal properties about an object-oriented application.

2 Rewriting Logic and the Maude Language

We outline here some basic notions of rewriting logic and its implementation in the specification and programming language Maude needed for the application case. For more information on the subject see [1, 7, 8].

A *rewrite theory* $\mathcal{R}$ is defined as a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where (Σ, E) is an equational signature, L is a set of labels, and R is a set of *rewrite rules* of the form $l : [t]_E \longrightarrow [t']_E$, where $l \in L$, t and t' are Σ-terms possibly involving some variables, and $[t]_E$ denotes the equivalence class of term t modulo the equations E. In order to simplify the presentation, in the following we will not make explicit the equivalence class of terms. Intuitively, the signature (Σ, E) of a rewrite theory describes a particular structure for the states of a system, and the rewrite rules describe which elementary local transitions are possible in the distributed state by concurrent local transformations.

Rewriting logic constitutes the foundation of the specification and programming language Maude. In this paper we only make use of unconditional rewrite rules, but both equations and rewrite rules can be conditional [7]. Systems in Maude are built out of basic elements called modules. *Functional modules* are used for the definition of algebraic data types and *object-oriented modules* for the definition of object-oriented classes.

An object is represented as a term $< O : C \mid a_1 : v_1, \ldots, a_n : v_n >$, where O is the object's name belonging to a set *Oid* of object identifiers, C is the class identifier, a_i

are the names of the object attributes, and v_i are their corresponding values, which typically are required to be in a sort appropriate for their corresponding attribute.

Rewrite rules represent the implementation of the method associated to a message received by an object. An unconditional rewrite rule has the form

$$\begin{array}{lll} \textbf{rl } [l]: & M_1 \ldots M_n < O_1 : C_1 \mid atts_1 > \ldots < O_m : C_m \mid atts_m > \\ \Longrightarrow & < O_{i_1} : C'_{i_1} \mid atts'_{i_1} > \ldots < O_{i_k} : C'_{i_k} \mid atts'_{i_k} > \\ & < Q_1 : D_1 \mid atts''_1 > \ldots < Q_p : D_p \mid atts''_p > \\ & M'_1 \ldots M'_q \end{array}$$

where $n, m, k, p, q \geq 0$, the M_s are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and l is a label. The result of applying such a rewrite rule is that: the messages $M_1, \ldots, M_n$ disappear; the state and possibly the class of the objects $O_{i_1}, \ldots, O_{i_k}$ may change; all the other objects O_j vanish; new objects $Q_1, \ldots, Q_p$ are created; and new messages $M'_1, \ldots, M'_q$ are sent.

By convention, the only object attributes $atts_1, \ldots, atts_m$ made explicit in a rule are those relevant for that rule. In particular, the attributes mentioned only on the lefthand side of the rule are preserved unchanged, the original values of attributes mentioned only on the righthand side of the rule do not matter, and all attributes not explicitly mentioned are left unchanged.

3 Verification Logic

A *verification signature* is defined by $\langle \Sigma^+, E^+, State, At, L \rangle$ where $\langle \Sigma^+, E^+ \rangle$ defines a conservative extension of $\langle \Sigma, E \rangle$, *State* is a designated sort of Σ, At is a family of observation attributes, and L is a collection of labels indexed over strings of sorts in Σ. The idea is to make available attributes for making observations of the state of a system and, since we use a conservative extension of the original signature, the set of states $T_{\Sigma,E,State}$ is the *same* set as $T_{\Sigma^+,E^+,State}$. Hence, we take models to be Kripke frames associated to the rewrite theories over an extended signature.

Action symbols account for the elementary state changes. Since rewrite rules may require the context in which they are being applied to be considered as actions, the following language of action terms defines a proper subset of all concurrent one-step rewrites [7]. *Pre-action terms* α correspond to the quotient of the set of proof terms obtained through the following rules of deduction [1]:

- *Identities*: for each $[t] \in T_{\Sigma,E}(X)$, $\quad \dfrac{}{[t] : [t] \to [t]}$,
- Σ-*structure*: for each $f \in \Sigma$, $\quad \dfrac{\alpha_i : [t_i] \to [t'_i]}{f(\overline{\alpha}) : [f(\overline{t})] \to [f(\overline{t'})]}$,
- *Replacement*: for each labelled rewrite rule $r(\overline{x}) : [t(\overline{x})] \to [t'(\overline{x})]$ in R,

$$\frac{}{r(\overline{w}) : [t(\overline{w}/\overline{x})] \to [t'(\overline{w}/\overline{x})]} \ ,$$

[1] Here and in the rest of the paper, an overbar is used to abbreviate sequences of expressions; for example, $f(\overline{t})$ denotes $f(t_1, \ldots, t_n)$.

modulo the following equations:

- Identity transitions: $f([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)]$,
- Axioms in $E : t(\overline{\alpha}) = t'(\overline{\alpha})$, for each equation $t = t'$ in E.

Action terms are the pre-action terms that rewrite terms of sort *State*.

The *term language* is the term algebra $T_{\Sigma_c^+}(X)$, where Σ^+ is extended to Σ_c^+ with attributes as constants and X has an infinite set of variables for each sort in Σ_c^+. The *modal language* is given by

$$\varphi ::= true \mid t_1 = t_2 \mid \neg\varphi \mid \varphi_1 \supset \varphi_2 \mid [\alpha]\varphi \mid f_{\overline{d}}[\varphi_1, \ldots, \varphi_n]$$

where $t_1, t_2 \in T_{\Sigma_c^+}(X)$ are two terms of the same sort, α is an action term, $f : s_1 \ldots s_m \to State \in \Sigma$, $\overline{d}$ is a sequence of data terms corresponding to the arguments of f that are not of sort *State*, and the φ_i are in one-to-one correspondence with the arguments of f that are of sort *State*. We apply this notational convention also to action and state terms.

Given an action term α, $[\alpha]$ is the usual modality that captures the state transitions performed by the action. $f_{\overline{d}}[\_]$ is a topological modality that allows us to reason about the *structure* of states, that is, about the fact that the current state of the system can be decomposed into components that satisfy certain properties.

We will make use of the usual derived propositional connectives for conjuntion ($\wedge$), disjunction ($\vee$), and logical equivalence ($\equiv$).

The term algebra T_{Σ^+,E^+} provides the structure over which formula satisfaction is defined. Satisfaction of formulae at a given state $[t] \in T_{\Sigma^+,E^+}$, for a given attribute interpretation I[2], and substitution σ of ground Σ^+-terms for variables, is defined as follows:

- $[\![x]\!]^{I,\sigma}[t] = \sigma(x)$,
- $[\![a]\!]^{I,\sigma}[t] = I(a)([t])$,
- $[\![f(t_1, \ldots, t_m)]\!]^{I,\sigma}[t] = f([\![t_1]\!]^{I,\sigma}[t], \ldots, [\![t_m]\!]^{I,\sigma}[t])$, for each $f : s_1 \ldots s_m \to s$ in Σ^+.
- $[t], I, \sigma \models true$,
- $[t], I, \sigma \models t_1 = t_2$ iff $[\![t_1]\!]^{I,\sigma}[t] = [\![t_2]\!]^{I,\sigma}[t]$,
- $[t], I, \sigma \models \neg\varphi$ iff it is not the case that $[t], I, \sigma \models \varphi$,
- $[t], I, \sigma \models \varphi_1 \supset \varphi_2$ iff $[t], I, \sigma \models \varphi_1$ implies $[t], I, \sigma \models \varphi_2$,
- $[t], I, \sigma \models [\alpha]\varphi$ iff $[\![\alpha]\!]^{I,\sigma} : [t] \to [t']$ implies $[t'], I, \sigma \models \varphi$,
- $[t], I, \sigma \models f_{\overline{d}}[\varphi_1, \ldots, \varphi_n]$ iff for each $f_{\overline{w}}(t_1, \ldots, t_n) \in [t]$, where $\overline{w} = [\![\overline{d}]\!]^{I,\sigma}[t]$, there is some $i \in \{1, \ldots, n\}$ such that $[t_i], I, \sigma \models \varphi_i$.

[2]The interpretation I provides an interpretation for the observation attributes: for each sort s, I maps observations of sort s to functions of type $T_{\Sigma,E,State} \to T_{\Sigma^+,E^+,s}$.

where $[\![\alpha]\!]^{I,\sigma}$ is the state transition given by the ground action term obtained by applying the substitution σ to the action term α.

We have two dual operators:

- $[t], I, \sigma \models \langle\alpha\rangle\varphi$ iff $[\![\alpha]\!]^{I,\sigma} : [t] \to [t']$ and $[t'], I, \sigma \models \varphi$.
- $[t], I, \sigma \models f_{\overline{d}}\langle\varphi_1, \ldots, \varphi_n\rangle$ iff there is a term $f_{\overline{w}}(t_1, \ldots, t_n) \in [t]$ with $\overline{w} = [\![\overline{d}]\!]^{I,\sigma}[t]$ such that $[t_i], I, \sigma \models \varphi_i$ for each $i = 1, \ldots, n$.

Notice that the modality $\langle\alpha\rangle$ requires the action to denote an existing transition from the current state. The modality $[\alpha]$ does not impose the requirement that such a transition actually exists. Hence, $[\alpha]\varphi$ holds trivially if, for the given substitution, α cannot rewrite the current state.

The dual of the topological operator allows us to say that the current state is decomposable according to the structure f into components that satisfy the formulae φ_i. The $f_{\overline{d}}[\varphi_1, \ldots, \varphi_n]$ modality introduced above uses instead the subjunctive condition *if the state is decomposable in the specified way, then ....*

Finally, we say that a formula is valid in a model iff it is satisfied at every state for any attribute interpretation and any ground substitution for the variables.

A partial axiomatization of the logic includes the following rules of deduction [3]

$$\vdash \langle\alpha\rangle\varphi \equiv [\alpha]\varphi \wedge \langle\alpha\rangle true \tag{1}$$

$$\{\varphi_i \supset [\alpha_i]\psi_i \mid i \in \{1, \ldots, n\}\} \vdash f_{\overline{d}}\langle\varphi_1, \ldots, \varphi_n\rangle \supset [f_{\overline{d}}(\alpha_1, \ldots, \alpha_n)] f_{\overline{d}}\langle\psi_1, \ldots, \psi_n\rangle \tag{2}$$

$$\vdash t_1 = t_2 \supset [\alpha](t_1 = t_2) \tag{3}$$

where t_1 and t_2 do not involve state attributes. Soundness of the rules can easily be proved. For proving completeness we are currently trying to adapt the techniques used to prove completeness of modal and dynamic logics.

4 Transitions *in context*

A natural extension of the modal language is to consider new modalities that simplify the treatment of contextual identity transitions. Given a state $[t]$, an attribute interpretation I, and a ground substitution σ:

- $[t], I, \sigma \models [[\alpha]]\varphi$ iff $[t], I, \sigma \models [\alpha]\varphi$ and $[t], I, \sigma \models [\beta(\alpha)]\varphi$, for all action terms $\beta(\alpha)$ constructed only by the identities and Σ-structure rules on top of α.
- $[t], I, \sigma \models \langle\langle\alpha\rangle\rangle\varphi$ iff either $[t], I, \sigma \models \langle\alpha\rangle\varphi$ or there is an action term $\beta(\alpha)$ constructed only by the identities and Σ-structure rules on top of α such that $[t], I, \sigma \models \langle\beta(\alpha)\rangle\varphi$.

The new modalities allow us to formulate properties about parts of the system in a wider context without complicating the notation by the need to make explicit the contextual identity transitions. The action modality $[[\alpha]]$ captures properties that hold after rewriting *any subterm* of the term that represents the state. The idea of

the action modality $\langle\langle\alpha\rangle\rangle$ is that we can perform an *action term in context* if either we can perform the action term in the whole state or we can perform the action term in a substate while performing identity transitions in the rest of substates in which we can divide the state. In this case it is required one of the transitions to take place.

We consider the following properties concerning the new modalities. First, we have a version of (1) for the new modalities:

$$\vdash [[\alpha]]\varphi \wedge \langle\langle\alpha\rangle\rangle true \supset \langle\langle\alpha\rangle\rangle\varphi \ .$$

The main difference with (1) is that, if we rewrite inside a state, the fact that a transition takes place in a substate yielding a property does not guarantee that the transition cannot take place in a different substate yielding a different property. Because of this, the converse is not always true. Consider, for example, an alphabet with two constants: `a` and `b`; two operations to construct strings: the constant `nil` and a concatenation operation, which is declared associative and with identity `nil`; and rules for changing one constant into the other

```
sorts Alpha String .
subsort Alpha < String .
ops a b : -> Alpha .
op nil : -> String .
op __ : Alpha String -> String [assoc id:nil] .
rl [ch1] : a => b .
rl [ch2] : b => a .
```

We consider an observation attribute *first* that returns the first character of a string. Then, for example, in the state `abaab`, the property $\langle\langle\texttt{ch1}\rangle\rangle(\mathit{first} = b)$ is fulfilled, since we can apply the rule `ch1` *in context* to the first character of the string. However, the property $[[\texttt{ch1}]](\mathit{first} = b)$ is not fulfilled, since we can apply the rewrite rule to the second or third a of the string.

The following are justified directly from the definition

$$\begin{aligned} &\vdash \langle\alpha\rangle\varphi \supset \langle\langle\alpha\rangle\rangle\varphi \\ &\vdash [[\alpha]]\varphi \supset [\alpha]\varphi \ . \end{aligned} \tag{4}$$

The following property expresses when an action can be done *in context*

$$\varphi_i \supset \langle\alpha\rangle\psi \vdash f_{\overline{w}}\langle\varphi_1,\ldots,\varphi_n\rangle \supset \langle\langle\alpha\rangle\rangle f_{\overline{w}}\langle\varphi_1,\ldots,\varphi_{i-1},\psi,\varphi_{i+1},\ldots,\varphi_n\rangle$$
for any $i \in \{1,\ldots,n\}$.

5 Temporal Logic

Different logics can be defined on top of VLRL depending on the nature of the properties about which one wants to reason. Each logic should be related with VLRL by means of interface inference rules that support the verification of programs against specifications. We will use a branching time temporal logic for writing specifications. Three logical operators are provided with the following meaning:

- AXφ: the property φ holds at all possible successor states,
- A(φWψ): in every computation φ will hold until ψ holds (weak until),
- EXφ: the property φ holds at some successor state.

Notice that AGφ (the property φ is valid at all possible future states) is equivalent to A(φW*false*).

The interface inference rules are:

$$\varphi \supset \psi,\ \{\varphi \supset [\alpha]\psi \mid \alpha \text{ action term}\} \vdash \varphi \supset \mathrm{AX}\psi \tag{5}$$

$$\{\varphi \wedge \neg\psi \supset [\alpha](\varphi \vee \psi) \mid \alpha \text{ action term}\} \vdash \varphi \supset \mathrm{A}(\varphi \mathrm{W} \psi) \tag{6}$$

$$\varphi \supset \langle\alpha\rangle\psi \vdash \varphi \supset \mathrm{EX}\psi \quad \text{where } \alpha \text{ is an action term.} \tag{7}$$

Notice that, since α can be of the form $\beta(\alpha')$ where β only adds identities and Σ-structure to an action α', by the definition of the modality $\langle\langle\alpha'\rangle\rangle$, (7) can also be expressed as $\varphi \supset \langle\langle\alpha\rangle\rangle\psi \vdash \varphi \supset \mathrm{EX}\psi$.

6 Examples: A blocks world

In this example, we illustrate the use of the topological and action modalities and we introduce the use of variables in the definition of properties. The example is based on an object-oriented approach to the world of blocks [1, Section 9.5]. A block is represented as an object with two attributes, `under`, saying whether it is under another block or it is clear, and `on`, saying whether the block is on top of another block or it is on the table. A robot arm is represented as another object with one attribute `hold`, saying whether the robot arm is empty or it holds a block. Actions are represented as messages.

First we present the system in Maude, then we prove some properties about the number of blocks in the system, and then, we introduce observation attributes with variables to prove properties about the blocks position in the system.

```
(omod OO-BLOCKSWORLDN is
  protecting QID .
  sorts BlockId RobotId Up Down Hold .
  subsorts Qid < BlockId RobotId < Oid .
  subsorts BlockId < Up Down Hold .
  op clear : -> Up .      *** block is clear
  op catchup : -> Up .    *** block is caught by the robot arm
  op table : -> Down .    *** block is on the table
  op catchd : -> Down .   *** block is caught by the robot arm
  op empty : -> Hold .    *** robot arm is empty
  class Block | under : Up, on : Down .
  class Robot | hold : Hold .
  msgs pickup putdown : RobotId BlockId -> Msg .
  msgs unstack stack : RobotId BlockId BlockId -> Msg .
  vars X Y : BlockId .
  var R : RobotId .
```

```
  rl [pickup] : pickup(R,X) < R : Robot | hold : empty >
                < X : Block | under : clear, on : table >
             => < R : Robot | hold : X >
                < X : Block | under : catchup, on : catchd > .
  rl [putdown] : putdown(R,X) < R : Robot | hold : X >
                 < X : Block | under : catchup, on : catchd >
              => < R : Robot | hold : empty >
                 < X : Block | under : clear, on : table > .
  rl [unstack] : unstack(R,X,Y) < R : Robot | hold : empty >
                 < X : Block | under : clear, on : Y >
                 < Y : Block | under : X >
              => < R : Robot | hold : X >
                 < X : Block | under : catchup, on : catchd >
                 < Y : Block | under : clear > .
  rl [stack] : stack(R,X,Y) < R : Robot | hold : X >
                 < X : Block | under : catchup, on : catchd >
                 < Y : Block | under : clear >
              => < R : Robot | hold : empty >
                 < X : Block | under : clear, on : Y >
                 < Y : Block | under : X > .
endom)
```

6.1 Observing the number of blocks

To formulate properties about the number of blocks in the system we define the following observation attributes: [3]

- *#blTable*, which represents the number of blocks on the table,
- *#blOnBl*, which represents the number of blocks that are on other blocks,
- *#blHold*, which represents the number of blocks that are held by robot arms,
- *#blocks*, which represents the total number of blocks in the system,
- *#empty*, which represents the number of empty robot arms,
- *#robots*, which represents the total number of robot arms.

In the following we will consider only consistent configurations, in the sense that they describe possible blocks world situations: blocks are not duplicated, if a block is on top of another block then the latter should be under the former, and so on.

6.2 Topological and action modalities

In this section we present some examples on the use of the topological modality and the action modalities in the specification of properties. In the configuration

```
< 'a : Block | under : 'c, on : table >
< 'c : Block | under : clear, on : 'a >
```

[3] The observed program is given in Appendix A.

```
< 'b : Block | under : clear, on : table >
< 'r : Robot | hold : empty > < 's : Robot | hold : empty >
```

the following formulae are satisfied:

- The state can be separated into a substate with two blocks, one over the other, and a robot arm, and a substate with one block on the table and the other robot arm:

$$\_\_\langle \#blocks = 2 \wedge \#blOnBl = 1 \wedge \#robots = 1, \\ \#blocks = 1 \wedge \#blTable = 1 \wedge \#robots = 1 \rangle$$

- The state can also be separated into the blocks and the robot arms:

$$\_\_\langle \#blocks = 3 \wedge \#robots = 0, \#blocks = 0 \wedge \#robots = 2 \rangle$$

With regard to the action modalities, in the state

```
< 'a : Block | under : catchup, on : catchd >
< 'c : Block | under : 'b, on : table >
< 'b : Block | under : clear, on : 'c >
< 'r : Robot | hold : empty > < 's : Robot | hold : 'a >
putdown('s,'a)  unstack('r,'b,'c)
```

the following formulae are satisfied:

- Rule `putdown` can be applied to the robot arm `s` and block `a`:

$$\langle\langle \texttt{putdown}_{\texttt{s,a}} \rangle\rangle \mathit{true}$$

  Notice that we use the modality $\langle\langle \_ \rangle\rangle$ instead of the *single* modality $\langle \_ \rangle$ since we want to apply the transition *in context* to the subterm

```
< 'a : Block | under : catchup, on : catchd >
< 's : Robot | hold : 'a > putdown('s,'a)
```

- Rules `unstack` and `putdown` can be applied concurrently:

$$\langle \_\_(\texttt{unstack}_{\texttt{r,b,c}}, \texttt{putdown}_{\texttt{s,a}}) \rangle \mathit{true}$$

  Here we can use the *single* modality $\langle \_ \rangle$ since we apply the action term to the whole configuration.

- Rules `unstack` and `putdown` can be applied concurrently, and after the transition is performed, a substate will have one block held by a robot arm and the other substate will have one block on the table:

$$\langle \_\_(\texttt{unstack}_{\texttt{r,b,c}}, \texttt{putdown}_{\texttt{s,a}}) \rangle \_\_\langle \#blHold = 1, \#blTable = 1 \rangle$$

- If rules `unstack` and `putdown` are applied concurrently, then after the transition takes places a substate will have two blocks, one of them held, and a robot arm, and the other substate will have an empty robot arm and a block on the table:

$$[\_(\mathtt{unstack}_{\mathtt{r,b,c}}, \mathtt{putdown}_{\mathtt{s,a}})]$$
$$\_\langle \#blHold = 1 \wedge \#blocks = 2 \wedge \#robots = 1, \#empty = 1 \wedge \#blTable = 1 \rangle$$

- If rules `pickup` and `putdown` are applied concurrently *in context*, then after the transition takes places a substate will have more than one block on the table and the other substate will have more than one block held:

$$[[\_\_(\mathtt{pickup}_{\mathtt{r,b}}, \mathtt{putdown}_{\mathtt{s,a}})]]\_\_\langle \#blHold \geq 1, \#blTable \geq 1 \rangle$$

- In all possible substates, if we apply rule `putdown` to the robot arm `s` and to the block `a`, then the number of blocks on the table in that substate will be greater than 1:

$$\_\_[[\mathtt{putdown}_{\mathtt{s,a}}]\#blTable \geq 1, true]$$

6.3 Basic properties about the number of blocks

In this section, we present some basic properties fullfilled by the `OO-BLOCKSWORLDN` system that can be directly proved from the satisfaction relation. We are investigating the possibility of automatically deriving these properties from the program.

Properties about the transitions of the system. For each rewrite rule we express the changes of each observation attribute when the transition associated with it takes place. For example, if we take the rewriting rule `pickup` and the corresponding action term $\mathtt{pickup}_{\mathtt{u,a}}$, where `u` and `a` represent a substitution of the rule variables, we can formulate the following properties about the number of blocks in the world:

$$\#blHold = N \supset [[\mathtt{pickup}_{\mathtt{u,a}}]](\#blHold = N + 1)$$
$$\#blTable = N \supset [[\mathtt{pickup}_{\mathtt{u,a}}]](\#blTable = N - 1)$$
$$\#blOnBl = N \supset [[\mathtt{pickup}_{\mathtt{u,a}}]](\#blOnBl = N)$$
$$\#empty = N \supset [[\mathtt{pickup}_{\mathtt{u,a}}]](\#empty = N - 1)$$
$$\#blocks = N \supset [[\mathtt{pickup}_{\mathtt{u,a}}]](\#blocks = N)$$
$$\#robots = N \supset [[\mathtt{pickup}_{\mathtt{u,a}}]](\#robots = N)$$

Properties that characterize the enabling conditions for the rewrite rules. Looking at the lefthand side of each rewrite rule we can define the following properties:

$$\langle\langle \mathtt{pickup}_{\mathtt{u,a}} \rangle\rangle true \supset \#blTable \geq 1 \wedge \#empty \geq 1$$
$$\langle\langle \mathtt{putdown}_{\mathtt{u,a}} \rangle\rangle true \supset \#blHold \geq 1$$
$$\langle\langle \mathtt{unstack}_{\mathtt{u,a,b}} \rangle\rangle true \supset \#blOnBl \geq 1 \wedge \#empty \geq 1$$
$$\langle\langle \mathtt{stack}_{\mathtt{u,a,b}} \rangle\rangle true \supset \#blHold \geq 1$$

Structural properties of attributes concerning the number of blocks. These properties express when a state can be separated in substates, and the properties fulfilled by

each substate. They depend on the system configuration, and the fact that the system admits empty states will influence them. For example we have:

$$\#blocks = N \equiv \exists N_1, N_2 . \_\_\langle \#blocks = N_1, \#blocks = N_2 \rangle \wedge (N = N_1 + N_2) \quad (8)$$

Notice that if the system did not admit empty states, then this property would no longer be an equivalence, since then there would be no way to decompose a state with only one block.

6.4 Temporal properties about the number of blocks

The number of blocks in the system is invariant:

$$\#blocks = N \supset \mathrm{AG}(\#blocks = N) \ .$$

By definition of the temporal connective AG, $\mathrm{AG}\varphi \equiv \mathrm{A}(\varphi \mathrm{W} false)$ and applying the second interface inference rule of temporal logic and VLRL (6), it is enough to prove

$$\{\#blocks = N \supset [\alpha](\#blocks = N) \mid \alpha \text{ action term}\} \ .$$

Using structural induction on action terms [3] to deal with the infinite set of action terms, we have to prove:

- *constants* [4]: In a state composed only by a constant the property is fulfilled for the action term related to that constant,

  1. $\texttt{<\_:\_|\ >}_{x_1,x_2}\langle\rangle \supset (\#blocks = N) \supset [[\texttt{<\_:\_|\ >}_{x_1,x_2}]](\#blocks = N)$
  2. $\texttt{<\_:\_|\_\ >}_{x_1,x_2,x_3}\langle\rangle \supset (\#blocks = N) \supset [[\texttt{<\_:\_|\_\ >}(x_1,x_2,x_3)]](\#blocks = N)$
  3. $\texttt{none}\langle\rangle \supset (\#blocks = N) \supset [[\texttt{none}]](\#blocks = N)$
  4. $\texttt{pickup}_{\texttt{u,a}}\langle\rangle \supset (\#blocks = N) \supset [[\texttt{pickup}_{\texttt{u,a}}]](\#blocks = N)$
  5. $\texttt{putdown}_{\texttt{u,a}}\langle\rangle \supset (\#blocks = N) \supset [[\texttt{putdown}_{\texttt{u,a}}]](\#blocks = N)$
  6. $\texttt{unstack}_{\texttt{u,a,b}}\langle\rangle \supset (\#blocks = N) \supset [[\texttt{unstack}_{\texttt{u,a,b}}]](\#blocks = N)$
  7. $\texttt{stack}_{\texttt{u,a,b}}\langle\rangle \supset (\#blocks = N) \supset [[\texttt{stack}_{\texttt{u,a,b}}]](\#blocks = N)$

- *rules*: The property is fulfilled for the action term related to each rewrite rule,

  8. $(\#blocks = N) \supset [\texttt{pickup}_{\texttt{u,x}}](\#blocks = N)$
  9. $(\#blocks = N) \supset [\texttt{putdown}_{\texttt{u,x}}](\#blocks = N)$
  10. $(\#blocks = N) \supset [\texttt{unstack}_{\texttt{u,x,y}}](\#blocks = N)$
  11. $(\#blocks = N) \supset [\texttt{stack}_{\texttt{u,x,y}}](\#blocks = N)$

- *state operations* (in this case, only $\_\_$): If the property is fulfilled for an action term in each substate then, if the state can be decomposed, the property is fulfilled for the action term that results from applying the state operation to the action terms of each substate,

[4] Some constants of sort *State* are obtained from the `CONFIGURATION` module, imported in the transformation process of an object-oriented module to a system module [8].

12. $((\#blocks = N_1) \supset [\alpha_1](\#blocks = N_1)),$
$((\#blocks = N_2) \supset [\alpha_2](\#blocks = N_2)) \vdash$
$\_\_\langle true, true\rangle \supset (\#blocks = N \supset [\_(\alpha_1, \alpha_2)](\#blocks = N))$

The proofs of the first seven properties are straightforward. Derivation of properties 8-11 is based on the properties about the transitions of the system, and on the property (4). We prove the last property as follows. Assume the hypotheses; using the inference rule (2) of VLRL, we derive

$$\_\_\langle \#blocks = N_1, \#blocks = N_2\rangle \supset [\_(\alpha_1, \alpha_2)](\_\_\langle \#blocks = N_1, \#blocks = N_2\rangle)$$

Using the structural property of attributes (8):

$$\#blocks = N \supset \exists N_1, N_2.[\_\_(\alpha_1, \alpha_2)](\_\_\langle \#blocks = N_1, \#blocks = N_2\rangle) \wedge (N = N_1 + N_2).$$

Since N, N_1, N_2 are not state variables, we can apply (3):

$$\#blocks = N \supset \exists N_1, N_2.[\_\_(\alpha_1, \alpha_2)](\_\_\langle \#blocks = N_1, \#blocks = N_2\rangle) \wedge [\_\_(\alpha_1, \alpha_2)](N = N_1 + N_2).$$

Applying $[\alpha]\varphi \wedge [\alpha]\psi \equiv [\alpha](\varphi \wedge \psi)$ and $\exists X.[\alpha]\varphi \supset [\alpha]\exists X.\varphi$, we have

$$\#blocks = N \supset [\_\_(\alpha_1, \alpha_2)]\exists N_1, N_2.(\_\_\langle \#blocks = N_1, \#blocks = N_2\rangle) \wedge (N = N_1 + N_2).$$

Finally, using again the structural property of attributes (8) we get the result:

$$\#blocks = N \supset [\_(\alpha_1, \alpha_2)](\#blocks = N).$$

6.5 Observing the blocks position

To formulate properties about the position of the blocks in the system we need to take into account each individual block. We consider the following observation attributes: [5]

- $blTable_x$, which is true iff block x is on the table.
- $blClear_x$, which is true iff there is no block on block x.
- $blOnBl_{x,y}$, which is true iff block x is on block y.
- $blHold_{r,x}$, which is true iff block x is held by the robot arm r.
- $blHold_x$, which is true iff block x is held by some robot arm.
- $empty_r$, which is true iff the robot arm r is empty.

[5] The *observed program* is given in Appendix A.

6.6 Basic properties about the blocks position

Properties about the transitions of the system:

$$
\begin{aligned}
&blTable_{\mathtt{a}} \wedge blClear_{\mathtt{a}} \wedge empty_{\mathtt{u}} \supset [[\mathtt{pickup_{u,a}}]]blHold_{\mathtt{u,a}} \\
&blOnBl_{\mathtt{a,b}} \supset [[\mathtt{pickup_{u,c}}]]blOnBl_{\mathtt{a,b}} \qquad (9)\\
&blHold_{\mathtt{u,a}} \supset [[\mathtt{putdown_{u,a}}]](empty_{\mathtt{u}} \wedge blClear_{\mathtt{a}} \wedge blTable_{\mathtt{a}}) \\
&blOnBl_{\mathtt{a,b}} \supset [[\mathtt{putdown_{u,c}}]]blOnBl_{\mathtt{a,b}} \qquad (10)\\
&empty_{\mathtt{u}} \wedge blClear_{\mathtt{a}} \wedge blOnBl_{\mathtt{a,b}} \supset \\
&\qquad [[\mathtt{unstack_{u,a,b}}]](blHold_{\mathtt{u,a}} \wedge blClear_{\mathtt{b}}) \qquad (11)\\
&blHold_{\mathtt{u,a}} \wedge blClear_{\mathtt{b}} \supset [[\mathtt{stack_{u,a,b}}]](empty_{\mathtt{u}} \wedge blClear_{\mathtt{a}} \wedge blOnBl_{\mathtt{a,b}}) \\
&blOnBl_{\mathtt{a,b}} \supset [[\mathtt{stack_{u,c,d}}]]blOnBl_{\mathtt{a,b}} \qquad (12)
\end{aligned}
$$

Properties that characterize the enabling conditions for the rewrite rules:
We define a property for each rewrite rule in the system:

$$
\begin{aligned}
&\langle\langle \mathtt{pickup_{u,a}} \rangle\rangle true \supset blTable_{\mathtt{a}} \wedge blClear_{\mathtt{a}} \wedge empty_{\mathtt{u}} \\
&\langle\langle \mathtt{putdown_{u,a}} \rangle\rangle true \supset blHold_{\mathtt{u,a}} \\
&\langle\langle \mathtt{unstack_{u,a,b}} \rangle\rangle true \supset blOnBl_{\mathtt{a,b}} \wedge blClear_{\mathtt{a}} \wedge empty_{\mathtt{u}} \qquad (13)\\
&\langle\langle \mathtt{stack_{u,a,b}} \rangle\rangle true \supset blHold_{\mathtt{u,a}} \wedge blClear_{\mathtt{b}}
\end{aligned}
$$

Properties of derived observation attributes. In the set of observation attributes that we have defined, some attributes can be expressed as a function of the values of other observation attributes.

$$
\begin{aligned}
&blClear_{\mathtt{a}} \supset \neg blOnBl_{\mathtt{b,a}} \wedge \neg blHold_{\mathtt{u,a}} \qquad (14)\\
&blOnBl_{\mathtt{a,b}} \supset \neg blTable_{\mathtt{a}} \wedge \neg blClear_{\mathtt{b}} \wedge \neg blHold_{\mathtt{u,a}} \wedge \neg blHold_{\mathtt{v,b}} \qquad (15)\\
&blHold_{\mathtt{u,a}} \supset blHold_{\mathtt{a}} \qquad (16)
\end{aligned}
$$

Structural properties of attributes concerning the blocks position. An observation attribute is true in a state, if it is true in some substate.

$$
\begin{aligned}
&blOnBl_{\mathtt{a,b}} \wedge empty_{\mathtt{u}} \equiv \_\_\langle blOnBl_{\mathtt{a,b}} \wedge empty_{\mathtt{u}}, true\rangle \qquad (17)\\
&blHold_{\mathtt{u,a}} \equiv \_\_\langle blHold_{\mathtt{u,a}}, true\rangle \qquad (18)\\
&blOnBl_{\mathtt{a,b}} \equiv \_\langle blOnBl_{\mathtt{a,b}}, true\rangle \qquad (19)\\
&blHold_{\mathtt{a}} \equiv \_\_\langle blHold_{\mathtt{a}}, true\rangle \qquad (20)
\end{aligned}
$$

6.7 Temporal properties about the blocks position

If a block is under another block then it cannot be caught by an empty robot arm:

$$blOnBl_{\mathtt{x,y}} \wedge empty_{\mathtt{r}} \supset \ \mathrm{AX}(\neg blHold_{\mathtt{r,y}}) \ .$$

Applying the first interface inference rule of temporal logic and VLRL (5), it is enough to prove

$$
\begin{aligned}
&blOnBl_{\mathtt{x,y}} \wedge empty_{\mathtt{r}} \supset \neg blHold_{\mathtt{r,y}}, \\
&\{blOnBl_{\mathtt{x,y}} \wedge empty_{\mathtt{r}} \supset [\alpha](\neg blHold_{\mathtt{r,y}}) \mid \alpha \text{ action term}\} \ .
\end{aligned}
$$

For the first part we get from (15):

$$blHold_{\mathtt{r,y}} \supset \neg blOnBl_{\mathtt{x,y}} \vee \neg empty_{\mathtt{r}}$$

For the second part, using structural induction on action terms, we have to prove:

- *constants* are derived like in the previous example.
- *rules*:
  1. $(blOnBl_{\mathtt{x,y}} \wedge empty_{\mathtt{r}}) \supset [\mathtt{pickup_{u,z}}](\neg blHold_{\mathtt{r,y}})$ Apply properties (9), (15) and (4) to obtain the result.
  2. $(blOnBl_{\mathtt{x,y}} \wedge empty_{\mathtt{r}}) \supset [\mathtt{putdown_{u,z}}](\neg blHold_{\mathtt{r,y}})$ Apply properties (10), (15) and (4) to obtain the result.
  3. $(blOnBl_{\mathtt{x,y}} \wedge empty_{\mathtt{r}}) \supset [\mathtt{unstack_{u,z1,z2}}](\neg blHold_{\mathtt{r,y}})$ We distinguish two cases: (i) If $blClear_{\mathtt{x}}$ then apply (11) and then (14) and (4) to obtain the result; (ii) If $\neg blClear_{\mathtt{x}}$ apply the *contrapositive law* of propositional logic to (13) and (4).
  4. $(blOnBl_{\mathtt{x,y}} \wedge empty_{\mathtt{r}}) \supset [\mathtt{stack_{u,z1,z2}}](\neg blHold_{\mathtt{r,y}})$ The property is derived like the first and the second ones.
- *state operations* (in this case, only $\_\_$):
  5. $((blOnBl_{\mathtt{x1,y1}} \wedge empty_{\mathtt{r1}}) \supset [\alpha_1](\neg blHold_{\mathtt{r1,y1}}))$,
     $((blOnBl_{\mathtt{x2,y2}} \wedge empty_{\mathtt{r2}}) \supset [\alpha_2](\neg blHold_{\mathtt{r2,y2}})) \vdash$
     $\_\_\langle true, true\rangle \supset (blOnBl_{\mathtt{x,y}} \wedge empty_{\mathtt{r}}) \supset [\_\_(\alpha_1, \alpha_2)](\neg blHold_{\mathtt{r,y}})$

Assume the hypotheses; using the inference rule (2) of VLRL, we derive

$$\_\_\langle (blOnBl_{\mathtt{x1,y1}} \wedge empty_{\mathtt{r1}}), true\rangle \supset [\_\_(\alpha_1, \alpha_2)](\_\_\langle \neg blHold_{\mathtt{r1,y1}}, true\rangle)$$

Using the structural property of attributes (17):

$$(blOnBl_{\mathtt{x,y}} \wedge empty_{\mathtt{r}}) \supset [\_\_(\alpha_1, \alpha_2)](\_\_\langle \neg blHold_{\mathtt{r,y}}, true\rangle).$$

Finally using the structural property of attributes (18) we get the result.

If a block is under another block, it will stay under that block until the block that is on it is caught:

$$blOnBl_{\mathtt{x,y}} \supset \mathrm{A}(blOnBl_{\mathtt{x,y}} \ \mathrm{W}\ blHold_{\mathtt{x}}) \ .$$

Applying the second interface inference rule of temporal logic and VLRL (6), it is enough to prove

$$\{blOnBl_{\mathtt{x,y}} \wedge \neg blHold_{\mathtt{x}} \supset [\alpha](blOnBl_{\mathtt{x,y}} \vee blHold_{\mathtt{x}}) \mid \alpha \text{ action term}\} \ .$$

Using structural induction, we have to prove:

- *constants* are derived like in the first example.

- *rules*:

  1. $(blOnBl_{\mathtt{x,y}} \wedge \neg blHold_{\mathtt{x}}) \supset [\mathtt{pickup_{u,z}}](blOnBl_{\mathtt{x,y}} \vee blHold_{\mathtt{x}})$ The property is derived directly from (9) and (4).
  2. $(blOnBl_{\mathtt{x,y}} \wedge \neg blHold_{\mathtt{x}}) \supset [\mathtt{putdown_{u,z}}](blOnBl_{\mathtt{x,y}} \vee blHold_{\mathtt{x}})$ The property is derived directly from (10) and (4).
  3. $(blOnBl_{\mathtt{x,y}} \wedge \neg blHold_{\mathtt{x}}) \supset [\mathtt{unstack_{u,z1,z2}}](blOnBl_{\mathtt{x,y}} \vee blHold_{\mathtt{x}})$ We distinguish two cases: (i) If $empty_{\mathtt{u}} \wedge blClear_{\mathtt{x}}$ then apply (11), (16) and (4) to obtain the result; (ii) If $\neg empty_{\mathtt{u}} \vee \neg blClear_{\mathtt{x}}$ we apply the *contrapositive law* of propositional logic to (13) and (4).
  4. $(blOnBl_{\mathtt{x,y}} \wedge \neg blHold_{\mathtt{x}}) \supset [\mathtt{stack_{u,z1,z2}}](blOnBl_{\mathtt{x,y}} \vee blHold_{\mathtt{x}})$ The property is derived directly from (12) and (4).

- *state operations*:

  5. $((blOnBl_{\mathtt{x1,y1}} \wedge \neg blHold_{\mathtt{x1}}) \supset [\alpha_1](blOnBl_{\mathtt{x1,y1}} \vee blHold_{\mathtt{x1}}),$
     $(blOnBl_{\mathtt{x2,y2}} \wedge \neg blHold_{\mathtt{x2}}) \supset [\alpha_2](blOnBl_{\mathtt{x2,y2}} \vee blHold_{\mathtt{x2}}) \vdash$
     $\_\_\langle true, true\rangle \supset (blOnBl_{\mathtt{x,y}} \wedge \neg blHold_{\mathtt{x}}) \supset$
     $[\_\_(\alpha_1, \alpha_2)](blOnBl_{\mathtt{x,y}} \vee blHold_{\mathtt{x}})$

Assume the hypotheses; using the inference rule (2) of VLRL, we derive

$$\_\_\langle blOnBl_{\mathtt{x1,y1}}, true\rangle \supset [\_\_(\alpha_1, \alpha_2)](\_\_\langle (blOnBl_{\mathtt{x1,y1}} \vee blHold_{\mathtt{x1}}), true\rangle)$$

Using the structural properties of attributes (19):

$$blOnBl_{\mathtt{x1,y1}} \supset [\_\_(\alpha_1, \alpha_2)](\_\_\langle (blOnBl_{\mathtt{x1,y1}} \vee blHold_{\mathtt{x1}}), true\rangle)$$

Finally using the structural properties of attributes (20) we get the result.

7 Related work

Work directly related to our proposal includes the modal μ-calculus proposed by Lechner for reasoning about object-oriented Maude specifications [5], and Denker's object-oriented distributed temporal logic [2]. The main difference of our approach is that we do not restrict ourselves to a special type of state configuration, thus including structures that need not to be associative or commutative. Therefore, we are able to treat any system specified in Maude, and not only object-oriented systems.

Comparing with the locally distributed temporal logic of Denker's work, we are interested in global properties of a system, object-oriented or not, instead of properties as seen by a local object. Then, we use Kripke frames based on state transitions as models instead of the concurrent labeled event structures used by Denker. While we emphasize the relation between properties of global states and properties of the substates, mainly when state transitions take place, Denker's work deals with the concurrent nature of a distributed system, mainly with causality, conflict and concurrency properties.

Comparing with Lechner's work, our atomic formulae are more general than simple propositions asserting the presence of messages and/or objects. This is the main reason behind the introduction of observation attributes in order to express more interesting properties of the system. Our approach admits using different specification logics for the definition of properties, by giving inference rules of the logic and VLRL, then the verification process is acomplished by the deductive system of VLRL. On the other hand, Lechner proposes a three level approach to specification: at the most abstract level, properties are expressed using the μ-calculus; at the intermediate level, formulae are blended with propositions on object states; and at the concrete level, Maude is used. The relation between these levels is achieved by a refinement process.

References

[1] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada, *A Maude Tutorial*, Computer Science Laboratory, SRI International, March 2000. `http://maude.csl.sri.com/tutorial`

[2] G. Denker. From rewrite theories to temporal logic theories, in: C. Kirchner and H. Kirchner (eds.), *Proc. Second Int. Workshop on Rewriting Logic and its Applications, Pont-à-Mousson, France*, Electronic Notes in Theoretical Computer Science, Vol. 15, Elsevier Science, September 1998.

[3] J. L. Fiadeiro, T. Maibaum, N. Martí-Oliet, J. Meseguer, and I. Pita. Towards a verification logic for rewriting logic. In D. Bert, C. Choppy, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques, WADT'99, Chateau de Bonas, France, September 1999, Selected Papers*, LNCS 1827, 438–458. Springer-Verlag, 2000.

[4] R. Goldblatt, *Logics of Time and Computation*, CSLI Lecture Notes 7, Center for the Study of Language and Information, Second edition, 1992.

[5] U. Lechner, *Object-Oriented Specification of Distributed Systems*, Ph. D. Dissertation, Universitat Passau, June 23, 1997.

[6] P. Lincoln, N. Martí-Oliet, and J. Meseguer, Specification, transformation, and programming of concurrent systems in rewriting logic, in: G. E. Blelloch *et al.* (eds.), *Specification of Parallel Algorithms, DIMACS Workshop, May 1994*, American Mathematical Society, 1994, 309–339.

[7] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, *Theoretical Computer Science* 96, 1992, 73–155.

[8] J. Meseguer, A logical theory of concurrent objects and its realization in the Maude language, in: G. Agha, P. Wegner, and A. Yonezawa (eds.), *Research Directions in Concurrent Object-Oriented Programming*, The MIT Press, 1993, 314–390.

[9] C. Stirling, Modal and temporal logics, in: S. Abramsky, D. Gabbay, and T. Maibaum (eds.), *Handbook of Logic in Computer Science*, Vol. II, Oxford University Press, 1992, 478–563.

A Observed programs

The *observed program* of the number of blocks example is given by the following rewrite theory, which is a conservative extension of the OO-BLOCKSWORLDN theory, since no additional rewrite rule is added:

```
(omod OO-BLOCKSWORLDN-NUMBLOCKS-OBSERVED is
  protecting OO-BLOCKSWORLDN .

  op #blTable : Configuration -> MachineInt .
  op #blOnBl : Configuration -> MachineInt .
  op #blHold : Configuration -> MachineInt .
  op #blocks : Configuration -> MachineInt .
  op #empty : Configuration -> MachineInt .
  op #robots : Configuration -> MachineInt .

  var C : Configuration .
  vars X Y : BlockId .
  var R : RobotId .
  var M : Msg .
  var O : Object .

  eq #blTable(< X : Block | on : table >) = 1 .
  eq #blTable(< X : Block | on : catchd >) = 0 .
  eq #blTable(< X : Block | on : Y >)  = 0 .
  eq #blTable(< R : Robot | >) = 0 .
  eq #blTable(none) = 0 .
  eq #blTable(M C) = #blTable(C) .
  ceq #blTable(O C) = #blTable(O) + #blTable(C) if C =/= none .

  eq #blOnBl(< X : Block | on : table >) = 0 .
  eq #blOnBl(< X : Block | on : catchd >) = 0 .
  eq #blOnBl(< X : Block | on : Y >)  = 1 .
  eq #blOnBl(< R : Robot | >) = 0 .
  eq #blOnBl(none) = 0 .
  eq #blOnBl(M C) = #blOnBl(C) .
  ceq #blOnBl(O C) = #blOnBl(O) + #blOnBl(C) if C =/= none .

  eq #blHold(< X : Block | >) = 0 .
  eq #blHold(< R : Robot | hold : empty >) = 0 .
  eq #blHold(< R : Robot | hold : X >) = 1 .
  eq #blHold(none) = 0 .
  eq #blHold(M C) = #blHold(C) .
  ceq #blHold(O C) = #blHold(O) + #blHold(C) if C =/= none .

  eq #blocks(C) = #blTable(C) + #blOnBl(C) + #blHold(C) .

  eq #empty(< X : Block | >) = 0 .
  eq #empty(< R : Robot | hold : X >) = 0 .
  eq #empty(< R : Robot | hold : empty >) = 1 .
```

```
  eq #empty(none) = 0 .
  eq #empty(M C) = #empty(C) .
  ceq #empty(O C) = #empty(O) + #empty(C) if C =/= none .

  eq #robots(C) = #blHold(C) + #empty(C) .
endom)
```

The *observed program* of the blocks position example is given by:

```
(omod OO-BLOCKSWORLDN-POSITION-OBSERVED is
  protecting OO-BLOCKSWORLDN .

  op blTable : Configuration Oid -> Bool .
  op blClear : Configuration Oid -> Bool .
  op blOnBl : Configuration Oid Oid -> Bool .
  op blHold : Configuration Oid Oid -> Bool .
  op blHold : Configuration Oid -> Bool .
  op empty : Configuration Oid -> Bool .

  var C : Configuration .
  vars X Y Z : BlockId .
  vars R S : RobotId .
  var M : Msg .
  var O : Object .

  eq blTable(< X : Block | on : table >,X) = true .
  eq blTable(< X : Block | on : Y >,X) = false .
  ceq blTable(< Y : Block | >,X) = false if X =/= Y .
  eq blTable(< R : Robot | >,X) = false .
  eq blTable(none,X) = false .
  eq blTable(M C,X) = blTable(C,X) .
  ceq blTable(O C,X) = blTable(O,X) or blTable(C,X) if C =/= none .

  eq blClear(< X : Block | under : clear >,X) = true .
  eq blClear(< X : Block | under : Y >,X) = false .
  ceq blClear(< Y : Block | >,X) = false if X =/= Y .
  eq blClear(< R : Robot | >,X) = false .
  eq blClear(none,X) = false .
  eq blClear(M C,X) = blClear(C,X) .
  ceq blClear(O C,X) = blClear(O,X) OR blClear(C,X) if C =/= none .

  eq blOnBl(< X : Block | on : table >,X,Y) = false .
  eq blOnBl(< X : Block | on : Y >,X,Y) = true .
  ceq blOnBl(< X : Block | on : Z >,X,Y) = false if Y =/= Z .
  ceq blOnBl(< Z : Block | >,X,Y) = false if X =/= Z .
  eq blOnBl(< R : Robot | >,X,Y) = false .
  eq blOnBl(none,X,Y) = false .
  eq blOnBl(M C,X,Y) = blOnBl(C,X,Y) .
  ceq blOnBl(O C,X,Y) = blOnBl(O,X,Y) or blOnBl(C,X,Y) if C =/= none .
```

```
  eq blHold(< X : Block | >,R,Y) = false .
  eq blHold(< R : Robot | hold : empty >,S,Y) = false .
  eq blHold(< R : Robot | hold : X >,R,X) = true .
  ceq blHold(< R : Robot | hold : X >,R,Y) = false if X =/= Y .
  eq blHold(none,R,Y) = false .
  eq blHold(M C,R,Y) = blHold(C,R,Y) .
  ceq blHold(O C,R,Y) = blHold(O,R,Y) or blHold(C,R,Y) if C =/= none .

  eq blHold(< X : Block | >,Y) = false .
  eq blHold(< R : Robot | hold : empty >,Y) = false .
  eq blHold(< R : Robot | hold : X >,X) = true .
  ceq blHold(< R : Robot | hold : X >,Y) = false if X =/= Y .
  eq blHold(none,Y) = false .
  eq blHold(M C,Y) = blHold(C,Y) .
  ceq blHold(O C,Y) = blHold(O,Y) or blHold(C,Y) if C =/= none .

  eq empty(< X : Block | >,R) = false .
  eq empty(< R : Robot | hold : empty >,R) = true .
  ceq empty(< R : Robot | hold : empty >,S) = false if R =/= S .
  eq empty(< R : Robot | hold : X >,S) = false .
  eq empty(none,X) = false .
  eq empty(M C,X) = empty (C,X) .
  ceq empty(O C,X) = empty (O,X) or empty (C,X) if C =/= none .
endom)
```