Operational Interpretation of the Requirements Specification Language ALBERT Using Timed Rewriting Logic*

Nasreddine Aoumeur Gunter Saake
ITI, FIN, Otto-von-Guericke-Universität Magdeburg
Postfach 4120, D–39016 Magdeburg
E-mail: {aoumeur|saake}@iti.cs.uni-magdeburg.de
Tel.: ++49-391-67-18659 Fax: ++49-391-67-12020

Abstract. ALBERT is a formal semi-graphical specification language for real-time distributed systems in general, and is particularly suitable for eliciting requirements engineering in distributed information systems conceived as a community of agents. Moreover, ALBERT has received an adequate property-oriented semantics in terms of real-time object oriented (OO) temporal logic. The contribution of the present paper is to provide this widely accepted language with a more operational semantics based on (timed) rewriting logic and described using an appropriate extension of the MAUDE language. Firstly, the real-time as well as the action composition aspects are captured mainly by taking benefits of the Wirsing's and al. works on Timed MAUDE and the process based control of rewriting in MAUDE. Secondly, for capturing the intra-concurrency and the notions of action, state and information perception that present the ALBERT language, we propose an adequate 'splitting/recombination' axiom that allows, on the one hand, to split and recombine the (usually indivisible) object state at a need; on the other hand, it allows for a clean separation between local and external features (attributes and events) of a given agent, which permit the construction of complex systems as interacting components without violating the encapsulated aspects of each agent. The translation is illustrated through the Generalized Railroad Crossing case study.

1 Introduction

ALBERT[1] [DB95] is one of the successful language for *naturally* eliciting requirements engineering, in real-time distributed (information) systems, without loose of *formality*. Indeed, on the one hand, ALBERT language proposes a rich graphical as well as textual ontology of concepts and constructions that allow to conceive such systems as a community of agents —regarded here as objects cooperating

* In Proc. of 5th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'99), In Join with CAiSE*99 Conference, Heidelberg, Germany, Presses Universitaires de Namur, 1999.

[1] As an acronym for 'Agent-oriented Language for Building and Eliciting Requirements for Real-Time systems'.

through explicit interfaces (i.e. import/export parts) and coordination rules using action, state and information perception. On the other hand, the ALBERT language is provided with a well-founded property-oriented semantics [DB95] expressed in an appropriate real-time extension of the OSL (i.e. Object Specification Logic) 'object' temporal logic [SSC95].

In this paper we are particularly interested on relating the rich concepts and constructions of the ALBERT and the MAUDE languages [Mes93]. More precisely, we propose to translate the ALBERT specifications into an appropriate extension of the MAUDE language, and provide in this way the ALBERT language with a more operational semantics expressed into rewriting logic [Mes92].

The needed MAUDE extensions that propose the paper are : first, for dealing with the real-time constraints as well as action composition aspects, we take profit of two recent proposals, namely the timed rewriting logic [KW97] and the extension of MAUDE with a process language for controlling its rewriting process [WK96]. Second, for exhibiting intra-object concurrency as well as distinguishing between local and external features of each agent we propose an appropriate 'splitting / recombining' axiom of the object state at a need.

The rest of the paper is organized as follows: in the next section, the main notions of the ALBERT language using as the generalized railroad crossing problem (shortly, GRC) are reviewed. The third presents an overview of the MAUDE language. The main section presents in incremental and constructive way the translation of the ALBERT specifications into the (extended) MAUDE language. Finally, we conclude the paper by some remarks and future work.

2 ALBERT Overview Using a Running Example

In what follows, the main concepts and constructions of the ALBERT language are illustrated using the Generalized Railroad Crossing Problem (shortly GRC); however, a complete presentation of this language is outside the scope of this paper and the interested reader may consult [DB95]. Basically, a specification in ALBERT is made up of (i) a graphical part where the vocabulary is *declared* (together with some general properties) and (ii) a textual part where the admissible behaviour is described.

2.1 Graphical Declaration

Figure 1 depicts the description of the society of agents associated with the GRC problem. Each agent is represented by an oval: shaded oval represents a *class* and plain ovals represent *individual* agents; therefore, for the GRC problem, we have several *Train* agents, one *Controller* agent and one *Gate* agent. The state components (i.e. the different attributes) of an agent are represented by rectangles with their names and associated types, while the actions are graphically depicted with ovals. Moreover, the graphical respresentation gives information about the responsibility that the different agents have on state components and actions. Due to space limitation, for the meaning of each action and attribute the reader is advised to consult [DBDZ97].

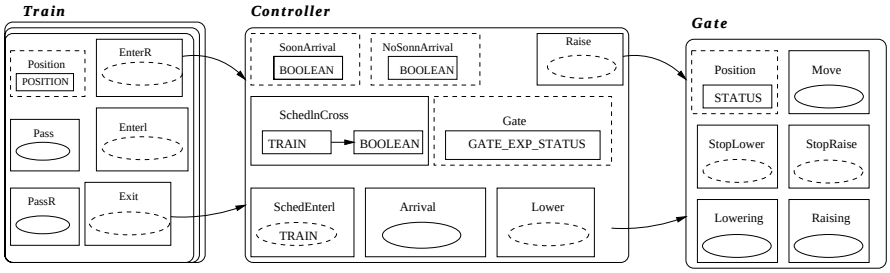

Fig. 1. Declaration of the Agents of the GRC Society

2.2 Textual Specification of Constraints

The textual constraints in an ALBERT specification allow for limiting the life cycle of different agents only to those that are considered as admissible. There are three kinds of constraints: The basic constraints permit to initialize the values of different attributes optionally; the local constraints specify the internal behaviour of each agent and finally, the cooperative constraints state of different features (actions or attributes) of a given agent that can be imported or exported to other ones (eventually) under some conditions.

The GRD - Textual Constraints

BASIC CONSTRAINTS
INITIAL VALUATION
Position = Elsewhere
LOCAL CONSTRAINTS
EFFECTS OF ACTIONS
EnterR : Position := R
EnterI:Position := I
CAPABILITY
(Pass / Position = Elsewhere)
ACTION COMPOSITION
Pass PassR ; Exit
PassR Enter ; EnterI
ACTION DURATION
Epsilon1 $\leq$ PassR $\leq$ Epsilon2
COOPERATION CONSTRAINTS
ACTION INFORMATION
(EnterR.c / TRUE)
(Exit.c / TRUE)

Gate

BASIC CONSTRAINTS
INITIAL VALUATION
Position = Up
LOCAL CONSTRAINTS
EFFECTS OF ACTIONS
c.Lower : Position := GoingDown
StopLower: Position := Down
c.Raise : Position := GoingUp
StopRaise : Position := Up
CAPABILITY
(StopLower / Position = GoingDown)
(StopRaise / Position = GoingUp)
ACTION COMPOSITION
Move Lowering ; Raising
Lowering c.Lower ;
StopLower
Raising c.Raise ;
StopRaise
ACTION DURATION
0 $\leq$ Lowering $\leq$ GammaDown
0 $\leq$ Raising $\leq$ GammaUp
COOPERATION CONSTRAINTS
ACTION PERCEPTION
$\mathcal{X}$ (c.Lower / Position = Up)
$\mathcal{X}$ (Exit.c / TRUE)

Controller

INITIAL VALUATION SchedInCross[t] = FALSE *(* The intial expected status of the Gate is Up. *)*
SoonArrival = FALSE *(*There is no scheduled train to pass the crossing at the initial time. *)*
SoonArrival = TRUE *(* There is no NoSoonArrival at the initial time. *)*
LOCAL CONSTRAINTS
STATE BEHAVIOUR
NoSoonArrival $\Longleftrightarrow$
$\exists$: $\neg$ SchedInCross[t] $\wedge$
$\Diamond_{=GammaDown}$ SchedCross[t]

(SoonArrival means that a train which suit up is not scheduled now but which will be scheduled within* GammaDown *seconds. *)*

SoonArrival $\Longleftarrow$
$\exists$: ($\neg$ SchedInCross[t] $\wedge$
$\Box_{GammaUp+Delta+GammaDown}$
SchedCross[t]) *(* NoSoonArrival means that there is no a train which scheduled to cross now neither during the next* GammaUp + Delta + GammaDown *seconds. *)*

EFFECTS OF ACTIONS

SchedEnterI(t) : SchedInCross[t] := TRUE *(* After the occurrence of a SchedEnterI event for a train t, this train belongs to the list of trains scheduled to pass the crossing. *)*

t.Exit : SchedlnCross[t] := FALSE

Lower : Gate := Down

Raise : Gate := Up

CAPABILITY

$\mathcal{X}$ $\mathcal{O}$(Raise/Gate = Down $\wedge$ NosoonArrival)

$\mathcal{X}$ $\mathcal{O}$(Lower/Gate = Up $\wedge$ SoonArrival)
(Lower/Gate = Up $\wedge$ SoonArrival)

ACTION COMPOSITION

Arrival(t) t.EnterR ;
SchedEnter(t)

ACTION DURATION
Arrival = Epsilon1

COOPERATION CONSTRAINTS

ACTION PERCEPTION
(t.EnterR / TRUE)
(t.Exit / TRUE)

ACTION INFORMATION
(Raise.g / TRUE)
(Lower.g / TRUE)

3 Rewriting Logic and MAUDE language

Firstly introduced by J. Meseguer in [Mes92], rewriting logic is nowadays one of the widely accepted unified model of concurrency. This logic is based on two straightforward, but yet powerful ideas: first, in contrast to the equational theory, rewriting logic interprets each rewrite rule not as an oriented equation (in the equational theory) rather as a change or becoming through a powerful categorical foundation. Second, it proposes to perform the rewriting process with maximal of concurrency on the basis of the four powerful inference rules. In addition, one of the most advantage of rewriting logic is its straightforward and sound capability of unifying the object oriented programming paradigm the concurrency. Such conceptualization go together with a very rich OO programming/specification language called MAUDE.

In MAUDE, the structural as well as the functional aspects are specified algebraically using notations that are very similar to the OBJ [GWM+92] ones, while the dynamic aspects are captured by appropriate modules called system modules that can be elegantly denoted using object modules.

The object states in MAUDE are conceived as terms —precisely as tuples— of the form $\langle Id : C | atr_1 : val_1, .., atr_k : val_k \rangle$; where Id stands for the object identity that we assume of appropriate sort denoted OId, C for the object class of sort CId; $atr_1, ..., atr_k$ denote the attribute identifiers and $val_1, ..., val_k$ are their respective (actual) values, they are respectively typed by AId and $Value$. The messages (i.e. method invocation also called action in ALBERT and event in TROLL) are regarded as operations sent or received by objects. Object and message instances flow together in the so-called configuration, which is no more than a multiset[2] of messages and (a set of) objects. The simplified form of such configuration, described as a functional module takes the following form [Mes93]:

[2] The associated union (associative and commutative) operator is denoted '_ _'.

```
fmod Configuration is
 protecting ID          **** provides Oid, Cid and Aid .
 sorts Configuration Object Msg .
 subsorts Oid < Value .
 subsorts Attribute < Attributes .
 subsorts Object Msg < Configuration .
 op _:_:Aid Value → Attribute .
 op _,_ : Attribute Attributes → Attributes [associ. commu. Id:nil]
 op ⟨_:_|_⟩ : Oid Cid Attributes → Object.
 op _ _ : Configuration Configuration → Configuration [assoc comm id:null].
endfm.
```

The effect of messages on objects to which they are addressed is captured by appropriate rewrite rules. The general form of such rules, known as a communication event pattern, takes the following form:

$$M_1 ... M_n \langle I_1 : C_1 | atts_1 [3] \rangle ... \langle I_m : C_m | atts_m \rangle \Rightarrow M'_1 ... M s'_q \langle I_{i1} : C'_{i1} | atts'_{i1} \rangle ... \langle I_{ik} : C'_{im} | atts'_{ik} \rangle \langle J_1 : D_1 | atts''_1 \rangle ... \langle J_{ik} : D_p | atts''_p \rangle \text{ if } C$$

In MAUDE, subclasses may be specified using the notion of class inheritance when they involve no methods overriding else module inheritance are used with different possibilities of redefining (renaming, redefining, etc). Finally, noting that in MAUDE object instances are created with control of the uniqueness of their identities (modeled as natural counter) using an appropriate rewrite rule.

4 Translating ALBERT Specification into an Extended MAUDE

The purpose of this section consists in translating step by step the different concepts and constructions of the textual part of the ALBERT language into a respective incremental extension of the MAUDE language. For this aim, in a first attempt we show how the (simple) MAUDE language allows for capturing, besides the structural and the functional ALBERT aspects, the associated OO signature of different agents in the system as well as the effect of different actions; however actions valuation is achieved without any form of intra-object concurrency that provides (the associated semantics of) the ALBERT language and without respect of the encapsulation property. The objective of the next subsection is to overcome these crucial limitations and take into account the action, state and information perception. The basic ideas for achieving that consist in introducing the so-called 'splitting/recombination' axiom. Third, we deal with different forms of actions composition in ALBERT language by mainly taking profit of the extension developed by Wirsing and al. in [WNL95] that allows for controlling the rewriting process in MAUDE specification using a simple but powerful message languages. Finally, for dealing the real-time constraints, we shortly report on how to use the

[3] $atts_i$ is a simplified notation for $atr_{i1} : val_{i1}, ..., atr_{ik} : val_{ik}$.

well-founded timed rewriting logic (TRL) [KW97] and its corresponding Timed MAUDE.

4.1 First Attempt of Translating ALBERT specifications to MAUDE

As it can be clearly understood, the functional as well as the static aspects of given ALBERT specification can be specified, in a more structured and semantically founded way, using the functional level of the MAUDE language.

For the specification of the structure or the OO signature of different agents with the effect of their actions, we propose the following straightforward translation:

- Associate with each agent a corresponding class those attributes are the state components of the agent and those messages are the actions declared in the agent (with adding to their parameters the identity of the invoked agent).
- Associate with each effect of an agent action a corresponding rewrite rules. In such rewrite rules, the values of the attributes concerned by the change are the initial ones in the left hand side of the rewrite rule and the ones specified in the action primitive (as assignement) in the the right hand side.
- The capability of each action is included as a condition in the corresponding rewrite rule.
- For actions that are perceived in other agents (included in the ACTION PERCEPTION) they have to involve more than one class (or agent) state in their corresponding rewrite rules; becasue they involve a state component change not only in the agent in which they are declared by also in the perceived agent.

Example 1. Following these guidelines, it is not difficult to translate the GRC specification in MAUDE that takes the following form:

```
omod GRC-Spec is
 protecting GRC-Structure  bool.
 class Train  |  Position : Position .
 class Controler  |  SoonAr: bool,  NSoonAr: bool, SInCr:Tr[t], Gate: Gate .
 class Gate  |  Position : Status .
 msgs Pass  PassR  EnterI  EnterR  Exit : OId → Msg .
 msgs SchedEnterI  Arrival  Lower  Raise: OId → Msg .
 msg SchedEnterI : OId  OId → Msg .
 msgs Move StopRaise  Lowering  Raising: OId → Msg .
 vars Pos : Position .
 Vars  T G C : OId .
 ********* The Train (local) behaviour.
 rl EnterR(T)⟨T : Train|Position : E⟩ ⇒ ⟨T : Train|Position : R⟩
 rl EnterI(T)⟨T : Train|Position : E⟩ ⇒ ⟨T : Train|Position : I⟩
 ********* The Gate (local) behaviour.
 rl StepLower(G)⟨G : Gate|Position : Var⟩ ⇒ ⟨G : Gate|Position : Dw⟩
  if Var ≠ GD[4]
```

rl $StopRaise(G)\langle G : Gate|Position : Pos\rangle \Rightarrow \langle G : Gate|Position : Up\rangle$ **if** $Pos \neq GU$

`********* The Control (local) behaviour.`

rl $SchedEnterI(C,t)\langle C : Controler|SchedIn[t] : False, ATTS^5\rangle$
$\Rightarrow \langle C : Controler|SchedIn[t] : True, ATTS\rangle$

`********* The Interaction between differents agents.`

rl $Exit(T)^6\langle T : Train|Position : I\rangle\langle C : Controler|ShedInC[T] : True, ATTS\rangle$
$\Rightarrow \langle T : Train|Position : E\rangle\langle C : Controler|ShedInC[T] : False, ATTS\rangle$

rl $Lower(C)\langle C : Controler|Gate : Up, SoonAr : True, ATTS\rangle\langle G : Gate|Position : Up\rangle$
$\Rightarrow \langle C : Controler|Gate : Dw, SoonAr : True, ATTS\rangle\langle G : Gate|Position : Dw\rangle$

rl $Raise(C)\langle C : Controler|Gate : Dw, NSoonAr : True, ATTS\rangle\langle G : Gate|Position : Up\rangle$
$\Rightarrow \langle C : Controler|Gate : Up, NSoonAr : True, ATTS\rangle\langle G : Gate|Position : Up\rangle$

endom.

4.2 On the Modeling of the Intra-Object Concurrency and the Agents Interaction

In order to be more close to the ALBERT philosophy, we have to make an explicit separation between the different agents. The first step, in this sense, consists in avoiding this 'global' notion of configuration that *violates* the encapsulation property. For this aim, we propose the following constructions:

- Instead of incorporating the class identifier in each object state, we propose to take it as a common parameter. We do so by conceiving an agent structure rather as a pair of the form $(Agent\_Identifier, associated\_subconfiguration^7)$; where the $associated\_subconfiguration$ should contain only message and object state (without class identifier) instances of the associated agent.
- For composing or relating such different agent communities, we propose to use an additional associative and commutative operator denoted as $\otimes$.

For the description of this 'agent' OO conceptualization, we use only system modules; because, as given above, in the object modules the class structure is predefined. The use of system modules necessitate the introduction of some sort constraints for controlling the compatibilities of different structures (see the translation of the object modules into system modules in [Mes93] section 4.2).

Example 2. The precise description of this new conceptualization for agents of the GRC, with the associated (local) rewrite rules, can be given as follows:

mod GRC-Spec **is**
extending CONFIGURATION .
protecting GRC-Structure .
sorts Conf_Train Conf_Controler Conf_Gate < Configuration.
subsorts Position State Gate < Value .
subsorts Train-Id Controler-Id Gate-Id < Agent-Id
subsorts Agent-Id < CId .

[7] Noting that the notion of subconfiguration has been well developed in [WNL95] but without an explicit separation from the global configuration.

subsorts Train Controler Gate < Agent .
subsorts Object_train Object_gate Object_controler < Object .
subsorts Msg_Train Object_Train < Conf_Train
subsorts Msg_Gate Object_Gate < Conf_Gate
subsorts Msg_Controler Object_Controler < Conf_Control
op (_,_) Agent-Id Configuration → Agent
op Pass PassR EnterI EnterR Exit : Train-Id → Msg_Train.
op SchedEnterI Arrival Lower Raise: Controler-Id → Msg_Controler .
op SchedEnterI : Control-Id Train-Id → Msg_Controler .
op Move StopRaise Lowering Raising : Gate-Id → Msg_Gate .
vars T : Train-Id C : Controler-Id G : Gate-Id .
vars Pos : Position Posi : Position1 .
op _⊗_ Agent Agent → Agent [assoc. comm. Id:null]
sct $\langle G|Position : Pos\rangle$: Object_Gate . (* **sct1** *)
sct $\langle T|Position : Posi\rangle$: Object_Train . (* **sct2** *)
sct $\langle C|SoonAr : bool, NSoonAr : bool, SchInC : Tr[t], Gate : Gate\rangle$: Object_Controler . (* **sct3** *)
sct Conf_Train Conf_Train : Conf_Train . (* **sct4** *)
sct Conf_Gate Conf_Gate : Conf_Gate . (* **sct5** *)
sct Conf_Control Conf_Control : Conf_Controler . (* **sct6** *)
sct (Gate_Id, Conf_Gate) : Gate . (* **sct7** *)
sct (Controler_Id, Conf_Controler) : Controler . (* **sct8** *)
sct (Train_Id, Conf_Train) : Train . (* **sct9** *)
sct Conf_Train Conf_Control : Undefined . (* **sct10** *)
sct Conf_Train Conf_Gate : Undefined . (* **sct11** *)
sct Conf_Control Conf_Gate : Undefined . (* **sct12** *)

********* The Train (local) behaviour.
rl $EnterR(T)\langle T|Position : E\rangle \Rightarrow \langle T|Position : R\rangle$
rl $EnterI(T)\langle T|Position : E\rangle \Rightarrow \langle T|Position : I\rangle$
rl $Enter(T)\langle T|Position : I\rangle \Rightarrow \langle T|Position : E\rangle$

********* The Gate (local) behaviour.
rl $StepLower(G)\langle G|Position : GD\rangle \Rightarrow \langle G|Position : Dw\rangle$
rl $StopRaise(G)\langle G|Position : GU\rangle \Rightarrow \langle G|Position : Up\rangle$

********* The Control (local) behaviour.
rl $Lower(C)\langle C|Gate : Up, SoonAr : True, ATTS\rangle \Rightarrow \langle C|Gate : Dw, SoonAr : True, ATTS\rangle$
rl $Raise(C)\langle C|Gate : Dw, NSoonAr : True, ATTS\rangle \Rightarrow \langle C|Gate : Up, NSoonAr : True, ATTS\rangle$
rl $SchedEnterI(C,t)\langle C|SchedIn[t] : False, ATTS\rangle \Rightarrow \langle C|C|SchedIn[t] : True, ATTS\rangle$
endom.

Remark 1. The sort constraints $sct1$ until $sct9$ requires that each sub-configuration have to be compatible with the actions and the object states of its associated

agent. In more details, the sort constraints $sct1, sct2, sct3$ determine precisely the (object) state form (i.e. the attributes identifiers and the sorts of their values) of each of the three agents that are all subsorts of the $Object$. The sort constraints $sct4, sct5, sct6$ define how to construct (inductively) w.r.t to the configuration operator '_ _' each of the three sub-configurations (associated with the three agent). The sort constraints $sct7, sct8, sct9$ also determine precisely the sort of each agent as a pair of the agent-identifier and its corresponding configuration (sort). Finally, in order to avoid direct composition (through the '_ _') of sub-configurations belonging to different agents, we have added the $sct10, scr11, sct12$ that exclude such composition.

The Intra-object Concurrency As mentioned above, the form of the MAUDE's object state as an indivisible tuple of the $\langle Id|atr_1 : V_1, ..., atr_n : V_n\rangle$ avoids any possibilities of performing concurrently two events dealing with the *same object* and acting on different attributes. To overcome this limit, we propose a simple but powerful axiom that allows to split (or recombine) the different components (i.e. attributes) of a given object at a need. This axiom that have to be included in the already presented 'state-object' specification can be described as follows.

vars _ _ : $attr_1, attr_2$: Attributes
ax $\langle Id|attr_1, attr_2\rangle = \langle Id|attr_1\rangle\langle Id|attr_2\rangle$

Example 3. Due to this axiom, and as a first step for exhibiting intra-object concurrency in the controler agent, we have to drop the attributes variable (that refers to the unchanged attributes denoted by $ATTS$) as well as the class identifier from the above described rewrite rules of this agent. Such rewrite rules now become:

********* The Control (local) behaviour.
rl $Lower(C)\langle C|Gate : Up, SoonAr : True\rangle \Rightarrow \langle C|Gate : Dw, SoonAr : True\rangle$
rl $Raise(C)\langle C|Gate : Dw, NoSoonAr : True\rangle \Rightarrow \langle C|Gate : Up, SoonAr : True\rangle$
rl $SchedEnterI(C,t)\langle C|SchedIn[t] : False\rangle \Rightarrow \langle C|C|SchedIn[t] : True\rangle$

With that it is not difficult to see, for instance, that the actions $SchedEnterI(C,t)$ and $Lower(C)$ may be processed in parallel.

On the Interaction Between Different Agents The second benefit of this axiom is the possibility to split up the state component of a given agent into a local part —and therefore *hidden* to the other agents part— and *observed* part that can be modified by the other agents. We use the notation 'observed_State' and 'hidden_State' to distinguish these two agent state part. This distinction can be easily conceptualized by introducing for each agent state sort, two correponding subsorts: one for the observed part and the other for the hidden part. On the other hand, we have to precise the form of each part using a sort constraint.

Example 4. In the GRC study, since the attribute $SchedInCross$ of the Controller agent and the attribute *position* of the Gate agent can be respectively manipulated by the event *Raise* declared in the train agent and the events *lower* or *Raise* declared in the Controller agent, we can explicitly described them as observed attributes using the following notation (to which we give hereafter the inherent semantics):

fmod State_Splitting **is** .
subsorts observed_state_Controler hidden_state_Controler < Object_Controler
subsorts observed_state_Gate < Object_Gate
sct $\langle G|Position : Pos\rangle$: observed_state_Gate
sct $\langle T|Position : Posi\rangle$: observed_state_Train
sct $\langle C|ShedInCross : Tr[t]\rangle$: observed_state_Controler
sct $\langle C|SoonAr : bool, NoSoonAr : bool, Gate : Exp{-}Gate\rangle$: hidden_state_Controler
endfm

Similarly, we can split the messages of a given agent into internal or local events and external as imported/exported messages. This distinction allow for capturing the notion of action perception (i.e. the exported messages) and action information (i.e. the imported messages). In addition, to be more close to the ALBERT specification, for the external messages it is possible to add to their arguments the identifiers of the agents that can be perceived and the conditions (on the associated attributes that have to be respected). The corresponding syntax may declared as follows:
op:$Mes\_Id : Message\_Parameters \quad List\_of\_Agent\_Identifiers \quad Condition\_on\_Atributes \rightarrow External\_Mes$

Example 5. In our running example, the messages that are perceived by other agents can be declared as follows (the other are of course declared as local).

fmod State_Splitting **is** .
subsorts Local_Msg_Train External_Msg_Train < Msg_Train
op Pass PassR EnterI : Train-Id → Local_Msg_Train.
op EnterR Exit : Train-Id Controler-Id → External_Msg_Train.
op SchedEnterI Arrival : Control-Id → Local_Msg_Control .
op Lower : Controler-Id Gate-Id $\langle Position : Up\rangle$[8] → External_Msg_Control .
op Raise : Controler-Id Gate-Id $\langle Position : Dw\rangle$ → External_Msg_Controler .
op SchedEnterI : Control-Id Train-Id → Local_Msg_Controler .
op Move StopRaise Lowering Raising: Gate-id → Local_Msg_Gate .
endfm .

On the basis of these extensions, we introduce a new axiom, called agent splitting/recombination, that allows for composing or interacting different agents using exclusively their external messages and the associated observed attributes. Thus, as described below, this axiom allows to split (and recombine) an agent configuration in two or more subconfigurations; and this in order to apply new

[8] The *Lower* message is perceived only if the *Position* is valued to Up.

general form of rewrite rules (given hereafter) that include more than one agent components. Noting that the function $\otimes$ has been already defined as a commutative and associative for relating different agents.

vars C_1, C_2 : Conf_Agent
ax $(Agent\_Id, C_1 \ C_2) = (Agent\_Id, C_1) \otimes (Agent\_Id, C_2)$

Before we present the general form of interaction pattern between different agents, we explain the ideas under the so far introduced extension concepts and how they have to be used. For the interaction between two agents, exclusively[9] through their external messages and observed attributes, first we have to apply the state splitting/recombination axiom in order to separate the external attributes from the local ones. Second, we apply the agent splitting/recombination axiom that allows to make in evidence the different (external) agent subconfigurations (i.e. observed attributes and external messages) that have to interact with one another. Third, we have to characterize explicitly the rewrite rule that perform such interaction. The general inter-interaction pattern for such rewrite rules follows the form:

($Agent_1$-Id, External_part_of_conf) $\otimes \ldots \otimes$ *($Agent_k$-Id, External_part_of_conf)* $\Rightarrow$
($Agent_1$-Id, Modif_External_part_of_conf) $\otimes \ldots \otimes$ *($Agent_k$-Id, Modif_External_part_of_conf)*

Example 6. In our case study example, by instantiating this general form and taking into account the different form of perception explicitly defined the GRC ALBERT specification, we have the following interaction rewrite rules:
$(Train, EnterR(T,C)\langle C|Position : E\rangle) \otimes (Control, \langle C|SchdC[t] : True\rangle)$
$\Rightarrow (Train, \langle C|Pos : R\rangle) \otimes (Control, \langle C|SchdC[t] : False\rangle)$
$(Controler, Lower(C,G,\langle G|Position : Up\rangle)\langle C|Gate : Up, SoonAr : True\rangle) \otimes (Gate, \langle G|Position : Up\rangle) \Rightarrow (Controler, \langle C|Gate : Dw, SoonAr : True\rangle) \otimes (Gate, \langle G|Position : GD\rangle)$

4.3 Modeling the Action Composition

The proposed in [WK96] rewriting control consists in, first, restricting the form of rewrite rules in MAUDE to those in simple MAUDE (i.e. only *one message* must appear in the left hand side of a rule), and using only one-step rewriting without transitivity. Second, under these conditions, use the following algebra of messages that allows to perform instead of a single message rather a *composite messages* in one atomic step rewriting.

mod MSG_Algebra **is**
protecting CONFIGURATION
op $\_+\_$, $\_;\_$, $\_;;\_$ $\_\_|\_$, $\_||\_ : Msg\ Msg \rightarrow Msg$
vars $m_1, m_2, n_1, n_2 : Msg, c, d, c_1, c_2, d_1, d_2, h$: Configuration
rl $(m_1 + m_2)\ c \Rightarrow d$ if $m_1\ c \Rightarrow d \ \vee \ m_2\ c \Rightarrow d$.

[9] We are not allowed to violate the encapsulation property, so the interaction is possible only through the explicit protocol defined between the different agents.

rl $(m_1\ ;\ m_2)\ c_1\ c_2 \Rightarrow d_1\ d_2$ if $m_1\ c_1 \Rightarrow d_1\ h\ \wedge\ m_2\ c_2\ h \Rightarrow d_2$.
rl $(m_1\ ;;\ m_2)\ c_1\ c_2 \Rightarrow d_1\ d_2\ (n_1\ ;;\ n_2)$ if $m_1\ c_1 \Rightarrow d_1\ n_1\ h\ \wedge\ m_2\ c_2\ h \Rightarrow d_2\ n_2$.
rl $(m_1\ |\ m_2)\ c_1\ c_2 \Rightarrow d_1\ d_2$ if $m_1\ c_1 \Rightarrow d_1\ \wedge\ m_2\ c_2 \Rightarrow d_2$.
rl $(m_1\ ||\ m_2)\ c_1\ c_2 \Rightarrow d_1\ d_2\ (n_1\ ||\ n_2)$ if $m_1\ c_1 \Rightarrow n_1\ d_1\ \wedge\ m_2\ c_2 \Rightarrow n_2\ d_2$.
endm

As it can be easily seen in this message algebra, three message combinators have been introduced: the choice(+), the sequential(; and ;;) and the parallel (| and ||) composition. Sequential composition of two messages m_1 and m_2 imply that —as state in the condition part— they should be performed in a sequence (without interference of other rules), while in parallel composition there must be no dependence.

Translating action composition of ALBERT **language** .

Using the above presented message algebra, the action composition in ALBERT specification dealing with the same agent can be directly interpreted in the extended MAUDE (i.e. the MAUDE language with this messages control). Syntactically, a composite action can be described either as an appropriate equation or using the additional notations introduced in [WK96] that takes the form:

cntrl [composite_message_name] corresponding messages expression.

Example 7. In our GRC case study, the corresponding action (or message) composition, dealing with the same agent, can be presented as follow:

cntrl [Pass] PassR ; Exit.
cntrl [PassR] EnterR ; EnterI.
cntrl [Move] Lowering ; Raising.

4.4 On the Translation of Timing-Constraints

Real-time sensitive (OO) system can be specified in rewriting logic using either timed MAUDE based on the well founded timed rewriting logic [KW97] or in a more direct way using the MAUDE language as it has been developed in [OM96]. The basis ideas under the two approach are the same: first, we have to specify algebraically the time as Archemidian group. Second, associate with each time-dependent rewrite rule a time-stamp (as an additional label) which represent the time that have to be taken by the rule to be performed. In other words, each (time-dependent) rewrite rule of the form $\mathbf{rl} : l \Longrightarrow r$ that takes t unit of time[10] is rather represented as $\mathbf{rl} : l \stackrel{t}{\Longrightarrow} r$.

For our translation, we use the timed MAUDE; therefore, we follow the first approach but without going into detail about its foundation (the interested reader is advised to consult the following references [KW97]).

[10] Noting that time intervals are also possible [KW97].

Example 8. In the GRC problem we have two simple timed dependent actions, namely: the action $PassR$ in the Train agent that takes a period of time defined as an internal between $Epsilon_1$ and $Epsilon_2$; the action $Arrival$ in the controller takes $Epsilon_1$ unit of time. However, the two actions are defined as composite actions and then they have no direct corresponding rules. This problem is overcome by splitting the time stamp into two intervals. For example, knowing that the $PassR$ is defined as sequence of the two actions $EnterR$ and $EnterI$, their associated rules have to be stamped respectively by the interval of time $[Epsilon_1, t]$, $[t, Epsilon_2]$; where $t \in [Epsilon_1, Epsilon_2]$. These rules described above as instantaneous are now time dependent.

$$\textbf{rl}\ EnterR(T)\langle T|Position : E\rangle \overset{[Eps_1,t]}{\Longrightarrow} \langle T|Position : R\rangle$$
$$\textbf{rl}\ EnterI(T)\langle T|Position : E\rangle \overset{[t,Eps_2]}{\Longrightarrow} \langle T|Position : I\rangle$$

The same reasoning may applied to the composite action $arrival(t)$.

5 Conclusion

In this paper we proposed an adequate and a complete translation of ALBERT specifications into an extended MAUDE. This proposed extension, that is closely related to the rich concepts of the ALBERT language, it concerns, first, the introduction of a simple and powerful process language inspired by the work in [WK96]. Second, for capturing real-time sensitive behaviour we have taken profit of the timed rewriting logic and timed MAUDE language syntax [KW97]. Third, for dealing with the intra-object concurrency as well as with the rich communication concepts —like import/export messages, state perception, action and information perception— of the ALBERT language, we have introduced an appropriate axiom that allows to split and recombine the object state at a need and to make a difference between local and external features of each agent.

The benefits of this work are many-fold: First, due to this operational semantics for the ALBERT specifications, it is possible to generate rapid-prototypes using current implementation of the MAUDE language. This can be achieved specifically before assessing the crucial properties of the ALBERT specification using its property-oriented temporal logic. Second, because of the well-founded class and module inheritance of the MAUDE language, it is now no more difficult to augment the graphical as well as the textual part of the ALBERT language with notations and concepts for dealing with the two forms of inheritance. Last but not least, on the MAUDE side, it is quite possible for using the graphical notations of the ALBERT language in front of any MAUDE specification which allows to made this language more practical and easy to use for complex and distributed information systems.

In the nearest future we plan to assess the proposed translation with more complex case studies like the one proposed in [DB95] using ALBERT. On the other hand, we plan to focus on relating the semantical frameworks of the two language, namely the operational rewriting logic and the real-time property oriented OSL.

References

[DB95] P. Du Bois. *The Albert II Language: On the Design and the Use of a Formal Specification Language for Requirements Analysis.* PhD thesis, Computer Department, University of Namur, Namur(Belgique), September 1995.

[DBDZ97] P. Du Bois, E. Dubois, and J.M. Zeippen. On the Use of a Formal Requirements Engineering Language – The Generalized Railroad Crossing Problem. In *Proc. of the IEEE International Symposium on Requirements Engineering - RE'97, Annapolis MD.* IEEE Computer Society Press, 1997.

[GWM+92] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, 1992.

[KW97] P. Kosiuczenko and Wirsing. Timed Rewriting Logic with an Application to Object-Based Specification. *Science of Computer Programming*, 28:225–246, 1997.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model for concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[Mes93] J. Meseguer. A Logical Theory of Concurrent Objects and its Realization in the Maude Language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*, pages 314–390. The MIT Press, 1993.

[OM96] P. Olverczky and J. Meseguer. Specifying real-time Constraints in Rewriting Logic. In J. Meseguer, editor, *Proc. of First International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science, pages 379–402, 1996.

[SSC95] A. Sernadas, C. Sernadas, and J. F. Costa. Object Specification Logic. *Journal of Logic and Computation*, 5(5):603–630, 1995.

[WK96] M. Wirsing and A. Knapp. A Formal Approach to Object-Oriented Software Engineering. In J. Meseguer, editor, *Proc. of the First Inter. Workshop on Rewriting Logic*, volume 4. Electronic Notes in Theoretical Computer Science, 1996.

[WNL95] M. Wirsing, F. Nickel, and U. Lechner. Concurrent Object-Oriented Specification in Spectrum. In Y. Inagaki, editor, *Workshop on Algebraic and Object-Oriented Approaches to Software Science, Nagoya/Japan*, Electronic Notes in Theoretical Computer Science, pages 39–70. Nagoya University, 1995.