On the Specification and Validation of Cooperative Information Systems Using an Extended MAUDE*

Nasreddine Aoumeur** Gunter Saake
Institut für Technische Informationssysteme
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. Most of present-day information systems (IS) are characterized as cooperating, distributed (over several sites) and autonomous components; inciting to an overwhelming need for new generation of conceptual models that transcend the limited capabilities of existing ones. In this paper we argue that MAUDE language, with its rewriting logic semantical framework and under appropriate extensions, could be one of the most satisfactory alternate for specifying and validating such systems. The main extensions that proposes the paper are: first, the liberation of the object state–as tuple– from its indivisibility by introducing an adequate splitting / merging axiom allowing thereby a full exhibition of intra-object concurrency. Secondly, dictated by the need of autonomy and at the same time cooperation in such systems, a clear distinction is made between hidden and observed (by other classes) attributes as well as between local and external (i.e. import/export) messages. Thirdly, for materializing such autonomy and cooperation, we propose to split the general form of event communication for rewrite rules into two patterns: an intra-component evolution pattern enhancing intra- as well inter-object concurrency in each component—regarded as a hierarchy of classes through different forms of inheritance— and an inter-component interaction pattern promoting concurrency and keeping encapsulated all local features of interacting components. We apply these ideas to a simplified railroad crossing case study.

1 Introduction

The ever-increasing rate in size- as well as space-complexity of present-day 'information-based' organizations coupled with an overwhelming need of real-time response, make designers (users and managers) of information systems face new challenges that transcend the up-to-now existing approaches [PS98]. Indeed, size-increasing has resulted in more and more complex and multi-layered systems those components are loosely connected and behaving mostly in a true concurrent way. On the other side, space dimension is reflected by a high distribution of such components over different (geographical) sites, where different forms of (synchronous/asynchronous) communication are often required.

With the aim to coping with these challenging dimensions, and thereby reliably specifying, validating and implementing such systems, recent research advocates particularly the *formal* integration of the *object paradigm* with *concurrency* as a *minimal* requirement in conceiving advanced conceptual models [FJLS96] [ECSD98]. These research efforts have forwarded very promising object oriented (OO) foundation frameworks. For instance, OO formal foundations based on the algebraic setting in the large sense include: ADT-process OO framework [EGS92] that is more elaborated recently to DTL (i.e. distributed temporal logic) [ECSD98]; hidden sorted algebra [GD93]; and rewriting logic [Mes92] that is nowadays widely accepted as a unified model of concurrency [Mes98].

The contribution of this paper consists in pushing forward the thesis— firstly initiated in [MQ93] and more elaborated recently in [PMO98]—stipulating that MAUDE language could be one of the most promising advanced conceptual modeling. Indeed, among the benefits that make MAUDE very promising for modelling distributed information systems there are: first, its true concurrent and intrinsic rewriting logic semantics; its capability for dealing with run-time specification changing through its reflective capabilities [CM96], and last but not least its great flexibility to be extended or tailored to the application-domain at hand. Extensions include, for example, the timed rewriting

* In Proc. of the 1st OBJ/Maude/CafeOBJ Workshop, Toulouse, France, K. Fuatsuki, J. Goguen, J. Meseguer (eds.), Beta Romania Publisher, Sept. 1999.
** This work is supported by a DAAD scholarship.

logic[KW97][1], while as significant adaptation is the enrichment of (simple) Maude with message process algebras [WNL95], [WK96].

The purpose of this paper aims exactly to exploit this high (extension and/or adaptation) flexibility in order to coping with most facets in modeling information systems as distributed, autonomous but yet cooperative components. Indeed, first, such systems should exhibit high distribution that have to be reflected by intra- and well as inter-object concurrency, whereas MAUDE language allows only inter-object concurrency. Second, the cooperation of different components have to be performed w.r.t. an explicit (interface) protocol [AG97] and hence without affecting the autonomy (i.e. the hidden private part) of each component or decreasing the level of distribution and concurrency, whereas MAUDE language make no distinction between private and external features of a given component (modelled as a class).

More precisely, the main adaptations, that we argue make MAUDE more suitable for specifying and validating cooperative information systems, can be highlighted as follows:

- For allowing exhibition of intra-object concurrency, we introduce an appropriate 'splitting / merging', (of the 'usual' indivisible object state) *axiom* that takes the form $\langle I|attrs_1, attrs_2\rangle = \langle I|attrs_1\rangle\ \langle I|attrs_2\rangle$; where the empty syntax (i.e. juxtaposition) as _ _ is exactly the associative commutative (multiset union) operator on configurations. As we describe later in detail, this axiom have to be performed in harmony with the associativity and commutativity of the operator '_ _'.
- For coping with autonomy and at the same time with cooperation between different components— that we regard as a hierarchy of classes through different forms of inheritance—, the first step consists in (structurally) extending the usual notion of class to the notion of *module*; where each module is rather characterized by strictly local, encapsulated part (including structural as well as behavioural features) and an observed, external part that plays the role of explicit interface with other classes. We achieve that, on the one hand, by distinguishing between local and observed attributes— which is now quite possible due the proposed 'splitting / merging' axiom. On other hand, we separate local (i.e. internal) messages from the external (i.e. imported / exported) messages.
- For capturing behaviourally such autonomy and cooperation, we propose instead of the MAUDE's single event communication pattern rather two patterns for the form of rewrites rules: An intra-component evolution pattern that enhances intra- as well as inter-object concurrency and an inter-component interaction pattern that promotes concurrency and keeps encapsulated all local features of interacting components.

The rest of this paper is organized as follows: first section reviews the main aspects of MAUDE language. In the second section, on the one hand, we informally present the Generalized Crossing Railroad (shortly, GRC) problem. On the other hand, we present a first straightforward but 'naive' description of this problem using MAUDE language. In the main section, we present in detail using this case study the main MAUDE extensions that we propose for modeling cooperative systems. We conclude this paper by some remarks and future work.

2 Rewriting Logic and MAUDE language

Firstly introduced by J. Meseguer in [Mes90], rewriting logic is nowadays one of the widely accepted unified model of concurrency [Mes98]. 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 but rather as a change or becoming through a powerful categorical foundation [Mes92]. Second, it proposes to perform the rewriting process with maximal of concurrency on the basis of four powerful inference rules (see [Mes92] for a more detail).

In addition, one of the most advantage of rewriting logic is its straightforward and sound capability of unifying the object oriented programming paradigm and concurrency without suffering from the well-known problem of inheritance anomaly [Mes93b] [LLNW96]. This conceptualization goes 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.

[1] Although in [OM96] it has been shown that real-time aspects may be taken into account directly using rewriting logic.

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 and it is of a sort denoted by OId, C identifies the object class and is of sort CId; $atr_1, ..., atr_k$ denote the attribute identifiers of sort AId, and $val_1, ..., val_k$ denote their respective current values and are of sort $Value$. The messages (i.e. method invocation) are regarded as operations sent or received by objects, and their generic sort is denoted Msg . Object and message instances flow together in the so-called configuration, which is no more than a multiset, w.r.t. an associative commutative operator denoted by '_ _', of messages and (a set of) objects. The simplified form of such configuration[2], described as a functional module takes the following form [Mes90]:

```
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:

$$r(\bar{x}) : M_1..M_n\langle I_1 : C_1|atts_1\rangle...\langle I_m : C_m|atts_m\rangle \longrightarrow \langle I_{i_1} : C'_{i_1}|atts'_{i_1}\rangle...\langle I_{i_k} : C'_{i_m}|atts'_{i_k}\rangle\langle J_1 : D_1|atts'_1\rangle...\langle J_{i_k} : D_p|atts'_p\rangle M'_1...Ms'_q \text{ if } C$$

where r is the label, and $\bar{x}$ is the list of the variables occurring in the rule, the Ms are message expressions, $i_1, .., i_k$ are different numbers among the original $1, .., m$, and C is the rule's condition. That is, a number of objects and messages can come together and participate in a transition in which some objects may be created, other may be destroyed, and other can change their state, and where new messages may be created.

In MAUDE, subclasses may be specified using the notion of class inheritance or module inheritance. Class inheritance is semantically captured by subsorts and technically materialized by the notion of attribute variables denoted by $ATTS$[3] . Module inheritance is used for allowing (different forms of) overriding of superclass(es) methods in the definition of subclass(es).

Finally, it is worth mentioning that in MAUDE object instances are explicitly created (and deleted) with guaranteeing the uniqueness of their identities (modeled as natural counters) using an appropriate rewrite rule. However referring to the above described rewrite form, 'implicit' creation or deletion is also possible.

3 The Generalized Railroad Crossing Problem

This section is devoted, on the one hand, to an informal presentation of a simplified version of the Generalized Railroad Crossing. On the other hand, we present a first straightforward but naive translation of this informal (OO) description into MAUDE.

3.1 GRC : Informal Presentation

This case study is mainly inspired from [HL94] and [DBDZ97]; however, by restricting ourselves to the main objectives of this paper, action composition as well as real-time features will be ignored —although it is quite possible to deal with action composition as well as real-time aspects in

[2] A complete description of the notion of configuration is given in [Mes93a].

[3] We see later that the proposed extensions in this paper allow for dropping this (some bit ad-hoc) notion of attribute variables that avoid for supporting for example the encapsulation between a subclass and its superclass.

rewriting logic following respectively process algebra proposed in [WK96] and real-time rewriting logic [KW97]. Also, for the informal GRC OO description, we use some object-oriented (semi-graphical) intuitive notations [Weg90].

As sketched graphically in figure 1, three classes can be distinguished in the GCR problem: Two classes belonging to the environment, namely *Train* (describing trains traveling on controlled tracks) and *Gate* as operating crossing gates. The *Controller* class corresponds to the controlling system to be installed for operating the gates. More precisely, attributes and methods of each class are as follows:

The Train Class: Besides the self-explained attributes —namely, $TrainNo$, $SeatNb$, $CarNb$—, the relevant state component in this system is the *position* of the train. This attribute may have three values: E(lsewhere) ("not in the section of interest"), R ("in the region of interest"), or I ("in the railroad crossing"). The operations acting on this (train-) state are the following: *EnterR* that allows for changing the attribute position from E to R and at the same time informs the *Controller* class of this arrival; *EnterI* allows for entering into the railroad; and *Exit* that corresponds to leaving the road crossing. Besides that, we have included an operation *AddCar* that allows for adding new car (with a corresponding number of sites)[4].

The Gate class: This class is mainly characterized by the position of the gate that may be *Up, Down* (shortly, *Dw*). The associated messages on this state are *lower, Raise* which have to be sent by the *Controller* class.

The Controller class: This class allows for coordinating the actions and the corresponding states of the Trains and the Gate. For this, it uses as attributes: *SchedTr* as a set of already scheduled trains (with corresponding crossing times and associated gate) allowed for crossing the gate, an (boolean) attribute *SoonArrival* indicating that a train will soon arrive (in less than *Gamma* seconds), and another boolean attribute *NoSoonArrival* indicating that no train is scheduled for a sufficient time (i.e. more that *Gamma* seconds) in which case the gate should be *Up*. As messages we have included just the scheduling of a given train (i.e. its addition to the *SchedTr* list).

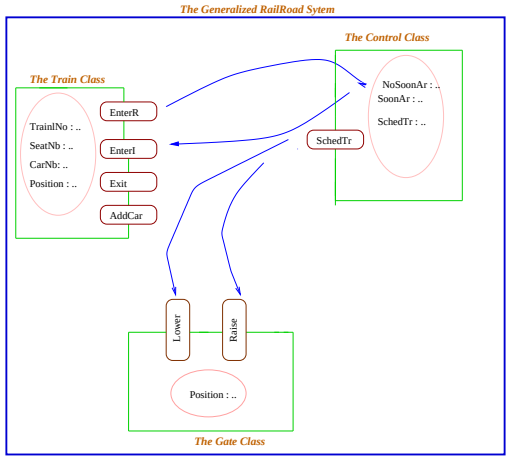

Fig. 1. The Generalized Cross Railroad classes

3.2 GRC : Maude Specification

From the informal GRC description above, the corresponding MAUDE formal specification is straightforward. First, we have to describe the functional as well as the structural aspects using the functional level of MAUDE, then the different classes with their behaviour are specified using the object modules.

[4] This operation is irrelevant for the system, but it allows us to make explicit the intra-object concurrency exhibition.

The data level. In the GRC specification, the used data types are the different values of object states and set of (scheduled) trains with their corresponding crossing time and gate. This specification can be easily given as follows.

```
fmod GRC_Sturcture is
 protecting OId Time Bool Nat .
 sorts Gate  PositionT  PositionG .
 sorts SchedSet Elt.
 subsort nil < SchedSet .
 subsort Elt < SchedSet .
 op : Up, Dw : → PositionG .
 op : Gamma : → Time .
 op : [_,_,_]: OId OId Time → Elt .
 op : _._ : SchedSet SchedSet → SchedSet [assoc. Comm. Id:nil] .
 op : _in_: Elt SchedSet → Bool .
 var S : SchedSet .
 vars E, E' : Elt .
 vars Now t : Time .
 eq E in nil = false .
 eq E in E'.S = if E == E' then true else E in S fi .
endfm.
```

Remark 1. The data type *Time* should reflect the specification of time as done for example in [KW97] (i.e. as Archemidian monoid). Each element in the Scheduled Train set denoted as $SchedSet$ is composed of the Train identity, the gate identity and the time at which it is scheduled for crossing the gate.

The object level. The Different GRC classes are directly modeled using object-oriented module. The attributes of each class are gathered into states as tuples with identity, while the behaviour of each class as well as the interaction of different classes are specified as rewrites rules.

```
omod GRC-Spec is
 protecting GRC-Structure.
 class Train | TrainNo:nat, Position : PositionT, SeatNb: nat, CarNb: nat .
 class Controler | SoonAr: Bool,  NSoonAr: Bool,  SchedTr: SchedSet .
 class Gate | Position : PositionG .
 msgs  EnterI  EnterR  Exit : OId OId → Msg .
 msg  AddCar  : OId  Nat → Msg .
 msg SchedTr : OId  Elt → Msg .
 msgs  Lower  Raise : OId → Msg .
 vars Pos : Position .
 Vars  T G C : OId .
 Vars  N N1 N2: Nat .

  ********* The GRC behaviour.
```

rl $EnterR(T,G)\langle T|Position:E, ATTS\rangle\langle C|SchedTr:S.[T,G,t], SoonAr:True, ATTS1\rangle$
$\Rightarrow \langle T|Position:R, ATTS\rangle\langle C|SchedTr:S, SoonAr:True, ATTS1\rangle EnterI(T,G).$

rl $EnterI(T,G)\langle T|Position:R, ATTS\rangle\langle G|Gate:Dw\rangle$
$\Rightarrow \langle T|Position:I\rangle\langle G|Gate:Dw\rangle Exit(T,G).$

rl $Exit(T,G)\langle T|Position:I, ATTS\rangle\langle G|Gate:Dw\rangle$
$\Rightarrow \langle T|Position:E\rangle Raise(T,G).$

rl $\langle C|SchedTr:S.[T,G,t], SoonAr:False, ATTS\rangle$
$\Rightarrow \langle C|SchedTr:S.[T,G,t], SoonAr:True\rangle Lower(G)\ if\ (Now-t) < Gamma.$

rl $\langle C|NoSoonAr:False, ATTS\rangle$
$\Rightarrow \langle C|NoSoonAr:True, ATTS\rangle Raise(G)\ if\ Not([T,G,t]inS) \wedge (t < (Now - Gamma)).$

rl $Lower(G)\langle G|Gate:Up\rangle \Rightarrow \langle G|Gate:Dw\rangle.$

rl $Raise(G)\langle G|Gate : Dw\rangle \Rightarrow \langle G|Gate : Up\rangle$.

rl $SchedTr(C,[T,G,t])\langle C|SchedTr : S, ATTS\rangle$
$\Rightarrow \langle C|SchedTr : S.[T,G,t], ATTS\rangle$.

rl $AddCar(T,N)\langle T|SeatNb : N1, CarNb : N2, ATTS\rangle$
$\Rightarrow \langle T|SeatNb : N1+N, CarNb : N2+1, ATTS\rangle \; if \; (N > 0)$.

Remark 2. In this specification, the attribute variables $ATTS$ and $ATTRS1$ stands for the unchanged attributes.The first rule say, for instance, that a given train is allowed to entering the region of interest if and only if it is already scheduled; this fact is expressed by a synchronization of Trains with the Controller. The fourth rule allows for a 'systematic' change of the variable *SoonAr* to $True$ if there is a scheduled train that should arrive in a less than $Gamma$ seconds in which case the Gate have to be lowered.

4 Extended Maude : Concepts and case study

As pointed out in the introduction and made more clear through the GRC problem, some crucial features in conceiving cooperative information systems—namely the intra-object concurrency and communication between components only through explicit interfaces—go beyond the capabilities of MAUDE. Such insufficiencies are illustrated in the GRC example by the following. First, due to the indivisibility of the object state, although acting of different attributes messages like *EnterI* (or *EnterR*) and *AddCar* cannot be performed simultaneously when they are sent to the same *Train* object. Second, there is no encapsulation (of object states) between the different classes; for instance, in the second or the third rule the *Train* state as well as the *Gate* state have to be known for performing this rule, whereas in reality *Train* class should have no knowledge about the Gate state.

By taking advantage of the flexibility of this language and particularly its inherent rewriting logic semantics, this section proposes to cope with these limitations. More precisely, first, we introduce a new axiom (in addition to the assoc. comm. of the configuration operator '_ _') that allows for *splitting / merging* the object state out of necessity, and show w.r.t. the GRC problem how it can be used for exhibiting full intra-object concurrency (without affecting the inter-object concurrency). Second, using system modules instead of (predefined) object modules and taking benefits of the proposed axiom, we propose, on the one hand, to split each object state into local and observed state parts, and split the messages into local and external (i.e. imported / exported) ones. Finally, two patterns of rewrite rules are proposed : an intra-component evolution pattern and an inter-component interaction pattern.

4.1 On the Modeling of classes as independent components

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 limitation, we propose a simple but powerful axiom that allows to split (or recombine) the different state components (i.e. part of attributes with identity of their object state) of a given object state at a need. This axiom that have to be included in the already presented configuration specification (as given below) can be described as follows; where the second axiom is just a simplified notation of an object state (after being split) without attributes.

vars $attr_1, attr_2$: `Attributes` .
ax1 $\langle Id|attr_1, attr_2\rangle = \langle Id|attr_1\rangle\langle Id|attr_2\rangle$
ax2 $\langle Id|nil\rangle = Id$

Thus, if two or more events act on the same object but on different attributes, first, we have to apply (repeatedly) this axiom to split the state of the addressed object into state components that correspond to concerned attributes, and then apply the appropriate rule. Hence, for making into practice this axiom, first, we have to remove the attribute variables denoted by $ATTSi$ from all

rules. Second, we have to mention in each rewrite rule *only* those attributes that are really involved (i.e. as conditions or change) in the rule.

The corresponding modified configuration that we describe rather as a system module—in order to interpret the two equalities as axioms (in the rewriting logic) and not as usual equations—takes the following form:

```
mod CONFIGURATION1 is
 protecting Value 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  Attributes → Object .
 op __ : Configuration  Configuration → Configuration [assoc comm id:null] .
 var Id : OId .
 vars attr1, attr2 : Attributes .
 eq ⟨Id|attr1, attr2⟩ = ⟨Id|attr1⟩⟨Id|attr2⟩ .
 eq ⟨Id|nil⟩ = Id.
endm .
```

Remark that we have dropped the class identifier from this adapted configuration. We will explain this in the next subsection.

Example 1. By removing the attribute variables from the first and the last rewrite rules of GRC problem, for instance, we obtain the following rules:

rl $EnterR(T,G)\langle T|Position : E\rangle\langle C|SchedTr : S.[T,G,t], SoonAr : True\rangle$
$\Rightarrow \langle T|Position : R\rangle\langle C|SchedTr : S, SoonAr : True\rangle EnterI(T).$

rl $AddCar(T,N)\langle T|SeatNb : N1, CarNb : N2\rangle$
$\Rightarrow \langle T|SeatNb : N1 + N, CarNb : N2 + 1\rangle \; if \; (N > 0).$

With these adapted rules and the *splitting / merging* axiom, it is obvious that we can now perform in parallel messages like $EnterR(T,G)$ and $AddCar(T,N)$ if they are sent to the *Train* state $\langle T|TrainNo : N, Position : E, SeatNb : N1, CarNb : N2\rangle$ (and the *Controller* state: $\langle C|SchedTr : S.[T,G,t], SoonAr : True, NoSoon : Var\rangle$) . For this aim, it suffices to split this state into $\langle T|Position : E\rangle$ $\langle T|SeatNb : N1, CarNb : N2\rangle$ $\langle T|TrainNo : N\rangle$(and the *Controller* state into $\langle C|SchedTr : S.[T,G,t], SoonAr : True\rangle$ $\langle C|NoSoon : Var\rangle$) and then apply the two rewrite rule in parallel.

Modeling classes as independent components. The second step towards modelling information systems as cooperative components is to avoid conceptualizing such systems as a *global* configuration. For this aim, we propose the following.

- Instead of incorporating the class identifier in each object state, we propose to take it as a common parameter. We achieve that by conceiving a class structure rather as a pair, that we referred to as component, of the form: $(Class\_Identifier, associated\_subconfiguration$[5]$)$; where the $associated\_subconfiguration$ should contain only message and object state (without class identifier) instances of the corresponding class.
- For composing and relating such different component communities, in case of more than one class, we propose to use an additional associative and commutative operator denoted as $\otimes$.

For the description of this 'components' 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 [Mes93a] section 4.2).

[5] Noting that the notion of subconfiguration was first developed in [WNL95] but without an explicit separation from the global configuration.

```
mod  COMPONENTS is
 extending  CONFIGURATION1 .
 sort  Component .
 op (_,_) :  CId  Configuration → Component .
 op _⊗_: Component  Component → Component [assoc. comm. Id:null]
 endm .
```

Example 2. Taking into account this notion of components, the resulting GRC system can be described as follows:

```
mod  GRC-Comp is
 extending  COMPONENTS .
 protecting  GRC-Structure .
 sorts  Conf_Train  Conf_Controler  Conf_Gate < configuration .
 subsorts  Train-Id  Controler-Id  Gate-Id < CId .
 subsorts Train  Controler  Gate < Component .
 subsorts Object_train  Object_gate  Object_controler < Object .
 subsorts Msg_train  Msg_gate  Msg_controler < Msg .
 subsorts Msg_Train  Object_Train < Conf_Train .
 subsorts Msg_Gate  Object_Gate < Conf_Gate .
 subsorts Msg_Controler  Object_Controler < Conf_Controler .
 ops  EnterI  EnterR  Exit : OId OId → Msg_Train .
 op  AddCar  : OId  Nat → Msg_Train .
 op SchedTr : OId  Elt → Msg_Controler .
 ops  Lower  Raise : OId → Msg_Gate .
 var Pg : PositionG .
 var Pt : PositionT .
 Vars  T G C : OId .
 Vars  N N1 N2: Nat .
 Var  B1, B2 : Bool .
 sct ⟨G|Gate : Pg⟩ : Object_Gate .          (* sct1 *)
 sct ⟨T|TrainNo : N, Position : Pt, SeatNb : N1, CarNb : N2⟩ : Object_Train .          (* sct2 *)
 sct ⟨C|SoonAr : B1, NoSoonAr : B2, SchedTr : S⟩ : 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 (internal) behaviour.
 rl AddCar(T, N)⟨T|SeatNb : N1, CarNb : N2⟩
     ⇒ ⟨T|SeatNb : N1 + N, CarNb : N2 + 1⟩ if (N > 0).
                          ********* The Gate (internal) behaviour.
 rl Lower(G)⟨G|Gate : Up⟩ ⇒ ⟨G|Gate : Dw⟩.

 rl Raise(G)⟨G|Gate : Dw⟩ ⇒ ⟨G|Gate : Up⟩.

                          ********* The Control (internal) behaviour.
 rl ⟨C|SchedTr : S.[T, G, t], SoonAr : False⟩
     ⇒ ⟨C|SchedTr : S.[T, G, t], SoonAr : True⟩Lower(G) if (Now − t)⟨Gamma.
```

Remark 3.

- With this structure of components and using the general communication pattern, it is *impossible* to capture the interaction between different components. For this reason, we have not (yet) described the component-interaction rewrite rules as done in the first attempt of translation in the precedent section. Later in this section, we show how to deal with this communication, but without violating the encapsulated aspects of each component.
- The sort constraints $sct1$ until $sct9$ ensure that elements (i.e. messages and object states) in each sub-configuration are only those declared in its corresponding class. In this sense,

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 components that all are subsorts of the *Object* sort. The sort constraints $sct4, sct5, sct6$ define how to construct (inductively) w.r.t. the configuration operator '_ _' each of the three sub-configurations. The sort constraints $sct7, sct8, sct9$ also determine precisely the sort of each component as a pair of the class-identifier and its corresponding configuration (sort). Finally, in order to avoid direct composition (through the '_ _') of sub-configurations belonging to different components, we have added the $sct10, scr11, sct12$ that exclude such composition.

The intra-component evolution pattern. As we have pointed out, the messages effect on object states into each component may be captured by the event communication pattern presented in section 2. However, because of the introduction of the splitting /merging axiom— that allows for manipulating state components instead of a whole object states— and the introduction of the notion of component—that prohibits interaction between different classes using this event pattern— we have to specialize this general communication pattern. More precisely, first, this specialized evolution pattern, that we referred to as *intra-component evolution pattern*, should involve messages and object states and /or state components which exclusively belong to a same component. Second, in order to avoid inconsistency—like deletion of just part of an object state—, we propose to exclude *implicit* object states (or state components) deletion of creation; in other word any deletion of creation should be expressed by sending a corresponding (delete or create) message. All for all, this intra-component evolution pattern can be depicted as follows:

$$r(\bar{x}) : M_1..M_n\langle I_1 : |atts_1\rangle...\langle I_m|atts_m\rangle \longrightarrow$$
$$\langle I_{i_1}|atts'_{i_1}\rangle...\langle I_{i_k}|atts'_{i_k}\rangle\langle J_1|atts'_1\rangle...\langle J_{i_k}|atts'_p\rangle\langle I_{u_1}|atts_{u_1}\rangle...\langle I_{u_s}|atts_{u_s}\rangle M'_1...Ms'_q \text{ if } C$$

where the Ms are message expressions, $\{i_1, .., i_k, u_1, .., u_s\} = \{1, .., m\}$ which reflects the fact that *no* implicit deletion or creation has performed performed; henceforth, all the state components, namely $\langle I_{u_1}|atts_{u_1}\rangle...\langle I_{u_s}|atts_{u_s}\rangle$ participating in this evolution as *indirect* conditions should remains unchanged and have not to be deleted. However, the new introduced messages $M'_1...Ms'_q$ may include delete of create messages.

Remark 4.

- It is worth mentioning that, due to the splitting / merging axiom, we should avoid deletion of *just* a part of a given object state. For this aim, we require that before performing any deletion of object of identity I[6] we should 'merge' all eventually split parts. We achieve that by applying this axiom rather as a rewrite rule: $\langle I|attrs_1\rangle\langle I|attrs_2\rangle \Rightarrow \langle I|attrs_1, attrs_2\rangle$ until finding the corresponding normal form (that always exists in this case); This ensures that whole object state will be deleted after.
- Creation and especially deletion (messages) of object states should be performed at first; this allows particularly to avoid manipulating an object state (or some of its state components) which is already deleted (i.e. there is a delete message in the corresponding sub-configuration). Simple solution to this problem[7] is to modify the condition C in the proposed intra-component pattern to $C \wedge not(delete(Id)\, in\, sub - configuration)$.

4.2 On the interaction between components

internal and external features of components. The second advantage of the proposed splitting / merging axiom is the possibility to split up different object states of a given component into a local part —and therefore *hidden* to the other components part— and *observed* part that plays the role of interface with the environment and other components. We use the notation 'observed_State' and 'hidden_State' to distinguish these two component state part.

Example 3. In the GRC case study, logically the attribute *SchedTr* containing the list of scheduled trains in the *Controller* component is the only state components that can considered as observed by the environment and the two other components. For taking this into account, the propose the conceptualization have to be extended as follows; where the non explicitly specified states as considered as hidden states (i.e. the Train state and the Gate state).

[6] Using, for instance, the rewrite rule [Mes93a]: $delete : \langle I|ATTS\rangle \Rightarrow null$.
[7] It could also be more approached using MAUDE reflection capabilities.

```
mod Obs_Hid_State_GRC is .
 extending GRC-Comp .
 subsorts observed_Controler hidden_Controler < Object_Controler .
 sct ⟨C|ShedSet : S⟩ : observed_Controler .
 sct ⟨C|SoonAr : bool, NoSoonAr : bool⟩ : hidden_Controler .
endm
```

Similarly, we can split messages in each component into internal or local messages and external as imported/exported messages, by associating with each kind a corresponding subsort.

Example 4. Taking into account the fact that the attribute $schedTr$ is the single observed state component in the whole system, we should rehabilitate the three GRC components in consequence. In the *Train* component, expect for the message $AddCar$ that have to considered as local, all the other (i.e. $EnterR, EnterI, Exit$) are considered as observed. Moreover, we propose two additional external messages namely $EnterR - Ok$ and $Stop$ that should reflect the result after sending the $EnterR$ message. In the *Controller* component, in addition to the message $SchedTr$ that is considered a local message, we introduce a new observed message denoted $SoonAr$-Ok (and $NoSoonAr$-Ok) that have to be send to the Gate component. Finally, for the *Gate* component in addition to the messages *Lower* and *Raise* that have now to considered as external, two additional observed messages denoted *Raise-Ok* and *Lower-Ok* are introduced. All for all, the modified informal graphical description is depicted in figure 2 and the MAUDE description takes the following form.

```
mod Obs_Hid_State_Message_GRC is .
 extending Obs_Hid_State_GRC.
 subsorts Local_Msg_Train  External_Msg_Train < Msg_Train .
 subsorts Local_Msg_Controller  External_Msg_Controller < Msg_Controller .
 subsorts Local_Msg_Gate  External_Msg_Gate < Msg_Gate .
 op AddCar : OId Nat → Local_Msg_Train.
 op EnterR Exit : OId OId → External_Msg_Train .
 op Enter-Ok Stop : OId OId → External_Msg_Train .
 op SchedTr : OId → Local_Msg_Controler .
 op SoonAr-Ok, NoSoonAr-Ok : OId OId → External_Msg_Controller .
 op Lower, Raise : OId OId OId → External_Msg_Gate .
 op Lower-Ok, Raise-Ok : OId OId OId→ External_Msg_Gate .
endfm .
```

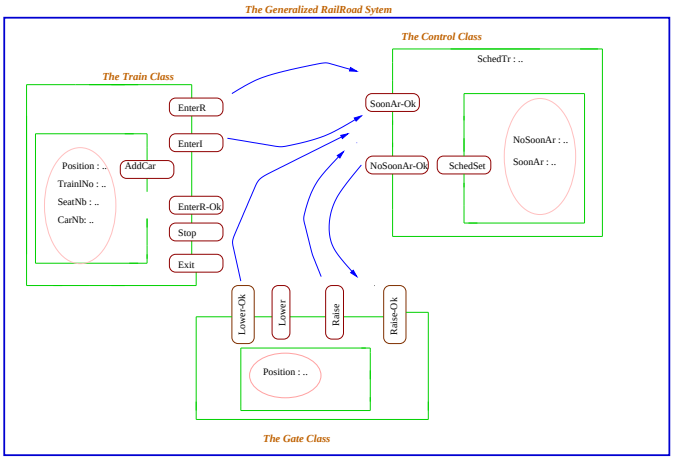

Fig. 2. The Generalized Cross Railroad classes as cooperating Components

The inter-component interaction pattern. On the basis of the introduced extensions, we first introduce a new axiom, that we refer to as *component* splitting / recombining axiom, that allows for composing different component parts using *exclusively* their external messages and associated observed attributes. Thus, as described hereafter, this axiom also allows to split (and recombine) but instead of the state rather the components into two or more subcomponents; and this in

order to apply corresponding rewrite rules which should another general form referred to as inter-component interaction pattern; such pattern may include more than one (sub-)component. Taking into account the already introduced operator $\oplus$ on components, this axiom takes the form:

vars C_1, C_2 : `Configuration` .
ax $(Component\_Id, C_1 C_2) = (Agent\_Id, C_1) \otimes (Agent\_Id, C_2)$.

Before we present the general form of interaction pattern between different components, let's shortly explain how the so far introduced 'components' ideas have to be performed. For interacting components, exclusively 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 have to apply the component splitting/recombination axiom that allows to make in evidence the different (external) subcomponents (i.e. observed attributes and external messages) that have to interact with one another. Third, we have to use the rewrite rule that perform such interaction. The general rewrite rule form that we propose for this intra-component interaction takes the form:

$r(\bar{x})$: *(Component$_1$-Id, External_part_of_conf)* $\otimes \ldots \otimes$ *(Component$_k$-Id, External_part_of_conf)* $\Rightarrow$
(Component$_1$-Id, Modif_External_part_of_conf) $\otimes \ldots \otimes$ *(Component$_k$-Id, Modif_External_part_of_conf)*

Where messages included in the *External_part_of_conf* of each component configuration should be declared as external in the associated signature, and they have to include in their arguments explicitly all the *component-Identifier* involved in the communication. On the other hand, the attributes involved in this communication should be declared as external.

Example 5. After introducing how components have to interact, we are now ready to present the complete specification of GRC problem conceived as a cooperative information systems. Indeed, in addition to the internal behaviour described in the specification *GRC-Comp* (and to slightly adapted as given below), we have to enrich this specification with the interaction between the three components. With respect to the informal description depicted in figure 2, this interaction that enters into contact only the external features of each component is expressed by the following rewrite rules:

rl $(Train, EnterR(T,G)) \otimes (Controler, \langle C | SchdSet : S \rangle)$
$\Rightarrow if\ [T,G,t]\ in\ S\ then\ (Train, null) \otimes (Controler, SoonAr(C,T,G) \langle C | SchdSet : S \rangle)$
$else\ (Train, Stop(T)) \otimes (Controler, < C | SchdSet : S \rangle)$

rl $(Controler, SoonAr - Ok(C,T,G)) \Rightarrow (Gate, Lower(C,T,G))$
rl $(Gate, Lower - Ok(G,T)) \Rightarrow (Train, EnterR - Ok(T,G))$
rl $(Train, Exit(T,G)) \Rightarrow (Gate, Raise(G,T))$

The first rewrite rule, for instance, expresses that in sending the message *EnterR(T,C)*, the train is allowed to consult the observed attribute *SchedTr* of the *Controller* component, and depending on the case that the *Train* is scheduled for crossing the Gate (at this time)— in which case there a sending of *SoonAr(C,T,G)* to the *Controler*— or not in which case it (the train) should stop there. The other rewrite rules are more straightforward.

Remark 5. Because of the introduction of the additional (external) messages in different components, some of the already (local) rewrite rules in each component have to be slightly adapted in consequence as follows.

`***** Internal Behaviour of the Train Component ********`
rl $EnterR - Ok(T,G) \langle T | Position : E \rangle \Rightarrow \langle T | Position : R \rangle EnterI(T,G)$
rl $EnterI(T,G) \langle T | Position : R \rangle \Rightarrow \langle T | Position : I \rangle Exit(T,G)$
rl $AddCar(T,N) \langle T | SeatNb : N1, CarNb : N2 \rangle$
$\Rightarrow \langle T | SeatNb : N1 + N, CarNb : N2 + 1 \rangle\ if\ (N > 0)$.

`***** Internal Behaviour of the Gate Component ********`
rl $Lower(G,T,C) \langle G | Gate : Up \rangle \Rightarrow \langle G | Gate : Dw \rangle Lower - Ok(G,T)$
rl $Raise(T,G,C) \langle G | Gate : Dw \rangle \Rightarrow \langle G | Gate : Up \rangle Raise - Ok(G,T)$

`***** Internal Behaviour of the Controller Component ********`
rl $SoonAr(T,C) \langle C | SchedTr : S.[T,G,t], SoonAr : False \rangle$

$\Rightarrow \langle C | SoonAr : True \rangle Lower(T, G)\ if\ (Now - t) < Gamma.$

rl $\langle C | NoSoonAr : False, ATTS \rangle$
$\Rightarrow \langle C | NoSoonAr : True, ATTS \rangle Raise(G)\ if\ Not([T, G, t] in S) \wedge (t < (Now - Gamma)).$

rl $SchedTr(C, [T, G, t]) \langle C | SchedTr : S, ATTS \rangle$
$\Rightarrow \langle C | SchedTr : S.[T, G, t], ATTS \rangle.$

5 Conclusion

For fulfilling most of requirements in specifying and validating cooperative information systems as distributed, autonomous but yet cooperative components, we showed in this paper that MAUDE language, under some extensions, could be one of the promising advanced conceptual modeling. The proposed extensions allow MAUDE language for dealing with intra-object concurrency (in addition to inter-object concurrency) using a very simple state splitting / merging axiom. On the other side, by pushing forward this axiom, we proposed to conceive such systems as interacting components that behave autonomously. The autonomy is captured by the introduction of an appropriate intra-component evolution pattern, while the interaction is conceived first by separating local, hidden from the outside features from observed ones, and second, by the introduction of an appropriate inter-component interaction pattern that enhances intra- as well as inter-object concurrency without violating the encapsulated, hidden features of the interacting components. The proposed extensions have been illustrated through a simplified real case study dealing with the generalized crossing railroad problem.

However, after this first step towards an adequate modeling of cooperative information systems, much work remains ahead. In particular, we plan for the nearest future the following improvements of the proposed approach. First, we have to specify the proposed extensions directly using object modules, which need to adopt appropriate 'sugar' notations and define their equivalences in system modules. Examples of that is for instance, introduction of clauses like: *local-messages*, *observed messages*, ect. Second, we plan to use the reflective capabilities of MAUDE for capturing the notion of transaction that is more than necessary for adequately capturing components interaction. Examples from the GRC problem is the sending of *EnterR* message by the *Train* component and the necessity of an immediate response from the *Controller* and the *Gate*; which means that all the associated rewrite rules should be included and thereby performed in an *atomic* transaction. Besides that, we are focusing on the verification phase, in developing such systems, mainly by adapting the work in [Den98] based on the (algebra of) proofs theory in rewriting logic.

References

[AG97] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. Technical report, School of Computer Science, Carnegie Mellon University, 1997.

[CM96] M. Clavel and J. Meseguer. Reflection and Strategies in rewriting logic. In G. Kiczales, editor, *Proc. of Reflection'96*, pages 263–288. Xerox PARC, 1996.

[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.

[Den98] G. Denker. From Rewrite Theories to Temporal Logic Theories. In H. Kirchner and C. Kirchner, editors, *Proc. of Second International Workshop on Rewriting Logic*, volume 15 of *Electronic Notes in Theoretical Computer Science*, 1998.

[ECSD98] H.-D. Ehrich, C. Caleiro, A. Sernadas, and G. Denker. Logics for Specifying Concurrent Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 6, pages 167–198. Kluwer Academic Publishers, Boston, 1998.

[EGS92] H.D. Ehrich, M Gogolla, and A. Sernadas. Objects and Their Specification. In M. Bidoit and C. Choppy, editors, *Proc. of 8th Workshop on Abstract Data Types*, volume 655 of *Lecture Notes in Computer Science*, pages 40–66. Springer-Verlag, 1992.

[FJLS96] B. Freitag, Cliff B. Jones, C. Lengauer, and H. Schek, editors. *Object Orientation with Parallelism and Persistence*. Kluwer Academic Publishers, 1996.

[GD93] J.A. Goguen and R. Diaconescu. Towards an Algebraic Semantics for the Object Paradigm. In *Proc. of 10th Workshop on Abstract Data types*, 1993.

[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.

[HL94] C.L. Heitmeyer and N. Lynch. The Generalized Railroad Crossing : A Case study in Formal Verification of real-time Systems. In *Proc. of the IEE Real-Time Systems Symposium*, 1994.

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

[LLNW96] U. Lechner, C. Langauer, F. Nickel, and M. Wirsing. (Objects + Concurrency) & Reusability – A Proposal to Circumvent the Inheritance Anomaly. In *ECOOP'96 - Object-Oriented Programming*, volume 1098 of *Lecture Notes in Computer Science*, pages 232–248. Springer Verlag, 1996.

[Mes90] J. Meseguer. Rewriting Logic as a unified Model of Concurrency. In Baeten, J. C. M. and Klop, J. W., editor, *13th Proc. of CONCUR'90*, volume 458 of *Lectures Notes in Computer Science*, pages 384–400, 1990.

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

[Mes93a] 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.

[Mes93b] J. Meseguer. Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming. In *ECOOP'93 - Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 220–246. Springer Verlag, 1993.

[Mes98] J. Meseguer. Research Directions in Rewriting Logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany*. Springer-Verlag, 1998.

[MQ93] J. Meseguer and X. Qian. A Logical Semantics for Object-orirnted Databases. In P. Bunemau and S. Jajodia, editors, *Proc. SIGMOD'93 (ACM Press)*, pages 89–98, 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.

[PMO98] I. Pita and N. Martí-Oliet. A Maude Specification of a Reflective Object-Oriented Database Model for Telecommunication Networks. In H. Kirchner and C. Kirchner, editors, *Proc. of Second International Workshop on Rewriting Logic*, volume 15 of *Electronic Notes in Theoretical Computer Science*, 1998.

[PS98] M. P. Papazoglou and G. Schlageter, editors. *Cooperative Information Systems : Trends and Directions*. Academic Press, Boston, 1998.

[Weg90] P. Wegner. Concepts and paradigms of Object-Oriented Programming. *OOPS Messenger*, 1:7–87, 1990.

[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.