Expressing GMoDS Models into Object-Oriented Models Using the Event-B Language

Marius Brezovan, Liana Stanescu and Eugen Ganea
University of Craiova, Romania
E-mail: {mbrezovan, lstanescu, eganea}@software.ucv.ro

Keywords: multi-agent systems, goal-oriented specification, object-oriented specification, Event-B

Received: July 24, 2015

Among the agent-oriented methodologies that use goals for specification of multi-agent systems, the Goal Model for Dynamic Systems (GMoDS) method allows to specify goals during requirements engineering process and then to use them throughout the system development and at runtime. Because the semantics of the GMoDS models involves the use of object-oriented concepts we choose to express a GMoDS model in an object-oriented specification. We use Event-B as a method for both specifying the GMoDS models and implementing the semantics of the runtime model of GMoDS. Because Event-B is not an object-oriented language, the goal of our research is to add support to Event-B for object-oriented modeling by using the modularization plug-in of the Rodin framework. This aim of paper is twofold: (a) to describe an object-oriented specification in Event-B, and (b) to express a GMoDS model into an object-oriented Event-B specification.

Povzetek: Razvit je agentni sistem z dodatnimi lastnostmi objektnih sistemom.

1 Introduction

In recent years the domain of *multi-agent systems* (MAS) is perceived as generating a new paradigm in order to cope with the increasing need for dynamic applications that adapt to unpredictable situations. This new software engineering domain, *agent-oriented software engineering*, provides the tools and techniques to use in designing complex, adaptive systems.

Several frameworks for multi-agent system specification have been proposed to deal with the complexity of large software systems, such as Tropos [22], Gaia [5], MaSE [11], and ROADMAP [21]. To reduce the complexity of a correct and effective design for such systems, *Organization-based Multi-Agent Systems* (OMAS) have been introduced as an effective paradigm for addressing the design challenges of large and complex MAS [18]. In OMAS there is a clear separation between agents and system, allowing a reduction in the complexity of the system. To support the design of OMAS, several methodologies have been proposed [16].

Among these proposals, the *Organization-based Multi-agent Systems* (O-MaSE) methodology [12] seems to be the only framework which integrates a set of concrete technologies aimed at facilitating industrial acceptance through situational method engineering. In O-MaSE methodology, goals are specified using *Goal Model for Dynamic Systems* (GMoDS) [13], a methodology that provides a set of models for capturing system level goals, for using them during both the design and runtime phases, in order to allow the system to adapt to dynamic problems and environments.

The development of correct/safe complex MAS is difficult with traditional software development methods. Hence, formal methods are needed in order to ensure their correctness and structure their development from specification to implementation. To that end, formalization is needed, which has begun to receive a substantial amount of interest. Several approaches for formalizing MAS development are proposed. For instance, in [19] a general framework for modelling MAS based on Object-Z and statecharts is proposed, which focuses on organizational aspects in order to represent agents and their roles. Similarly, in [24] Z notations are combined with linear temporal logic to specify the internal part of agents and the specification of the communication protocols between agents. In [8], an approach based on capturing interaction protocols between requesters, providers and middle-agents as finite state processes represented using FSP process algebra is proposed, and the resulting specifications are formally verifiable using FLTL temporal logic.

However these approaches do not address the the problem of using formal methods within a well-defined MAS development methodology. This is the reason for our attempt to use the *Event-B* both as a method to specify the O-MaSE models, and as a tool to implement the semantics and the runtime model of O-MaSE. Event-B is a state-based formal method that supports a refinement process in which an abstract model is elaborated towards an implementation in a step-wise manner. In addition Event-B is proven to be applicable in a wide range of domains, including distributed algorithms and multi-agent system development. Its deployment is supported by the *Rodin* toolset, which includes proof obligation generation and verification through a collection of mechanical provers. *Rodin* was used in several academic and complex industrial size systems.

We started our research with the study of the GMoDS methodology, an important part of O-MaSE, by translating GMoDS models in object-oriented specifications in Event-B. GMoDS represents a framework for developing complex multi-agent systems using *goals* to capture requirements, the same set of goals being used for MAS design, and at runtime. In GMoDS, goals are organized in a *goal tree* such that each goal is decomposed into a set of subgoals using *AND/OR* decomposition. Leaf goals are simple goals that must be achieved by agents. Within O-MaSE, each MAS contains a set of *roles* that it can use to achieve its goals. The roles for MAS can be derived from the goal tree, each leaf goal should have at least one role that can achieve it. For simplicity, we assume that is an one-to-one mapping between the set of goals and the set of roles. Each *agent* from a MAS is capable of playing at least one role, with the property that at every moment, an agent can have only one role. Thus, at every moment, an agent from MAS is related to a leaf goal from the goal tree of the GMoDS framework.

In GMoDS, there are two types of goals: *goal classes* and *goal instances*. Goal classes define templates from which goal instances are created. A goal class contains a set of goal attributes that are used to define the state od a goal instance. When a goal is instantiated, all its attributes must be given explicit values. While goal classes are used in the design process of MAS, the goal instances are used at runtime, or during a simulation process. Goal classes and goal instances are analogous to object classes and object instances from the object-orientation paradigm. This is the reason for using an object-oriented framework to specify the GMoDS models.

Event-B extended with several facilities, such as modularity, decomposition, the use of records and generic instantiation, shows a good potential for the use in the industrial practice. Unfortunately the Event-B language is not object-oriented because it does not support the main object-oriented concepts, such as inheritance, subtyping, class instantiation, calling of public methods of class instances, and polymorphism. Some approaches, such as records [17], modularisation [20], generic instantiation [26], and especially the *UML-B* method [27], bring closer Event-B to an object-oriented language. The UML-B graphical modelling notation provides four kind of diagrams: package, context, class and state machine diagrams. However, UML-B does not address some important object-oriented concepts, such as subtyping, polymorphism, and calling public methods of the class instances. Because GMoDS models involve the use of calling operations of some objects within the plans from the plan models, we use interfaces and modules from the modularisation plug-in of the Rodin framework, and the principles from the UML-B method for managing classes, class instances, class attributes and associations, in order to allow appropriate object-oriented specifications in Event-B. In addition we model in Event-B specifications other two main object-oriented concepts: *inheritance* and *polymorphism*. *Inheritance* is needed for creating dynamic trees of goal instances from the GMoDS runtime model, while *polymorphism* is needed for calling the appropriate operation, when a class hierarchy is used.

In conclusion, the aim of this paper is twofold: (a) to propose an extension of the Event-B method that allows the creation and destruction of class objects, as well the call of public methods of classes, inheritance, and polymorphism, as well as (b) to use this extension for translating GMoDS models into Event-B object-oriented specifications.

The rest of this paper is organized as follows. Section 2 presents the O-MaSE methodology framework, and its associated GMoDS methodology. Section 3 presents the main concepts of the Event-B method, and some of its extensions that will be used in the paper. Section 4 presents a proposal for constructing an object-oriented specification in Event-B that allows calling public methods of class instances. In Section 5 this proposal is used to express the main GMoDS models using object-oriented Event-B specifications. Finally, conclusions are given in Section 6.

2 O-MaSE and GMoDS methodologies

The Organization-Based Multiagent System Engineering [12] methodology extends the original MaSE [11] methodology to allow the design of organizational multi-agent systems. The definition of O-MaSE consists of three main components: the O-MaSE meta-model, method fragments, and guidelines.

The O-MaSE *Meta-Model* is based on an organizational approach, which extends the organization model for adaptive computational systems (OMACS) [10]. OMACS defines an organization as a set of *Goals* that the organization is attempting to accomplish, a set of *Roles* that must be played to achieve those goals, a set of *Capabilities* required to play those roles and a set of *Agents* who are assigned to roles in order to achieve organizational goals. The environment is modeled using the *Domain Model*, which defines the types of objects in the environment and the relations between them. In addition to OMACS, the O-MaSE meta-model adds new concepts, such as: *Plans* that capture algorithms that agents use to carry out specific tasks, *Actions* that allow agents to perceive or sense objects in the environment, *Organisational agents* that capture the notion of organizational hierarchy, and *Protocols* that define interactions between roles or between the organization and external actors. Figure 1 shows a simplified OMACS meta-model.

In a multi-agent organization (MAO), organizational goals are typically organized in a goal tree. In OMACS, and thus in O-MaSE, goals are specified using GMoDS

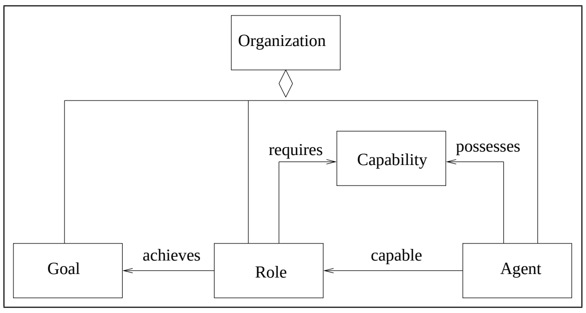

Figure 1: Simplified OMACS metamodel.

[13]. The GMoDS specification model includes the notions of goals, goal decomposition, events, precedence, and goal instantiation. The GMoDS instance model captures the dynamics of the system state while maintaining the structure of the specification model. The execution model implements these models in an efficient manner. The GMoDS specification model is used in the design process of a MAO, while the GMoDS instance and execution models are used in execution, or simulation processes of MAOs. Both in the design process, and in the execution process, the leaf goals are directly related to the agent plans. A GMoDS goal specification tree is presented in Fig. 2 (from [13]).

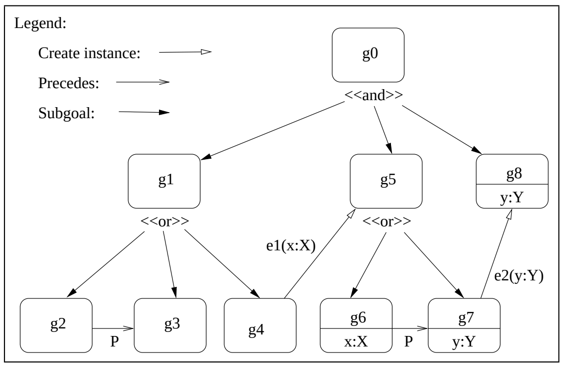

Figure 2: A GMoDS Goal specification tree.

A basic O-MaSE process is presented in Fig. 3.

A centralized *Organization-based agent architecture* is presented in Fig. 4 [12]. The *Control Component* contains *Goal Reasoning* and the *Reasoning Algorithm* that use specifications of the organisation goal, role, and agent models to perform reasoning about goals, and the assignment of agents to roles. The *Execution Component* contains the agents of the MAO specified by their roles and capabilities.

From the *Control Component*, *Goal Reasoning* is the module that implements the GMoDS framework. In this paper we describe a specification of the Goal Reasoning module using an object-oriented extension of the Event-B method.

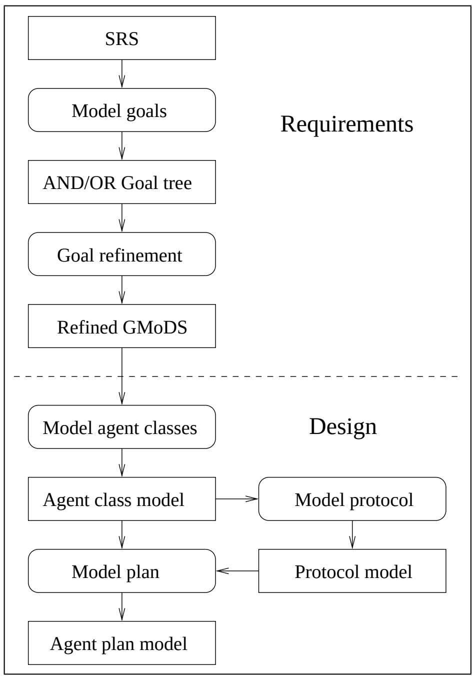

Figure 3: A basic O-MaSE Process.

3 Event-B method

Event-B [2] is a formal method for modelling concurrent systems by adopting a top-down development process. The Event-B method is influenced by the B Method [1] by using typed set theory as the mathematical language for defining state structures and events. However there is a conceptual difference between these two formal methods: while the B Method is aimed at software development, the Event-B is aimed at system development.

In order to support construction and verification of Event-B models, *RODIN*, an open toolset implemented on the top of the Eclipse platform, was constructed. The RODIN tool was initially developed as part of the European Union ICT Project RODIN (2004 to 2007) [25], and then continued by the EU ICT research projects DEPLOY (2008 to 2012) [14] and ADVANCE (2011 to 2014) [3]. The tool is implemented in Java and it uses several plug-ins that extend the basic functionality of the Event-B framework.

Event-B models are described in terms of two basic components: *contexts*, which contain the static part of a model, and *machines*, which contain the dynamic part. Contexts may contain *carrier sets*, *constants*, *axioms*, and *theorems*, where carrier sets are similar to types, while machines, which provide behavioral properties of Event-B models, may contain *variables*, *invariants*, *theorems*, and *events*. The state of a machine is described by its variables, which are constrained by invariants.

Each machine may contains a set of *events*, which describe possible state changes. Each event is a specialized B operation, and it is composed of a guard $G(t, v)$ and an action $A(t, v)$, where t represents parameters the event may contain, and v a subset of the variables of the machine. A

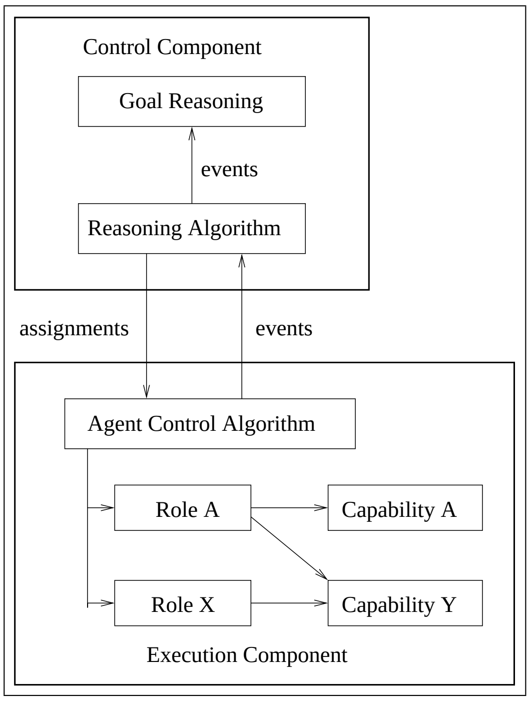

Figure 4: Organization-based agent architecture.

special event, *initialisation*, is used for describing the initial state of the machine. A machine can *see* multiple contexts. During the development, a context can *extend* one or more contexts by declaring additional carrier sets, constants, axioms or theorems.

The *refinement* is the only operation that can be applied to a machine. If a machine N refines another machine M, then M is called the *abstract machine*, while N is a *concrete machine*. Event-B uses two principal types of refinement: superposition refinement [6] and data-refinement [7]. Superposition refinement corresponds to a spatial and temporal extension of a model, while data refinement is used in order to modify the state of the machine.

The Event-B language does not allow a modular development of a system. In order to manage this development method some plug-ins have been added to the RODIN platform. In the following we shortly present the *Modularisation* plug-ins that we use for constructing our proposal. The *Modularisation* plug-in allows a modular development of a specification by defining *modules* [20], a new type of Event-B components containing groups of callable operations. A module description consists of two parts, *module interface* and *module body*. A *module interface* is a separate Event-B component that consists of a set of external module variables (v), constants (c), and sets (s), the external module invariant, and a description of module operations, specified by their pre- and post-conditions. In addition, an interface can see its context. Denoting by M a module, by MI its interface, and by MI_ctx the context of MI, the interface MI has a structure as follows:

```
**INTERFACE** $MI$
**SEES** $M\_ctx$
**VARIABLES** $v$
**INVARIANT** $M\_Inv(c, s, v)$
**INTIALISATION** $M\_Init(v)$
**OPERATIONS**
  $oper_1 \mathrel{\widehat{=}}$
    **ANY** $par_1$
    **PRE** $M\_Pre_1(c, s, par_1, v)$
    **RETURN** $res_1$
    **POST** $M\_Post_1(c, s, par_1, v, v', res_1')$
    **END**
  ...
**END**,
```

where the primed variables of the interface and the variables representing the result of the operation, specified in the predicates representing the postcondition of the operation, stand for the variable values after operation execution.

A *module body* is an Event-B machine, where the operations specified in its interface are implemented. Each operation is implemented by a *group* of events, one group for each operation. Some events from a group play a special role of operation termination events and are called *final* events. A final event returns the control to a caller.

An *operation* defined into a module M can be invoked into a Event-B machine, which can be another module, only if the module M is *imported* into this machine. The inclusion of a module into a Event-B machine is specified by a clause **USES** in the importing machine. This clause specify the interface of the imported module, and a prefix that is used to emulate a dedicated namespace for the imported module. All the names of the imported module are modified by adding this prefix.

The syntax for an operation invocation is similar to a function call. The semantics of an operation invocation is also similar to the standard semantics of a function call as in the most programming languages. Because an operation invocation is atomic, the events from the group corresponding to the operation in the module body run until termination without interference from other groups.

4 Writing object-oriented specifications in event-B

Because B and Event-B methods are not object-oriented, there are several proposals in the last years to bring object-oriented concepts into these methods. First of all, both B and Event-B have only static structuring mechanisms: they allow to define abstract machines with a static architectural structure that do not change at run-time [15]. In [4] an extension of the syntax of B is proposed for supporting the management of dynamic populations of components. In this proposal a *population manager* is associated to a machine for managing its instances. Although this extension is not object-oriented, the population manager for a machine M is a machine which represents a dynamic set of M instances, including operations for the creation and deletion of machine instances.

A similar mechanism is used also in the UML-B method: for each machine representing a class hierarchy, an implicit context is generated, which defines the set of all instances, A_SET, for each class A from the class hierarchy. As opposed to the B method, where an abstract machine can represent a class, and a hierarchy of classes is constructed by several machines that use the clause **USES** to include other the classes from the hierarchy, the Event-B method does not have **USES** and **INCLUDE** clauses, thus a hierarchy of classes must be defined in a single machine. Another weakness of the UML-B method is the absence of the method calls of the class instances, because an Event-B machine has only events (or transitions in UML-B), not operations as in the case of the B abstract machines.

The aim of this Section is to bring some object-oriented concepts into Event-B modelling, without changing the syntax of Event-B, and thus allowing the Event-B specifications to be realized and verified with the *Rodin* tool. We do not use the UML-B method because of the weakness above mentioned, related to the absence of the method calls. In fact, we use the modularization approach [20] in order to allow this action, while preserving some object-oriented elements from the UML-B, such as management of class instances, attributes, associations, and inheritance.

We use *interfaces* for describing class hierarchies and the methods (operations) of the classes, and *modules* for implementing the class methods. For a hierarchy H containing the classes, $A_1, \ldots, A_k$, the following Event-B components are defined:

– A context, H_Ctx, which contain: the set $INST$ of all instances of all class from the H hierarchy, the constant $Void$ representing the $null$ instance, and the set of all instances of the classes $A_1\_Inst, \ldots, A_k\_Inst$ respectively, with the property that:

$$INST = \bigcup_{i=1}^{k} A_i\_Inst \cup \{Void\},$$
$$A_i\_Inst \cap A_j\_Inst, \forall i \neq j.$$

– An interface, H_Intf, having:
  – as variables, the sets $A_i \in \mathbb{P}(A_i\_Inst)$ representing the set of active objects of the class A_i, $i = 1, \ldots, k$, and relations and functions representing attributes of these classes and the associations between some classes,
  – as operations, the constructor and the destructor for each class, and other operations representing the methods of the classes A_i, $i = 1, \ldots, k$

– A module, H_Impl, where the operations defined in H_Intf are implemented.

As an example, we consider two classes, $Node$ and $List$, where each list is an ordered sequence of nodes, and each node has as attribute with an integer value. The context related to classes $Node$ and $List$ is defined as follows:

```
CONTEXT $H\_Ctx$
SETS $INST$
CONSTANTS $Void,\ Node\_Inst,\ List\_Inst$
AXIOMS
  $Void \in INST$
  $Node\_Inst \subseteq INST$
  $List\_Inst \subseteq INST$
  $\text{partition}(INST, \{Void\}, Node\_Inst, List\_Inst)$
END
```

From the interface H_Intf, the variables, their invariants and initializations are defined as follows:

```
INTERFACE $H\_Intf$
SEES $H\_Ctx$
VARIABLES $Node,\ value,\ List,\ first,\ next$
INVARIANTS
  $Node \in \mathbb{P}(Node\_Inst \cup \{Void\})$
  $value \in Node\_Inst \to \mathbb{N}$
  $List \in \mathbb{P}(List\_Inst \cup \{Void\})$
  $first \in List\_Inst \to Node\_Inst \cup \{Void\}$
  $next \in List\_Inst \to (Node\_Inst \to (Node\_Inst \cup \{Void\}))$
INTIALISATION
  $Node := \varnothing,\ value := \varnothing$
  $List := \varnothing,\ first := \varnothing,\ next := \varnothing$
OPERATIONS
  ...
```

From the operations related to the $Node$ class we present only the constructor $newNode$, and the function $getValue$:

```
$newNode \;\widehat{=}$
  ANY $self,\ v$
  PRE
    $self \in Node\_Inst \setminus Node$
    $v \in \mathbb{N}$
  RETURN $ret$
  POST
    $Node' = Node \cup \{self\}$
    $value' = value \cup \{self \mapsto v\}$
    $ret' = self$
  END
$getValue \;\widehat{=}$
  ANY $self$
  PRE $self \in Node$
  RETURN $ret$
  POST $ret' = value(self)$
  END
```

From the operations related to the $List$ class we present only the the destructor $deleteList$ and the operation

$insertFront$:

```
$deleteList \mathrel{\widehat{=}}$
  ANY $self$
  PRE $self \in List$
  RETURN $ret$
  POST
    $first' = (\mathrm{dom}(first) \setminus \{self\}) \lhd first$
    $next'(self) = \varnothing$
    $List' = List \cup \{self\}$
    $\forall a, b \cdot a \in Node\_Inst \ \wedge a \in Node\_Inst \ \wedge$
        $a \mapsto b \in next(self) \ \Rightarrow a \notin Node'$
    $ret' = Void$
  END
$insertFront \mathrel{\widehat{=}}$
  ANY $self,\ n,\ v$
  PRE
    $self \in List$
    $n \in Node\_Inst \setminus Node$
    $v \in \mathbb{N}$
  RETURN $ret$
  POST
    $value'(n) = v$
    $next'(self) = next(self) \cup \{n \mapsto first(self)\}$
    $first'(self) = n$
    $ret' = first'(self)$
  END
```

In the implementation module, H_Impl, each operation is implemented by a group containing one or more events. Other defined operations can be called in the action part of these events. For example, in the implementation of operation $insertFront$, the constructor $newNode$ of the class $Node$ is called:

```
$insertFront \mathrel{\widehat{=}}$
  ANY $self, n, v$
  WHERE
    $self \in List$
    $n \in Node\_Inst \setminus Node$
    $v \in \mathbb{N}$
  THEN
    $n := newNode(v)$
    $next(self) := next(self) \cup \{n \mapsto first(self)\}$
    $first(self) := n$
    $insertFront\_ret := n$
  END
```

From all operations of the class $List$, only the operation $deleteList$ has a group containing two events: $deleteListNonEmpty$, that occurs for each node in a non-empty list, and $deleteListEmpty$ that occurs when the list is empty.

In order to describe the modeling of the inheritance and polymorphism concepts in Event-B, we use an example of a class hierarchy with three classes, B, $D1$ and $D2$, as in Fig. 5, where $D1$ and $D2$ inherit the class B. In addition, all three classes have the same operation, *op*.

Denoting with $INST$ the set of all instances of the classes form the above hierarchy, with B_Inst, B_Inst, $D1\_Inst$, and $D2\_Inst$ the set of all possible instances of the classes B, $D1$, and $D2$ respectively, the fact that $D1$ and $D2$ *inherit* the class B can be described as follows:

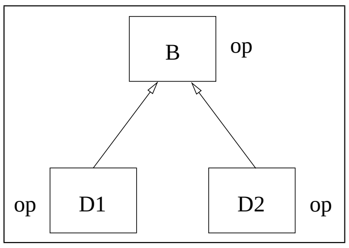

Figure 5: A class hierarchy with one class root.

```
CONTEXT $Ctx$
SETS $INST$
CONSTANTS $Void,\ B\_Inst,\ D1\_Inst,\ D2\_Inst$
AXIOMS
  $Void \in INST$
  $\mathrm{partition}(INST, \{Void\},\ B\_Inst,\ D1\_Inst,\ D2\_Inst)$
  $\mathrm{partition}(B\_Inst, D1\_Inst,\ D2\_Inst)$
END
```

The *polymorphism*, related to the operation *op* in this case, is modeled in the interface, I, which has only one operation, denoted by *op* in this case, and in its associated module, M, which contains a group with two different final events, denoted by $op1$ and $op2$ in this case, one event for each each operation from a inherited class.

```
INTERFACE $I$
SEES $Ctx$
VARIABLES $B,\ D1,\ D2$
INVARIANTS
  $B \in \mathbb{P}(B\_Inst \cup \{Void\})$
  $D1 \in \mathbb{P}(D1\_Inst \cup \{Void\})$
  $D2 \in \mathbb{P}(D2\_Inst \cup \{Void\})$
INTIALISATION
  $B := \varnothing,\ D1 := \varnothing,\ D2 := \varnothing$
OPERATIONS
$op \mathrel{\widehat{=}}$
  ANY $self$
  PRE $self \in B\_Inst \setminus B$
  RETURN $ret$
  POST
    $B' = B \cup \{self\}$
    $self \in D1\_Inst \setminus D1 \ \Rightarrow \ D1' = D1 \cup \{self\}$
    $self \in D2\_Inst \setminus D2 \ \Rightarrow \ D2' = D2 \cup \{self\}$
    $ret' = self$
  END
```

The module M can be described as follows:

```
MACHINE $M$
  IMPLEMENTS $I$
  SEES $Ctx$
  ...
  GROUP $op$ BEGIN
    FINAL $op1$ $\widehat{=}$
      ANY $self$
      WHERE
        $self \in D1\_Inst \setminus D1$
      THEN
        $D1 := D1 \cup \{self\}$
        $op1\_ret := self$
      END
    FINAL $op2$ $\widehat{=}$
      ANY $self$
      WHERE
        $self \in D2\_Inst \setminus D2$
      THEN
        $D2 := D2 \cup \{self\}$
        $op2\_ret := self$
      END
  END
END
```

5 Expressing GMoDS models in object-oriented specifications in event-B

As stated in Section 2, we describe a specification of the GMoDS framework using an object-oriented extension of the Event-B method, which represents the *Goal Reasoning* module from the *Control Component* of an *Organization-based agent architecture*.

The GMoDS definition contains three different models [13]: (i) a *Specification model* that contains a tree structure of *goal classes* and their associations, and (ii) a *Runtime model* that contains a tree structure of *goal instances* and the actions that are executed, each action being related to a association between classes, and (iii) an *Execution model* that implements GMoDS using and updating continuously a collection of sets of goal instances, according to the current state of each goal instance.

5.1 GMoDS Models

The *Specification model* of GMoDS contains the *goal specification tree*, G_{Spec}, which describes how the goal classes are related to one another, and where upper level goals (parents) are decomposed into lower level sub-goals (children) and each parent has either a conjunctive or disjunctive achievement condition as shown via the $\langle\langle and\rangle\rangle$ and $\langle\langle or\rangle\rangle$ decoration in Fig. 2. Goals without children are known as *leaf goals*.

In addition to goals, the specification model uses another concepts, such as, *relations*, *events*, and *parameters*. The main relation type used by this model is the *goal precedence*, specified by the $precedes$ relationship, that ensures that no agents work on a specific goal until all goals that precede that goal have been achieved. In Fig. 2 there are two $precedes$ relations: $precedes(g2, g3)$, and $precedes(g6, g7)$. Events in GMoDS are represented by *triggers*:

- a *positive trigger*, or simply $trigger$, which allows a new goal instance of a certain class g_j to be created when and event e_k occurs during the pursuit of a goal instance of a class g_i, eventually by passing some parameter values p. In Fig. 2 there are two triggers: $trigger(g1, e1, x) = \{g5\}$, and $trigger(g7, e2, y) = \{g8\}$.
- a *negative trigger*, or $\neg trigger$, which allows an active goal instance of a certain class g_i to eliminate another active goal instance of a certain class g_j from the set of *active goal instances* when an event e_k occurs.

There is always an *initial trigger*, usually denoted by e_0, that is used when the system starts, which creates an instance of the root goal (and, recursively, it can create others goal instances).

The *Runtime model* is represented by a dynamic tree of goal instances, G_{Inst} that retains the structure of G_{Spec} while allowing dynamism by way of triggering and precedence. For each goal instance from G_{Inst}, four predicates are dynamically set:

- $achieved$, which determines whether a goal has been achieved by the system. For *leaf goals* $achieved$ becomes true when the agent pursing the goal notifies the system of its achievement, while for parent goals, the value of the $achieved$ is based on the achievement condition (conjunction or disjunction) and the state of its children.
- $obviated$, which states whether a goal is no longer needed by the system. A goal becomes obviated if it is a child of a disjunctive goal that has been achieved that does not precede any other system goal.
- $preceded$, which becomes $true$ if a goal preceding it has not been achieved, or if a new goal may still be instantiated that may precede it.
- $failed$, which becomes $true$ if the system has deemed that the goal can never be achieved by the system.

For example, after the *initial trigger*, the instance tree, G_{Inst}, has a structure as presented in Fig. 6. Instances of the goals $g5$ (and subsequently $g6$ and $g7$) and $g8$ are not created because they will be created (triggered) by $g4$ and $g7$ when the events $e1$ and $e2$ respectively will occur.

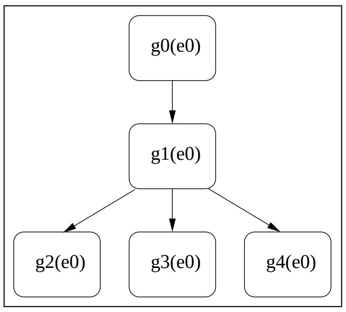

Figure 6: The tree G_{Inst} associated to the tree G_{Spec} from Fig. 2 after the initial trigger.

In the *Runtime model* there are maintained and updated six sets, $G_{I-Triggered}$, $G_{I-Active}$, $G_{I-Achieved}$, $G_{I-Removed}$, $G_{I-Failed}$ and $G_{I-Obviated}$ as shown in Fig. 7.

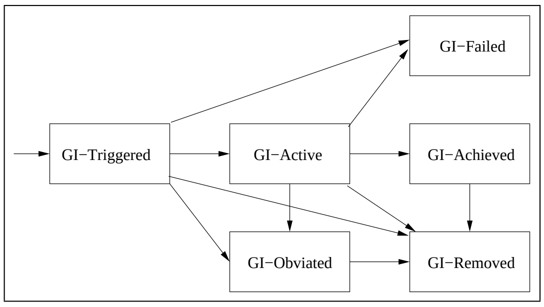

Figure 7: Goal execution model.

Each set contains current instance goals having the same state:

- $triggered$, for all instances created by a trigger event, or, recursively, by a parent goal,
- $active$, for all triggered instances that are not preceded,
- $obviated$, that is based on the $obviated$ predicate,
- $achieved$, that is based on the $achieved$ predicate,
- $failed$, that is based on the $failed$ predicate,
- $removed$, for all goal instances destroyed by a negative trigger.

When the state of a goal instance is one of the last three state, this goals remains in this state until the system stops.

5.2 Expressing GMoDS Models into an Object-Oriented Model

For specifying in Event-B the GMoDS framework (in fact the Goal Reasoning module), all the three GMoDS models must be specified. I3n the case of the Specification model, the goal tree G_{Spec} is defined by using goal classes as nodes. All classes from a goal tree will form a hierarchy having an abstract class, denoted by $Goal$ as the root of this hierarchy. For the goal tree from the Fig. 2, the goal class hierarchy is presented in Fig. 8, where the classes $g0$, $g1$, . . ., $g8$ inherit the class $Goal$.

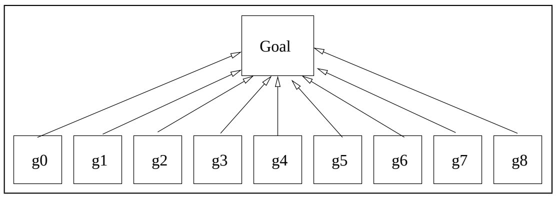

Figure 8: Goal class hierarchy.

The main two attributes of the class $Goal$ are $goal\_state \in Goal\_STATES$ and $goal\_type \in Gol\_TYPE$, where:

$$Goal\_STATES = \{triggered, active, achieved, failed, obviated, removed\},$$
$$Goal\_TYPE = \{AND, OR, LEAF\}.$$

In order to allow the specification of:

- the trigger events from the specification model,
- the predicates from the runtime model,
- the sets of goal instances, from the implementation model,

the following associations between goal classes are used:

- $down$ and $right$, which allows to specify the goal tree from G_{Spec},
- $creates$, $created$ and $destroy$, which allow to specify the positive and negative triggers,
- $precedes$ and $preceded$, which allow to specify the precedence relation between goals,
- up, which allows to retrieve the parent of a goal.

These associations are presented in Fig. 9.

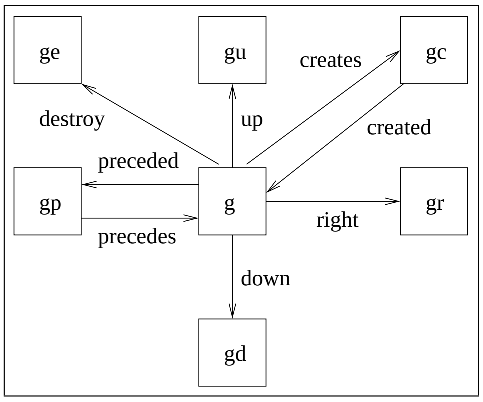

Figure 9: Goal class associations.

For the specification the GMoDS runtime model, a tree of goal instances must be specified. Because the type of goal instances does not need to be specified, nor the associations $precedes$, $creates$ and $destroy$, in this case only the tree structure of the instances is specified. Unfortunately the associations up, $down$, and $right$ from G_{Spec} can not be used, because the tree structure of goal instances, $G_{Instances}$ is not always identical with the tree structure of G_{Spec}: a positive trigger event can create multiple instances of the same goal that are "sibling" nodes (having the same parent). For solving this problem we use different associations related to the sets of goal instances: $upInst$, $downInst$, and $rightInst$. The class used to specify the runtime model is $Tree$, a tree of goal instances, which is related to the set $G\_Instance$ from the runtime model.

There is no need to implement the six sets of goal instances from the implementation model, as presented in Fig. 7, because the current state of each goal instance specifies exactly the set to that instance belongs to.

For translating the GMoDS framework into an object-oriented model, we use the following classes:

- The class *G_Spec*, which is the main class of the translated model, because it contains both the static tree of goal classes, and the dynamic tree of the goal instances.
- The class $GName$, whose elements represent the nodes of the static static tree of goal classes.
- The class *Tree*, which represents the dynamic tree of the goal instances.
- The class $Goal$, whose elements represent the nodes of the dynamic tree of the goal instances.
- The class Env, that implements the rest of the Organization-based agent architecture: the Reasoning algorithm, and the Execution component.

In fact, $GName$ is not really a class, because it does not have constructors and destructors (the tree of the goal classes from *G_Spec* is static). We use instead the notion of *Records* for $GName$, an extension of the Event-B method. The main components of $GName$ are the following:

- $state$, which represents the current state of the corresponding goal class, from the set *Goal_*$STATES$.
- *curr_inst*, which represents the set of active goal instances of the corresponding goal class.
- up, $down$, $right$, $precedes$, and $preceded$ that represent the relations between goal classes, as presented in Fig. 9.

The class $Goal$ represents the goal class hierarchy, as presented in Fig. 8. It contains only the attributes $upInst$, $downInst$, and $rightInst$, representing the relations between goal instances in a dynamic tree structure. the only operations allowed by $Goal$ are the constructor $newGoal$ and the destructor $delGoal$.

The singleton class *Tree* contains only one attribute, *rootInst*, which represents the root of the dynamic tree of goal instances. *Tree* is a singleton class because there is a single object of *Tree*, which is an attribute of *G_Spec*. In addition, *Tree* has three operations:

- *deleteInst*, which deletes all the sub-tree having as parameter its root.
- *addChildInst*, which adds a new created instance as the first child of the parent specified as parameter.
- $addBrotherInst$, which adds a new created instance as the right of the goal instance specified as parameter.

The elements of *Tree* are instances of the class $Goal$. There is only one instance of the class $Tree$, which is a member of the class *G_Spec*.

The singleton class *G_Spec* contains only two attributes:

- *rootG*, an element of the $GName$ set, representing the root of the static tree of goal classes (e.g. $g0$ in our example).
- tr, the unique instance of the class *Tree*, representing the dynamic tree of goal instances.

In addition, *G_Spec* has several operations, according to the relations between the classes *G_Spec* and *Env*, as presented in Fig. 10:

- $start$, representing the event that starts the execution (or simulation) process of the MAO, and thus the Goal reasoning algorithm.
- *achivedInstGoal*, which informs *G_Spec* that a goal instance have been achieved.
- *createInstGoal*, which informs *G_Spec* that an instance of a goal class must be created.
- *deleteInstGoal*, which informs *G_Spec* that a goal instance must be deleted.
- *failedInstGoal*, which informs *G_Spec* that an active goal has failed.
- *createdInstGoal*, which informs Env that a goal instance has been created.
- *deletedInstGoal*, which informs Env that a goal instance has been deleted.

The unique instance of the class *G_Spec*, gsp, represents the *Goal reasoning* module, a part of the *Control component*, from the *Organization-based agent architecture*.

Env represents the environment for the GMoDS framework that:

- Contains the *Reasoning algorithm* from the *Organization-based agent architecture* that performs the reorganisation structure of a MAO, based of information received from the Goal reasoning algorithm (e.g. from the GMoDS framework).
- Contains the *Execution components* from the *Organization-based agent architecture*, which contains the agents that achieve the roles related to the instances of the leaf goals in the goal tree, and send messages to those instances, when a goal has been achieved, or when it failed,
- Can send a message to the GMoDS system to start its execution (i.e. it sends the initial trigger to the parent goal of the goal hierarchy).

The relations between the classes Env and the *G_Spec* are presented in Fig. 10, where:

- The relation $start$ exists between Env and the root of the goal hierarchy (e.g. $g0$ in our example),
- The relations $achieved$ and $failed$ exist between Env and the leaves of the goal hierarchy (e.g. $g2$, $g3$, $g4$, $g6$, and $g7$ in our example),
- The relation $create$ exists between Env and some nodes from the goal hierarchy having a positive trigger (e.g. $g5$ and $g8$ in our example),
- The relation $delete$ exists between Env and some nodes from the goal hierarchy having a negative trigger.
- Relations $created$ and $deleted$ exist between the goal classes from *G_Spec* and the Env, indicating to the Reasoning algorithm that some goal instances have been created, or deleted.

All these relation represent in fact operations of the class Env. Excepting the operations $start$, $achieved$, $failed$, $create$ and $delete$, the rest of the class Env is not specified in this paper. This will be the subject of a future research.

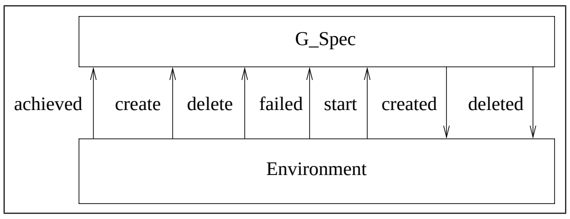

Figure 10: Environment and Goal classes associations.

5.3 An Example of Expressing GMoDS Models in Event-B

Using the patterns specified in Subsection 5.2 we can express the GMoDS system from Fig. 2 into an object-oriented model in Event-B. In fact, the model specified in Event-B encapsulates the GMoDS framework representing the Goal reasoning module into an object, gsp, instance of the class G_Spec, while the rest of the Organization-based agent architecture is represented by the object env, instance of the class Env.

The context of the modeled system will contain the sets and the constants that follows the object-oriented patterns specified in Subsection 5.2.

CONTEXT $OBAA\_Ctx$
SETS
$INST,\ GName,\ Goal\_STATES,\ Goal\_TYPE$
CONSTANTS
$Void,\ Goal\_Inst,\ G\_Spec\_Inst,\ Tree\_Inst,\ Env\_Inst$
$g0\_Inst,\ g1\_Inst,\ g2\_Inst,\ \ldots,\ g8\_Inst$
$gsp,\ tr, env$
$g0,\ g1,\ g2,\ \ldots,\ g8$
$triggered,\ active,\ achieved,\ failed,\ obviated,$
$removed,\ inactive$
$AND,\ OR,\ LEAF,\ NONE$
AXIOMS
$Void \in INST$
$\text{partition}(INST, \{Void\}, Goal\_Inst, Tree\_Inst,$
$Env\_Inst, G\_Spec\_Inst)$
$\text{partition}(Goal\_Inst,\ g0\_Inst,\ \ldots,\ g8\_Inst)$
$\text{partition}(G\_Spec\_Inst,\ \{gsp\})$
$\text{partition}(Tree\_Inst,\ \{tr\})$
$\text{partition}(Env\_Inst,\ \{env\})$
$\text{partition}(GName,\ \{g0\},\ \{g1\},\ \ldots,\ \{g8\})$
$\text{partition}(Goal\_TYPE, \{AND\}, \{OR\}, \{LEAF\})$
$\text{partition}(Goal\_STATES, \{triggered\}, \{active\},$
$\{achieved\}, \{failed\}, \{obviated\}, \{removed\})$
END

In the interface $OBAA\_Intf$, the variables and their invariants allow to specify the main object-oriented concepts, as defined in the Subsection 5.2:

INTERFACE $OBAA\_Intf$
SEES $OBAA\_Ctx$
VARIABLES
$Env,\ G\_Spec,\ Tree,\ Goal$
$type,\ state,\ curr\_inst$
$creates,\ destroy,\ up,\ down,\ right,\ precedes,\ preceded$
$rootG,\ rootInst,\ treeInst$
$lastInst,\ lastGoalChild$
$goalName$
INVARIANTS
$Env \in \mathbb{P}(Env\_Inst \cup \{Void\})$
$G\_Spec \in \mathbb{P}(G\_Spec\_Inst \cup \{Void\})$
$Tree \in \mathbb{P}(Tree\_Inst \cup \{Void\})$
$Goal \in \mathbb{P}(Goal\_Inst \cup \{Void\})$
$rootG \in G\_Spec \to GName \cup \{Void\}$
$treeInst \in G\_Spec \to Tree \cup \{Void\}$
$rootInst \in Tree \to Goal\_Inst \cup \{Void\}$
$lastInst \in GName \to Goal\_Inst \cup \{Void\}$
$goalName \in Goal\_Inst \to GName$
$type \in GName \to Goal\_TYPE$
$state \in Goal\_Inst \to Goal\_STATES$
$curr\_inst \in GName \to \mathbb{P}(Goal\_Inst \cup \{Void\})$
$available\_inst \in GName \to \mathbb{P}(Goal\_Inst \cup \{Void\})$
$up, down, right \in GName \to GName \cup \{Void\}$
$creates, created \in GName \to GName \cup \{Void\}$
$precedes, preceded \in GName \to GName \cup \{Void\}$
$upInst, downInst, rightInst \in Goal\_Inst \ \to\ Goal\_Inst$
$\cup \{Void\}$
INTIALISATION
$\ldots$

The *initialization* event means in fact the creation of the static tree structure of G_{Spec}, as defined in Fig. 2, and some other initializations, such as (a) initialization of singleton classes, (b) defining the goal types, (c) managing the goal instances, (d) specifying the hierarchical structure of the goal tree, (e) specifying the positive and negative triggers, and (f) specifying the precedence relations between goals:

$Env := \{env\},\ G\_Spec := \{gsp\},\ Tree := \{tr\}$
$rootG(gsp) := g0,\ treeInst(gsp) := tr$
$rootInst(tr) := Void$
$\ldots$
$type(g0) := AND,\ type(g1) := OR$
$\ldots$
$curr\_inst(g0) := \varnothing,\ curr\_inst(g1) := \varnothing,$
$\ldots$
$available\_inst(g0) := g0\_Inst,\ available\_inst(g0) := g0\_Inst,$
$\ldots$
$lastInst(g0) := Void,\ lastInst(g1) := Void,$
$\ldots$
$up(g0) := Void,\ up(g1) := g0,$
$\ldots$
$down(g0) := q1,\ down(g1) := g2,\ down(g5) := g6,$
$\ldots$
$right(g1) := g5,\ right(g5) := g8,$
$\ldots$
$creates(g4) := g5,\ created(g5) := g5,$
$\ldots$
$precedes(g2) := g3,\ preceded(g3) := g3,$

In the following we present only the operations related to the operation $create$ of the environment. The other operations are similar. [1]

[1]The entire Event-B model is available at `http://software.ucv.ro/~mbrezovan/fm/gmods_model.zip`

Because some of the operations, such as $createGoalInstance$, of the class *G_Spec* creates recursively all the nodes that from a sub-tree having a root a goal instance, for correctly specifying the post-condition of this operation we need to define the transitive closure of the relation $down$. The same operation is needed for $createGoalInstance$, when the transitive closure of the relation up is needed. Unfortunately, the Event-B language has a strict mathematical language, which is based on a set-theoretic model and corresponding proofs for modeling and refinement consistencies, and on the First Order Predicate Calculus for decomposition. For extending this mathematical language, the Theory plug-in was implemented, which is a Rodin extension that provides the facility to define mathematical extensions as well as prover extensions. There three kinds of extension, one of them is related to extensions of set-theoretic expressions or predicates. One example extensions of this kind consist of adding the transitive closure of relations or various ordered relations.

Butler [9] proposes propose a special case of an operator defined as the solution of some predicate, namely a fixed-point definition. For example, transitive closure of a relation R may be defined as follows [9]:

```
**operator** $tcl$
  **prefix**
  **args** $r$
  **type parameters** $T$
  **condition** $down \in T \leftrightarrow T$
  **fixpoint** $y$ **where**
    $r \cup r\,;y$
    **order**$\{a \mapsto b \mid a \in T \leftrightarrow \leftrightarrow \wedge a \subseteq b\}$
**end**
```

We can define the transitive closure of the relation $down$ (as well as for the relation up) following the above example:

```
**operator** $tcl$
  **prefix**
  **args** $down$
  **type parameters** $GName$
  **condition** $r \in GName \leftrightarrow GName$
  **fixpoint** $y$ **where**
    $down \cup down\,;y$
    **order**$\{a \mapsto b \mid a \in GName \leftrightarrow GName \wedge a \subseteq b\}$
**end**
```

The transitive closure of the relation $down$ allow us to determine all the pairs $(g1 \mapsto g2)$ such that $up(g1) = g2$.

The operation $newGoal$ of the class $Goal$ class can be defined as follows:

```
$newGoal \mathrel{\widehat{=}}$
  **ANY** $self, g$
  **PRE**
    $self \in Goal\_Inst$
    $g \in Goal\_Inst \setminus Goal$
  **RETURN** $ret$
  **POST**
    $Goal' = Goal \cup \{g\}$
    $ret' = g$
  **END**
```

For allowing the polymorphism in this case, in the implementation module there will be eight final events related to the group $newGoal$: $newGoal\_g1$, $newGoal\_g2$, ..., $newGoal\_g8$.

The operation $addGoalInst$ of the class $Tree$ will add a single goal instance to the tree.

```
$addGoalInst \mathrel{\widehat{=}}$
  **ANY** $self, ge, gi$
  **PRE**
    $self \in Tree\_Inst$
    $ge, gi \in Goal\_inst$
  **RETURN** $ret$
  **POST**
    $ge = Void \Rightarrow rootInst'(self) = gi$
    $ge \neq Void \wedge downInst(ge) = Void \Rightarrow downInst'(ge) = gi$
    $ge \neq Void \wedge downInst(ge) \neq Void \Rightarrow lastInst'(ge) = gi$
    $ret' = self$
  **END**
```

In the implementation module there are three events in the group $addGoalInst$: $addRootInst$, $addChildInst$ and $addBrotherInst$, corresponding to the three above cases.

The main operations of create and destroy a goal instance of the interface *OBAA_Intf* are related to the class *G_Spec*, which contains the tree of goal instances as attribute. The creation of an instance of a parent goals recursively creates children to the corresponding subtree that are non-triggered subgoals.

The operation $createGoalInstance$ of the *G_Spec* class can be defined as follows:

```
$createGoalInstance \mathrel{\widehat{=}}$
  **ANY** $self, gc, gi$
  **PRE**
    $self \in G\_Spec$
    $gi \in available\_inst(gc) \setminus curr\_inst(gc)$
    $gc \in GName$
    $created(gc) \neq Void$
  **RETURN** $ret$
  **POST**
    $curr\_inst'(gc) = curr\_inst(gc) \cup \{gi\}$
    $preceded(gc) = Void \Rightarrow state'(gi) = active$
    $preceded(gc) \neq Void \Rightarrow state'(gi) = triggered$
    ...
```

The following two predicates specify the recursive creation of the sub-tree having gi as root for non-preceded goals:

```
$\forall gc \mapsto g \in tcl(down) \wedge created(g) \neq Void \wedge$
  $\exists i \in available\_inst(g) \setminus curr\_inst(g) \wedge$
  $preceded(g) = Void \wedge down(up(g)) = g$
  $\Rightarrow curr\_inst'(g) = curr\_inst(g) \cup \{i\} \wedge$
  $state'(i) = active \wedge$
  $downInst'(lastInst(up(g))) = i$
$\forall gc \mapsto g \in tcl(down) \wedge created(g) \neq Void \wedge$
  $\exists i \in available\_inst(g) \setminus curr\_inst(g) \wedge$
  $preceded(g) = Void \wedge down(up(g)) \neq g \wedge$
  $\exists gl \in GName \wedge right(gl) = g$
  $\Rightarrow curr\_inst'(g) = curr\_inst(g) \cup \{i\} \wedge$
  $state'(i) = active \wedge$
  $downInst'(lastInst(right(gl))) = i$
```

The case for preceded goals is similar to the non-preceded case. The last predicates specify the linking of the sub-tree root, gi, in the tree $treeInst$ of G_Spec:

```
    $rootInst(tr) = Void \Rightarrow rootInst'(tr) = gi$
    $rootInst(tr) \neq Void \wedge down(up(gc)) = gc \wedge$
      $\mathrm{card}(curr\_inst(gc)) = 1 \Rightarrow down(lastInst(up(gc))) = gi$
    $rootInst(tr) \neq Void \wedge down(up(gc)) = gc \wedge$
      $\mathrm{card}(curr\_inst(gc)) > 1 \Rightarrow right(lastInst(gc) = gi$
    $rootInst(tr) \neq Void \wedge down(up(gc)) \neq gc \wedge$
      $\exists gl \in GName \wedge right(gl) = gc \wedge$
      $\mathrm{card}(curr\_inst(gc)) = 1 \Rightarrow right(lastInst(gl))) = gi$
    $rootInst(tr) \neq Void \wedge down(up(gc)) \neq gc \wedge$
      $\exists gl \in GName \wedge right(lastInst(gl)) = gc \wedge$
      $\mathrm{card}(curr\_inst(gc)) > 1 \Rightarrow right(lastInst(gc)) = gi$
    $ret' = gi$
  END
```

When implementing this operation in the implementation module, $OBAA\_Impl$, the function $createGoalInstance$ can be recursively applied, because the two associations, $down$ and $right$ can be viewed as the two links, $left$ and $right$ of a binary tree. There are four events in the group $createGoalInstance$ in the implementation module, two related to the leaf nodes, and two related to the non-leaf nodes:

– $createGoalInstanceLeafNotPrededed$,
– $createGoalInstanceLeafPrededed$,
– $createGoalInstanceNotPreceded$,
– $createGoalInstancePreceded$.

From the implementation module, $OBAA\_Impl$, we present only a single event for each described above operation.

For the operation $newGoal$ of the class hierarchy Goa we present the event $newGoal\_g1$:

```
$newGoal\_g1 \mathrel{\widehat{=}}$
  ANY $self, g$
  WHERE
    $self \in G\_Spec$
    $g \in g0\_Inst \setminus curr\_inst(g0)$
  THEN
    $curr\_inst(g0) := curr\_inst(g0) \cup \{g\}$
    $newGoal\_g1\_ret := g$
  END
```

For the operation $addGoalInst$ of the class $Tree$ we present the event $addChildInst$:

```
$addChildInst \mathrel{\widehat{=}}$
  ANY $self, ge, gi$
  WHERE
    $self \in Tree\_Inst$
    $ge, gi \in Goal\_inst$
    $ge \neq Void$
  THEN
    $downInst(ge) = gi$
    $addChildInst\_ret := gi$
  END
```

When implementing the operation $addGoalInstance$ of the class G_Spec in the implementation module, the function $createGoalInstance$ can be recursively applied, because the two associations, $down$ and $right$ can be viewed as the two links, $left$ and $right$ of a binary tree. We present the event $acreateGoalInstanceNotPreceded$.

```
$createGoalInstanceNotPreceded \mathrel{\widehat{=}}$
  ANY $self, gn, gi$
  WHERE
    $self \in G\_Spec$
    $gn \in GName$
    $gi \in available\_inst(gn)$
    $created(gn) = Void$
    $down(gn) \neq Void$
    $right(gn) \neq Void$
    $preceded(gn) = Void$
  THEN
    $curr\_inst(gn) = gi$
    $state(gi) := active$
    $addChild(treeInst(self), gi,$
        $createGoalInstance(down(gn)))$
    $addBrother(treeInst(self), gi,$
        $createGoalInstance(right(gn)))$
    $createGoalInstanceNotPreceded\_ret := gi$
  END
```

Finally, the environment class, Env, has five operations that simply call the operations of the class G_Spec: $start$, $create$, $delete$, $achieved$, and $failed$. We present the operation $create$:

```
$start \mathrel{\widehat{=}}$
  ANY $self$
  WHERE
    $self \in Env$
  THEN
    $createGoalInstance(gsp, g0)$
    $start\_ret := Void$
  END
```

We uses the *Pro-B* plug-in [23] for the *Rodin* platform [25] to verify the consistency of the modeled system. *ProB* is an animator and model checker for Event-B. It allows animation of Event-B specifications, and it can be used for model-checking, and for evaluating a variety of provers or tactics on a selection of proof obligations.

6 Conclusions

In this paper we presented an initial research related to express Organisation-based multi-agent software engineering (O-MaSE) to an object-oriented model in Event-B. We started to study the *Goal Model for Dynamic Systems* (GMoDS), a methodology that defines the operational semantics of a dynamically changing model of system goals, which has been used as the requirements modeling for the O-MaSE methodology.

Because the object-oriented model translated from the GMoDS models use some object-oriented concepts, such as inheritance, and calling of class methods, we used the modularisation plug-in of Rodin for implementing these concepts. We presented some pattern to translate GMoDS models to an object-oriented specification in Event-B, and we have illustrated these patterns for implementing an example from [13].

We planned to accomplish this work by testing the proposed patterns on real multi-agent systems, and to extend the research to the O-MaSE framework.

References

[1] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] Jean-Raymond Abrial. *Modeling in Event-B. System and Software Engineering*. Cambridge University Press, 2010.

[3] ADVANCE. `http://www.advance-ict.eu/`.

[4] Nazareno Aguirre, Juan Bicarregui, Theo Dimitrakos, and Tom Maibaum. Towards Dynamic Population Management of Abstract Machines in the B Method. In *ZB 2003: Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 528–545. Springer-Verlag, 2003.

[5] Michael Wooldridge ans Nicholas R. Jennings and David Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[6] R.-J. Back and J. von Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In J.W. deBakker, W.-P. deRoever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer, 1990.

[7] Ralph-Johan Back. Refinement Calculus, Part II: Parallel and Reactive Programs. In J.W. deBakker, W.-P. deRoever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer, 1990.

[8] Amelia Badica and Costin Badica. FSP and FLTL framework for specification and verification of middle-agents. *International Journal of Applied Mathematics and Computer Science*, 21(1):9—25, 2011.

[9] Michael Butler and Issam Maamria. Mathematical Extension in Event-B Through the Rodin Theory Component. Technical Report 251, Electronics and Computer Science, University of Southampton, 2010.

[10] S.A. DeLoach, W. Oyenan, and E.T. Matson. A Capabilities Based Model for Artificial Organizations. *J. of Autonomous Agents and Multiagent Systems*, 16(1):13—56, 2008.

[11] S.A. DeLoach, M.F. Wood, and C.H. Sparkman. Multiagent Systems Engineering. *Intl. J. of Software Engineering and Knowledge Engineering*, 11(3):231–258, 2001.

[12] Scott A. DeLoach and Juan Carlos Garcia-Ojeda. O-mase: A Customizable Approach to Designing and Building Complex, Adaptive Multiagent Systems. *Intl. J. of Agent-Oriented Software Engineering*, 4(3):244–280, 2010.

[13] Scott A. DeLoach and Matthew Miller. A Goal Model for Adaptive Complex Systems. *Intl. J. of Computational Intelligence: Theory and Practice*, 5(2), 2010.

[14] DEPLOY. `http://www.deploy-project.eu/`.

[15] T. Dimitrakos, J. Bicarregui, B. Matthews, and T. Maibaum. Compositional Structuring in the B-Method: A Logical Viewpoint of the Static Context. In *ZB 2000: Formal Specification and Development in Z and B*, volume 1878 of *Lecture Notes in Computer Science*, pages 107–126. Springer-Verlag, 2000.

[16] A. Estefania, J. Vicente, and B. Vicente. Multi-Agent System Development Based on Organizations. *Electronic Notes in Theoretical Computer Science*, 150(3):55—71, 2006.

[17] Neil Evans and Michael Butler. A Proposal for Records in Event-B. In *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2006.

[18] J. Ferber, O. Gutknecht, and F. Michel. From Agents to Organizations: An organizational View of Multi-Agent Systems. In P. Giorgini, J.P. Müller, and J.J. Odell, editors, *Agent-Oriented Software Engineering*, volume 2935 of *Lecture Notes in Computer Science*, pages 214—230. Springer, 2004.

[19] V. Hilaire, P. Gruer, A. Koukam, and O. Simonin. Formal Specification Approach of Role Dynamics in Agent Organisations: Application to the Satisfaction-Altruism Model. *Intl. J. of Software Engineering and Knowledge Engineering*, 16(3), 2007.

[20] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala. Supporting Reuse in Event-B Development: Modularisation Approach. In *Abstract State Machines, Alloy, B, and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 2010.

[21] T. Juan, A. Pearce, and L. Sterling. ROADMAP: Extending the Gaia Methodology for Complex Open Systems. In *Proc. 1st Intl. Joint Conf. on Autonomous Agents And Multiagent Systems*, pages 3–10, 2002.

[22] J. Mylopoulos, J. Castro, and M. Kolp. Tropos: A Framework for Requirements-Driven Software Development. In J. Brinkkemper and A. Solvberg, editors, *Information Systems Engineering: State of the Art and Research Themes*, pages 261–273. Springer-Verlag, 2000.

[23] ProB. `http://www.stups.uni-duesseldorf.de/ProB/index.php5/ProB_for_Rodin`.

[24] A. Regayeg, A. H. Kacem, and M. Jmaiel. Specification and Verification of Multi-Agent Applications Using Temporal Z. In *Proc. Intl. Conf. on Intelligent Agent Technology*, pages 260—266, 2004.

[25] RODIN. `http://www.event-b.org/`.

[26] Renato Silva and Michael Butler. Supporting Reuse of Event-B Developments through Generic Instantiation. In *Formal Methods and Software Engineering*, volume 5885 of *Lecture Notes in Computer Science*, pages 466–484. Springer, 2009.

[27] Colin Snook and Michael Butler. UML-B and Event-B: An Integration of Languages and Tools. In *Proc. IASTED Intl. Conference on Software Engineering*, pages 336–341, 2008.