Bridging KAOS and Event B

Xavier Devroey

Facultés Universitaires Notre-Dame de la Paix
2009 - 2010

Contents

Chapter 1

Bridging KAOS and Event B: existing approaches

This chapter presents three existing methods to derive Event-B model from a KAOS model. The first one, proposed by Matoussi works on a KAOS goal diagram, build with "Immediate Achieve" goals, built with milestone-driven and or-refinement patterns. The second approach, proposed by Aziz *et al.* add the notion of trigger conditions for events to derive an Event-B model from a KAOS model. The last approach, proposed by De Landtsheer takes linear temporal logic formula expressed exclusively with past operator on input and produce a event-based security policy expressed in Polpa. A syntactic change can translate this policy to Event-B.

1.1 Expressing KAOS Goal Models with Event-B: A. Matoussi

Matoussi describes in [Matoussi, 2009, Gervais et al., 2009, Matoussi et al., 2008] a process to transform a KAOS goal model into an Event-B specification. This process takes on input a KAOS goal model that is not operationalized and produces an Event-B model corresponding to a specification that satisfies the requirements described in the input model.

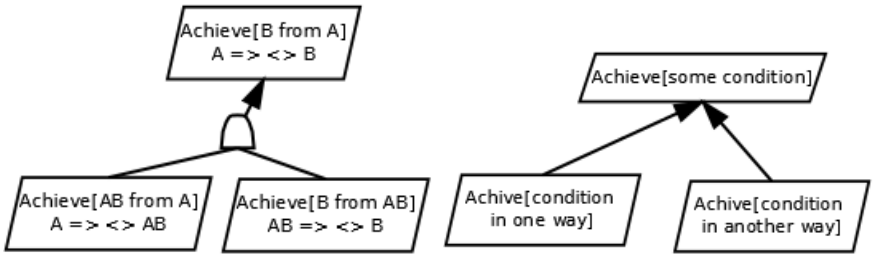

Figure 1.1: Milestone-driven refinement and Or-refinement

This process is based on refinement patterns. Each refinement pattern used in the KAOS model will correspond to a refinement step in the Event-B model. Actually the process works with functional "Immediate Achieve" goals which are the most commonly used goal type. Those goals have to be formally defined with an assertion of the form $A \Rightarrow \Diamond B$, which says that from a state where A is true, another state where B is true can be reached someday. The supported patterns are the milestone-driven refinement pattern, used when a target condition B can be reached from a current condition A with an intermediate condition AB and the or-refinement pattern, used when a goal can be satisfied in different ways.

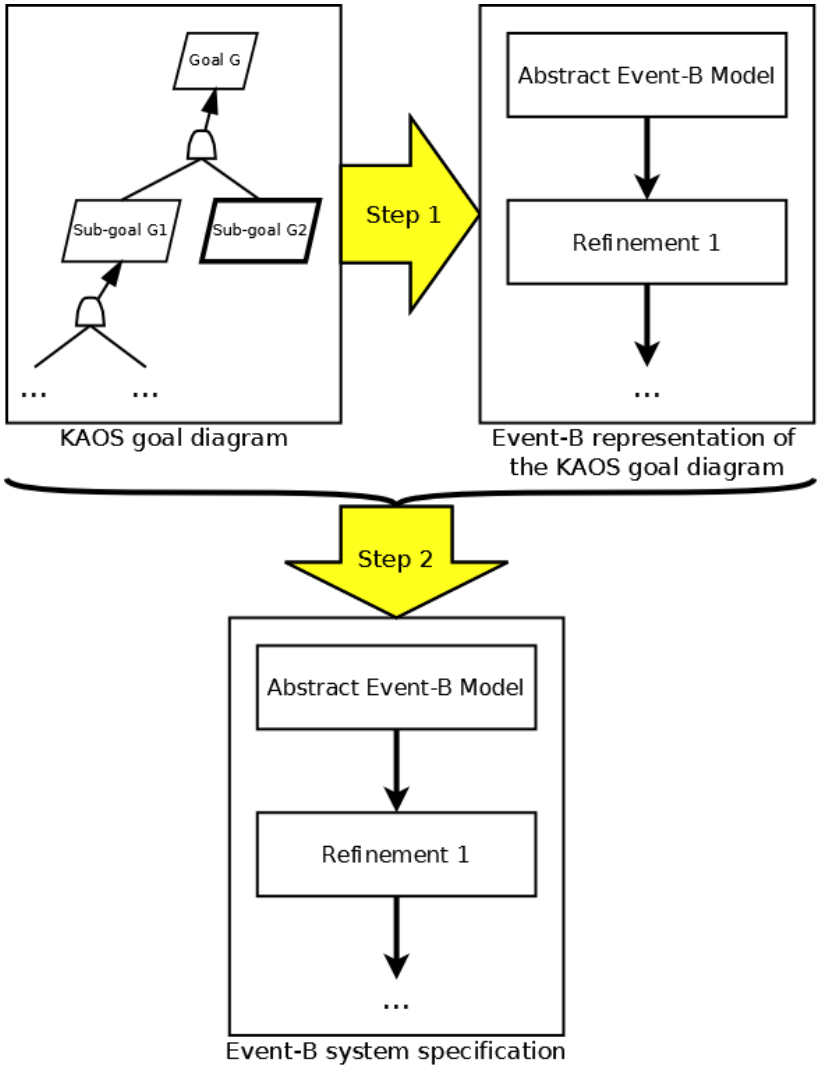

Figure 1.2: Expressing KAOS Goal Models with Event-B: process overview

The process in figure 1.2 has two phases: the first one creates an Event-

B representation of the goal model. The initial Event-B model includes the definition of a context with all the types used for data and the definition of an initial machine. This initial machine represents the root goal of the KAOS model and each refinement in this model has to follow one of the two patterns described here above. Each refinement step in the goal model will correspond to a refinement step of the Event-B machine, so we have a chain of refined machines where each machine will correspond to a "stage" in the goal model.

The second phase formally derives an Event-B specification that satisfies the requirements expressed in the goal model. To do this, it takes on input the goal model and the Event-B representation of this model created in the first phase. This second phase correspond to the operationalization process that can be performed in KAOS and guaranty that operations preserve all the properties of the goal model. As in the first phase, the initial Event-B model will be defined for the root goal of the model and each refinement in the goal model following one of the two patterns will correspond to a refinement in the Event-B model.

1.1.1 First phase

Formally speaking, a KAOS goal is seen as a property that the system has to establish:

$$\text{Achieve[G]}$$
$$A \Rightarrow \Diamond B$$

This property will be represented as an event in the Event-B model where the premise of the implication is transcribed in the initialization event of the machine and the consequence of the implication is transcribed in the then part of the event `EvtG` associated to the goal. An execution of this event means that the goal G has been satisfied. The guard of `EvtG` is set to true to express the fact that at this level the goal can always be achieved.

Listing 1.1: KAOS expressed in Event-B: initial machine

MACHINE EventBGoalModel_level_0
SEES ModelContext
VARIABLES
    Manipulated data
INVARIANTS
    `inv :` Data types definitions
EVENTS
Initialisation
    begin
        `act :` A
    end
Event *EvtG* $\widehat{=}$

```
    where
        grd : TRUE
    then
        act : B
    end
END
```

Milestone-driven refinement

When we have a milestone-driven refinement, it means that the parent goal is satisfied when all the sub-goals have been satisfied. The `EvtG` event of the parent machine is refined into a new event `EvtG` taking as pre-condition the conjunction of the functional post-conditions of the children. The refinement of goal G following the pattern described in figure 1.1 will give a machine:

Listing 1.2: KAOS expressed in Event-B: milestone refinement machine

```
MACHINE EventBGoalModel_level_1
REFINES EventBGoalModel_level_0
SEES ModelContext
VARIABLES
    Manipulated data
INVARIANTS
    inv : Data types definitions
EVENTS
Initialisation
    begin
        act : A ∧ AB
    end
Event EvtG1 ≙
    where
        grd : TRUE
    then
        act : AB
    end
Event EvtG2 ≙
    where
        grd : TRUE
    then
        act : B
    end
Event EvtG ≙
refines EvtG
    where
        grd : AB ∧ B
    then
        act : B
    end
END
```

Or-refinement

When we have an or-refinement, it means that the parent goal is satisfied when one or more of the sub-goals have been satisfied. The `EvtG` event of the parent machine is refined into a new event `EvtG'` taking as pre-condition a formula expressing that one or more of the two sub-goals have been satisfied. It does not seem to be a generic approach here and the knowledge and competence of the analyst will play an important role. For instance in the case described by Matoussi *et al.* in [Gervais et al., 2009], the guard of a refined `EvtG'` event uses the union of two sets, one for each of the sub-goals and compare it to the set of all the elements:

$$\ldots \wedge LocalisedElements = (LocalisedByGPSElements \cup LocalisedByWIFIElements) \wedge \ldots$$

1.1.2 Second phase

In the second phase, functional and non-functional goals are treated the same way. The main idea here is to say that an operation can be executed while the associated goal has not been satisfied (considering the non-functional properties too), which is the same as while it's post-condition has not been verified. However, this is not sufficient to ensure that an "Achieve" goal has been reached. A new event called "closing" is added with a guard equals to the post-condition (without the non-functional properties) of the goal to reach. So for the initial machine corresponding to the root goal G we will have an event `EvtOpG` that can be executed while G has not been reached and an event `Closing` that can be executed when G is satisfied. This `Closing` event will finalize the system. As in the first phase, the machine will be refined following the refinement pattern used in the goal model and each level in the goal model will correspond to a machine in the Event-B model.

Note that in their example, Matoussi *et al.* in [Gervais et al., 2009] are working with sets and express the negation of the initial goal post-condition with universal quantifiers. The initial machine for goal G will be:

Listing 1.3: Operationalization Event-B: initial machine

```
MACHINE EventBOperationalSpecification_level_0
SEES ModelContext
VARIABLES
    Manipulated data
INVARIANTS
    inv : Data types definitions
EVENTS
Initialisation
    begin
        act : A
    end
```

Event *EvtOpG* $\widehat{=}$
  where
    grd : $\neg B$
  then
    act : Do something that makes things going further
  end
Event *Closing* $\widehat{=}$
  where
    grd : B without non-functional properties
  then
    act : $Exit := OK$
  end
END

As in the first phase, the initial model will be refined according to the refinement patterns used in the goal model. The Closing event is taken as it and the sub-goals will be translated to events like in the machine here over.

Milestone-driven refinement

When a parent goal G is refined into sub-goals $G_1, ..., G_n$ according to the milestone-driven refinement pattern, it means that the goal G can be decomposed into n steps and that G is satisfied if the final step G_n is reached. The sub-machine will thus have `EvtOpG1,...,EvtOpGn` declared events where the pre-condition is the negation of the post-condition of the corresponding `EvtGi` event in the Event-B model of phase one and the action is something that makes things going further to the step G_{i+1}. The realization of the last sub-goal G_n implies the realization of the patent goal G, so the last event `EvtOpGn` will refine the `EvtOpG` event of the parent machine. The refinement of goal G following the pattern described in figure 1.1 will give a machine:

Listing 1.4: Operationalization Event-B: initial machine

MACHINE EventBOperationalSpecification_level_1
REFINES EventBOperationalSpecification_level_0
SEES ModelContext
VARIABLES
  Manipulated data
INVARIANTS
  inv : Data types definitions
EVENTS
Initialisation
  begin
    act : A
  end
Event *EvtOpG1* $\widehat{=}$
  where
    grd : $\neg AB$
  then
    act : Do something that makes things going further
  end
Event *EvtOpG2* $\widehat{=}$
refines *EvtOpG*

where
grd : $\neg B$
then
act : Do something that makes things going further
end
Event *Closing* $\widehat{=}$
refines *Closing*
where
grd : B without non-functional properties
then
act : $Exit := OK$
end
END

Or-refinement

As for phase one, when we have an or-refinement, it means that the parent goal is satisfied when one or more of the sub-goals have been satisfied. The `EvtOpG` event of the parent machine is refined into a new event `EvtOpG`' taking as pre-condition the negation of the corresponding event in the Event-B model of phase one, possibly simplified and where possible ambiguities have been removed.

The two sub-goals are handled as in the general case by having a pre-condition equals to the negation of the post condition of the corresponding event in the model coming from phase one.

1.2 From Goal-Oriented Requirements to Event-B Specification: B. Aziz *et al.*

1.2.1 Notion of triggered event

1.2.2 Operationalisation patterns

Table 1.1 presents the operationalisation patterns for the three most used goals types. A and B in the KAOS requirement's formal definition represents first-order logical formulae defined over objects of the KAOS model. Those objects are translated into variables in the Event-B model and thus A' represent the equivalent to A formula defined over those variables and B' represent the generalised substitution derived from predicate B, which will be seen as the post-condition of the substitution.

Table 1.1: Patterns for Operationalising Requirements into Event-B [Aziz et al., 2009]

Requirements	**Formal Definition**	**Event-B Operationalisation**
Immediate Achieve	$A \Rightarrow \circ B$	EVENT e WHEN A' NEXT B' END
Bounded Achieve	$A \Rightarrow \Diamond_{\leqslant d} B$	EVENT e WHEN A' WITHIN d NEXT B' END
Unbounded Achieve	$A \Rightarrow \Diamond B$	EVENT e WHEN A' EVENTUALLY B' END

1.3 Deriving Event-based Security Policy from Declarative Security Requirements: R. De Landtsheer

Chapter 2

Bridging KAOS and Event B: proposed approach

2.1 Overview of the approach

Chapter 1 presents existing techniques to translate a goal requirement model to an Event-B model. Some of them, like those proposed by Matoussi or Aziz *et al.* can be directly used with KAOS. Others like the procedure described by De Landtsheer can be used to derive Event-B model from first order temporal logic formula.

The problem with all those methods is that they are limited to a subset of KAOS elements. Matoussi's approach [Matoussi et al., 2008,Matoussi et al., 2009, Matoussi, 2009, Gervais et al., 2009] is limited to one goal type, the unbounded Achieve goals which correspond to the formal definition pattern $A \Rightarrow \Diamond B$ and to two refinement patterns, the milestone refinement and the or-refinement. The method proposed by Aziz *et al.* [Aziz et al., 2009] uses the notion of trigger, which is not standard in Event-B, to translate the three more used goal patterns, the immediate Achieve, the unbounded Achieve and the bounded Achieve into triggered events. Although more patterns can be discovered, the notion of trigger hide quite a complex mechanism of event scheduling that can quickly introduce misinterpretation errors. De Landtsheer's procedure [Landtsheer, 2007a] was created to work with Polpa policy language, a language to express acceptable sequences of events. As underlined by De Landtsheer the notions of events, conditions and actions present in Polpa are similar to those present in Event-B with a syntactic translation. The limitation is that the procedure works exclusively with the since $A \ \boldsymbol{S} \ B$, the always been $\blacksquare A$ and the once $\blacklozenge A$ past operators.

Another difficulty with all those methods is that they are monolithic in the sense that they are executed once from the requirement to the Event-B model and a change in the first need the replay of the method to regenerate the second. It is also not possible to go on the other way, modify the Event-B

model and have the modifications reflected in the KAOS model.

To answer these two problems, we propose here a semi-formal method to build a bridge between the KAOS model and the Event-B model. Starting from the requirements expressed in a KAOS model, we will build step by step an Event-B model where each element will be justified by a requirement. This justification will be implemented through traceability links between the two models and a set of rules that have to be respected to keep the links between the models consistent. The KAOS model may be incomplete and enriched later, even if the elaboration of the Event-B model has started. Contrary to the methods here over, the construction process may be iterative and the analyst can travel between the two models as long as the traceability rules are respected.

Figure 2.1 presents an overview of the process. Starting from the KAOS object model, an initial machine and context are created to represents the data and very general update events to represent the fact that those data evolve in time. Those elements are then reused to dispatch the update events between the different machines, where each machine correspond to an agent, according to the control links defined in the agent model. The requirements and expectations under the responsibility of the agents express the effective update of the controlled data. The responsibility model is thus used for the refinement of update events. As an element of the object model can be controlled by one and only one agent, the update event corresponding to this element will be refined in one and only one machine. In one agent's machine, the update of the elements that are not controlled by the agent will be represented by the very general update events defined in the initial machine. The elaboration of concrete machines for each agent can be parallelized and may be recomposed after to get a general model.

The first step is presented in section 2.2. The second step is described in section 2.3. Section 2.4 presents the traceability links between the KAOS model and the Event-B model with a list of criteria to keep the links between the two models consistent. In section 2.5, some examples describe what happen if one model is modified.

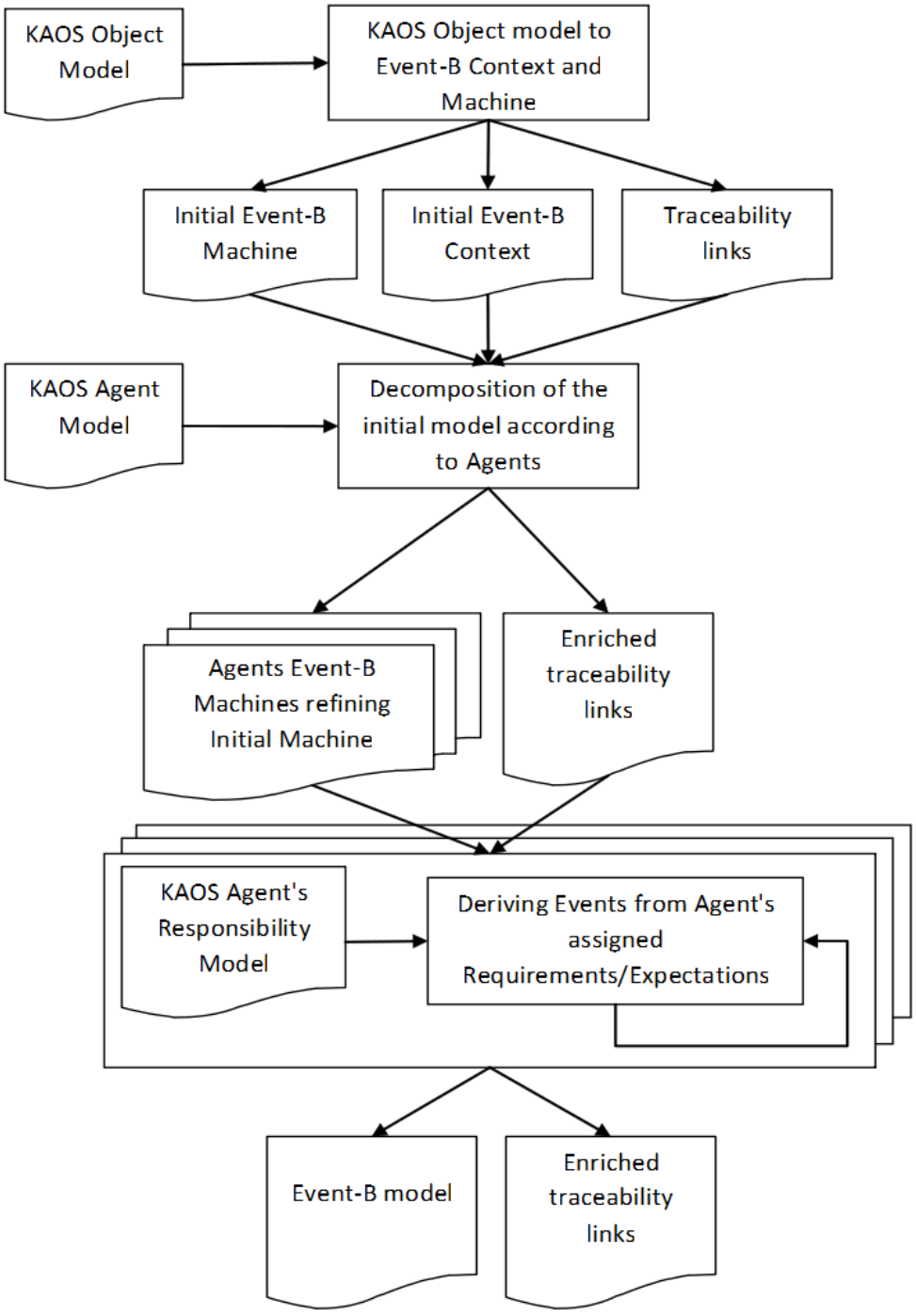

Figure 2.1: Proposed method overview

2.2 KAOS Object model to Event-B Context and Machine

In KAOS, every concept used in a definition in the goal model has to be defined in the object model. It means that when the goal model is complete, all predicates used in the formal definition of goals and in particular requirements have been defined in the object model [van Lamsweerde, 2009, Landtsheer, 2007b]. It seems thus interesting to translate in a way or another the object model to Event-B, so concepts manipulated in formulas have an equivalent in the Event-B model.

As Event-B uses the set theory to define and manipulate data, the KAOS object model could be quite easily transformed into an ERA model. Tools like DB-Main [REVER, 901] can automatically transform such model into a relational model compliant with relational databases. The relational nature of the diagram allows getting an Event-B model from it with a simple syntactic transformation. Moreover, as relational databases are the most used database management systems, the relational diagram could be used to generate SQL data definition code. This method implies more than one transformation. Another negative point is that the generated data definition in the Event-B Context and Machine may be more difficult to manipulate.

Snook *et al.* define in [Snook and Butler, 2006, yah Said et al., 2009] a method to transform a UML Class diagram into a classical B machine. This method may be adapted to transform the KAOS Object model which corresponds to a simplified UML Class diagram to an Event-B Machine and its associated Context.

From now we will take the following conventions: the name of the KAOS model elements will be those defined in the KAOS meta-model [van Lamsweerde, 2009]; the first letter of those meta-concepts will be in capital.

2.2.1 Object types and Attributes

A set `OBJECT_SET` of all possible objects belonging to a certain Object type is defined in the Context for each Object type. The set `OBJECTS` of all the existing instances of a certain Object type is defined in the Machine that will see the Context and belongs to the powerset of `OBJECT_SET`.

The domains of the Attributes have to be defined in the Context. In particular, non standard types or enumerated domains have to be specified in comprehension or in extension. Attributes are represented in the Machine by a partial or total function according to the Multiplicity of the Attribute, from an element of the `OBJECT` set to an element of the domain of the attribute. The table 2.1 gives the transformation rules for the different Multiplicities of an attribute of Object type `T`.

Table 2.1: Transformation rules for KAOS Attributes

KAOS attribute	**Corresponding function**	**Event-B Invariant**
a : type [1..1]	Total function to `TYPE`	$a \in T \rightarrow TYPE$
a : type [0..1]	Partial function to `TYPE`	$a \in T \nrightarrow TYPE$
a : type [1..n]	Total function to non-empty subset of `TYPE`	$a \in T \rightarrow \mathbb{P}_1(TYPE)$
a : type [0..n]	Total function to subsets of `TYPE`	$a \in T \rightarrow \mathbb{P}(TYPE)$

2.2.2 Associations and Specializations

Associations may be directed or not and will be represented in the Machine by functions. Table 2.2 gives the transformation rules for the different kinds of directed associations. An undirected association corresponds to two opposite directed associations and can be manage as two directed associations with an additional invariant saying that if on exists, then the other exists too. For an association linking A to B with multiplicities [a1..a2] and [b1..b2]

A —a1..a2————————————————b1..b2—B

The result in Event-B will be :

A set AtoB according to the rules in table 2.2
A set BtoA according to the rules in table 2.2
An additional invariant:
$\forall x, y \cdot (x \in A \wedge y \in B) \Leftrightarrow (AtoB(x) = y \Leftrightarrow BtoA(y) = x)$

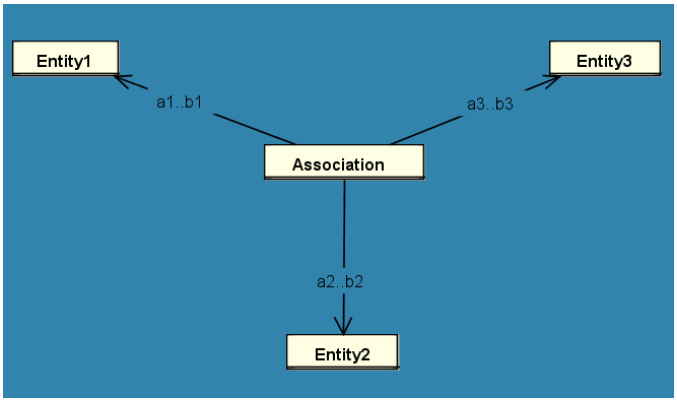

Figure 2.2: N-Ary Association are seen as an Entity with N directed Associations

As show in figure 2.2, an N-Ary Association will be seen as an Entity with N directed Associations to the different Objects of the N-Ary Association.

In case of Specialization, usually instances belong to one and only one sub-Object type and sub-Objects instances are disjoints. As stated by Snook and Butler [Snook and Butler, 2006], when translating from KAOS to Event-B, the instances of the sub-Objects will be declared as a subset of super-Object's current instances. Three Object types, one Parent and two sons Son1 and Son2 specializing Parent will become in Event-B :

$$PARENT \in \mathbb{P}(PARENT\_SET)$$
$$SON1 \in \mathbb{P}(PARENT)$$
$$SON2 \in \mathbb{P}(PARENT)$$
$$SON1 \cap SON2 = \varnothing$$

The Specialization may be more precise like in ERA, *e.g.* if all the instances must be one of a sub-Object type then the sub-Objects instances sets cover the set of super-Object instances :

$$SON1 \cup SON2 = PARENT$$

Table 2.2: Transformation rules for KAOS directed Associations

The two Object types are A and B and $a1..a2 \rightarrow b1..b2$ in the table represents the multiplicities for an association : A —a1..a2————————b1..b2—> B According to our convention, the Objects sets in Event-B will be called `A` and `B`.
The *disjoint* macro in the table is defined as: $(\forall a1, a2 \cdot (a1 \in dom(AtoB) \wedge a2 \in dom(AtoB) \wedge a1 \neq a2 \Rightarrow AtoB(a1) \cap AtoB(a2) = \varnothing))$

KAOS association multiplicity	**Corresponding function**	**Event-B Invariant**
$0..* \rightarrow 0..1$	Partial function to `B`	$AtoB \in A \nrightarrow B$
$0..* \rightarrow 1..1$	Total function to `B`	$AtoB \in A \rightarrow B$
$0..* \rightarrow 0..*$	Total function to subset of `B`	$AtoB \in A \rightarrow \mathbb{P}(B)$
$0..* \rightarrow 1..*$	Total function to non-empty subset of `B`	$AtoB \in A \rightarrow \mathbb{P}_1(B)$
$0..1 \rightarrow 0..1$	Partial injection to `B`	$AtoB \in A \rightarrowtail B$
$0..1 \rightarrow 1..1$	Total injection to `B`	$AtoB \in A \rightarrowtail B$
$0..1 \rightarrow 0..*$	Total function to subsets of `B` which don't intersect	$AtoB \in A \rightarrow \mathbb{P}(B) \wedge disjoint$
$0..1 \rightarrow 1..*$	Total function to non-empty subsets of `B` which don't intersect	$AtoB \in A \rightarrow \mathbb{P}_1(B) \wedge disjoint$
$1..* \rightarrow 0..1$	Partial surjection to `B`	$AtoB \in A \nrightarrow B$
$1..* \rightarrow 1..1$	Total surjection to `B`	$AtoB \in A \rightarrow B$
$1..* \rightarrow 0..*$	Total function to subsets of `B` which cover `B`	$AtoB \in A \rightarrow \mathbb{P}(B) \wedge \text{union}(\text{ran}(AtoB)) = B$
$1..* \rightarrow 1..*$	Total function to non-empty subsets of `B` which cover `B`	$AtoB \in A \rightarrow \mathbb{P}_1(B) \wedge \text{union}(\text{ran}(AtoB)) = B$
$1..1 \rightarrow 0..1$	Partial bijection to `B` (partial injection defined for all the elements of `B`)	$AtoB \in A \rightarrowtail B \wedge \forall b \cdot (b \in B \Rightarrow (\exists a \cdot (a \in A \wedge (a \mapsto b) \in AtoB)))$
$1..1 \rightarrow 1..1$	Total bijection to `B`	$AtoB \in A \rightarrowtail B$
$1..1 \rightarrow 0..*$	Total function to subsets of `B` which cover `B` without intersecting	$AtoB \in A \rightarrow \mathbb{P}(B) \wedge \text{union}(\text{ran}(AtoB)) = B \wedge disjoint$
$1..1 \rightarrow 0..*$	Total function to non-empty subsets of `B` which cover `B` without intersecting	$AtoB \in A \rightarrow \mathbb{P}_1(B) \wedge \text{union}(\text{ran}(AtoB)) = B \wedge disjoint$

2.3 Decomposition of the initial model according to Agents

Decomposition makes it possible to manage the complexity of models that increases through the refinement process. It may be interesting to have an early decomposition to break an initial machine into smaller pieces pertinent with the KAOS agents. This choice is made because the KAOS meta-model says that an association or an attribute can be controlled by one and only one agent [van Lamsweerde, 2009, Landtsheer, 2007b, Letier, 2001]. The idea is thus to have separate machines with the attributes monitored and controlled by the agent. Let us recall that an attribute or association is controlled by an agent if the agent performs one or more operation that modifies the attribute value and that an attribute is monitored by an agent if the attribute is an input of one or more operation performed by the agent.

Ball presents in [Ball, 2008] a description of the two techniques used to split a machine into smaller pieces. The first one, called Event-Based Decomposition or B-style decomposition [Pascal and Silva, 2009] encapsulates the variables in different machines together with the events or parts of events that concern those variables. The events that have been split will need to be synchronized in order to ensure the functionalities of the original machine. The synchronization will take place by an exchange of inputs and outputs between the synchronized machines events [Butler, 2009].

The second technique, called State-Based Decomposition or A-style decomposition [Pascal and Silva, 2009] splits the variables in different machines with some shared variables. Events are added to components to simulate how the shared variables are used in other components. Shared variables and events must be kept synchronized between the different machines during the refinement. Theoretically the system could be rebuilt into a single machine at the end of the process, but in practice this will never be done since the different machines will lead to different software components.

This State-Based Decomposition, proposed by Abrial in [Abrial, 2009b, Abrial, 2009a, Métayer et al., 2005] seems to fit more our problem. For a general model, variables and events will be distributed to several sub-machines with some of those variables presents in more than one sub-machine. It is important to notice here that the sub-machines are not refining the general machine, but are decomposing it. In the sub-machines, a distinction is made between the internal variables used only in a particular sub-machine and the shared variables used in more than one sub-machine. So, shared variables can be modified by more than one event in more than one sub-machine. Figure 2.3 shows an example of decomposition, a sub-machine `A` has an event `evtA` that will modify the value of a shared variable and another sub-machine `B` has an event `evtB` using the variable's value in its guard. To express the fact that the variable is not a constant in `B`, an event `evtExtA` will be added

to `B` corresponding to an abstraction of the event `evtA` in `A`. The added event `evtExtA` will be called an external event, which is just present in `B` to synchronize the update of the shared variable in the general machine.

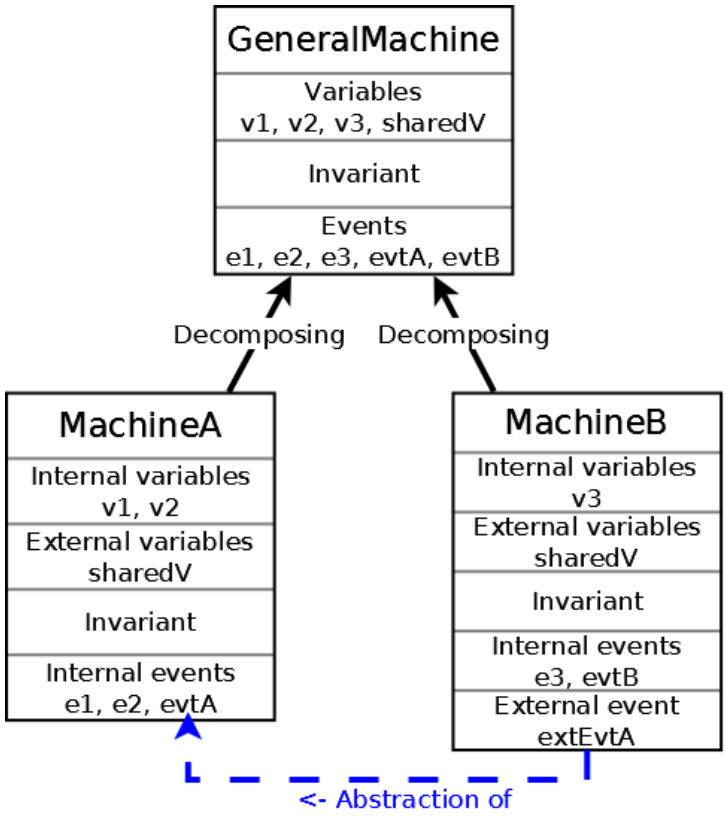

Figure 2.3: Decomposition of a general machine into two sub-machines

It is clear now that shared variables coming from the abstract machine will be replicated in each sub-machine. The problem is that each sub-machine could normally refine its variables and the same replicated variable could be refined in one way in one refinement and in another way in another refinement. If this happens, the two sub-machines can't communicate any longer as they are not using the same convention on the shared variable. Such a variable has a special status in the sub-machines where they stay saying that this variable has to be always present in the state space of any refinement of the machine. A shared variable can thus not be data-refined or if it is, the variable has to be refined in the same way in each sub-model using the variable, which can be quite heavy.

2.3.1 State-Based Decomposition

We propose to use the State-Based Decomposition after an initial creation of the Event-B model from the KAOS object model, as presented in section 2.2, with one sub-machine per agent. The reason of this choice is simple, the KAOS meta-model states that an attribute or association cannot be controlled by more than one agent [van Lamsweerde, 2009, Letier, 2001, Landtsheer, 2007b]. So it means that in Event-B, a shared variable will be updated in one and only one sub-machine, while an external event will be placed with each variable coming from the KAOS object model in all other sub-machines.

The question is: do we have to place each variable coming from the KAOS object model in all sub-machines? On one side, if we place the variables coming from the controlled and monitored attributes and associations of the KAOS object model only in the sub-machines representing the concerned agent, the model in its all will be more readable. On the other side, decomposition link is for now informal and not implemented in existing tools [RODIN, v 11] and have thus to be done manually. Moreover, the re-composition of all sub-machines in one big machine proposed in [Métayer et al., 2005], which could be used at some moment in the development process as a verification of the consistency of the model, could not be done in RODIN since a machine cannot refine more than one other machine. It could thus be interesting to have a more "concrete" decomposition.

For recall, an external event representing the update of a certain shared variables has to be an abstraction of the concrete event updating the variable in another sub-machine. Since KAOS meta-model impose to have only one agent controlling the update of an attribute or an association, the update of a variable coming from the KAOS object model will not be performed in more than one sub-machine. The idea is to add to the general machine coming from the KAOS object model very general update operations for each variable, and generate from this machine one refinement per agent. The variables that are not controller by the agent will be marked as shared variables and the events updating those variables will be marked as external events in the sub-machines. Those events and variables cannot be refined one the sub-machine or its refinements. All the events that update the controlled variables of the agent will be refinements of the general update event defined in the general machine. The re-composition of sub-machines will simply be a new machine, declared as a refinement of the initial machine generated from the KAOS object model where each non-external events and internal variables coming from the different sub-machines will be copy-pasted. By doing so, we guaranty that each external event is indeed an abstraction of the update of a non-controlled shared variable, because of the refinement link. The cost here is to have each shared variables and each abstract update event of the non-controlled variables repeated in each machine and its refinement, whether the corresponding agent is controlling or monitoring the variable

or not. This may be overcome in the modelling tools by hiding in a sub-machine the variables and corresponding external update events that are not controlled or monitored by the corresponding agent.

Example

Here is a small example inspired by the mine pump model presented in [Aziz et al., 2009]. In this model we have a mine that has to be kept safe from flooding and explosion. For this we have a mine pump that start pumping if the water level is too high and if there is no methane detected.

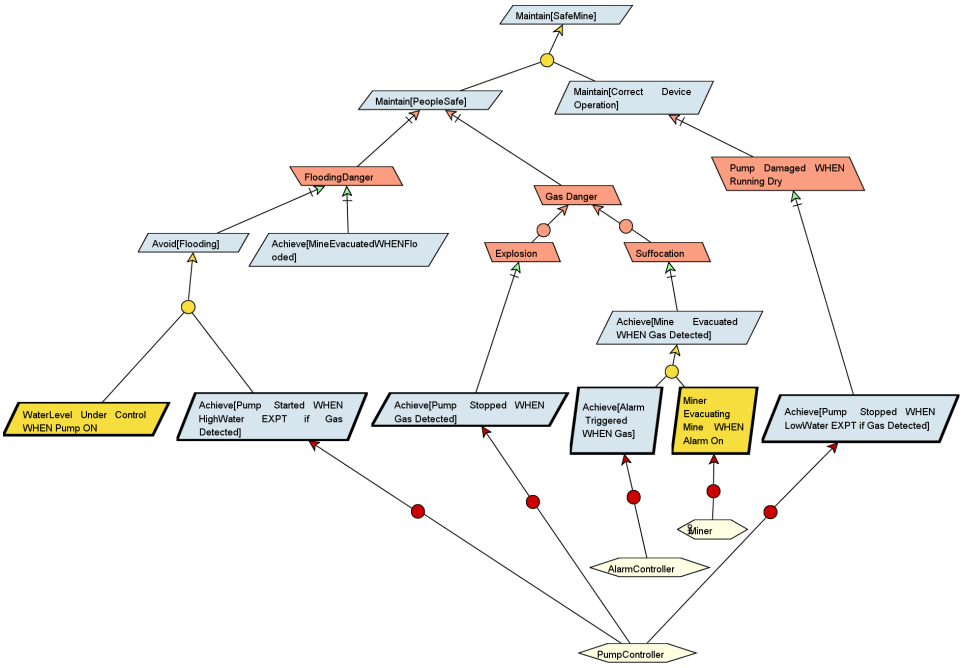

Figure 2.4: Mine pump goal model

Figure 2.5 presents the goal model and the different agents responsible for the requirements and expectations. Figure 2.5 shows the agent model with controlled and monitored objects: the PumpController controls the pump attribute and monitors the methane and waterLevel attributes, the Alarm-Controller controls the bell attribute and monitors the methane attribute, the WaterLevelSensor controls the waterLevelAttribute, the MethaneSensor controls the methane attribute and the Miner monitors the bell attribute.

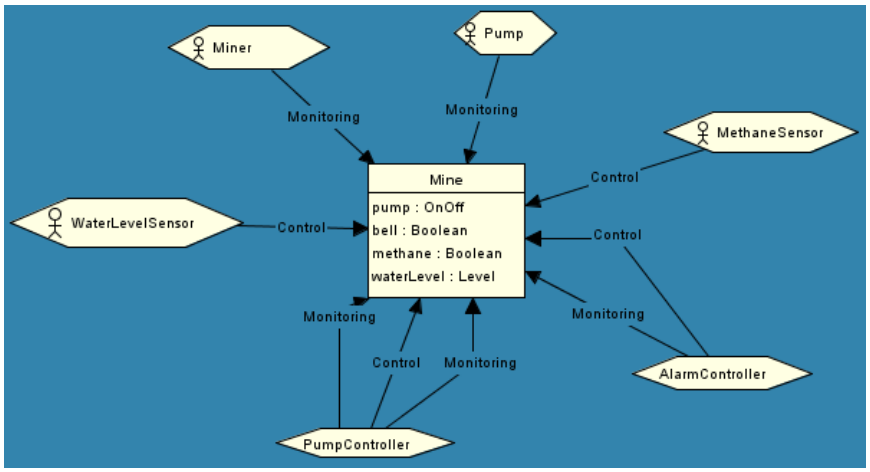

Figure 2.5: Mine pump agent model

By applying the procedure described in section 2.2, we get an initial Context in listing 2.1 and an initial machine in listing 2.2 describing the objects of the KAOS object model. The initial machine includes the attributes and the update methods for all those attributes, note here that in the listing 2.2 only the update method for the pump has been shown. The update methods of the others attributes follows the same pattern. The complete machines of this example can be found in annex B.

Listing 2.1: Mine pump example: Initial context

CONTEXT MineContext
SETS
ONOFF, LEVEL, MINE_SET
CONSTANTS
ON, OFF, LOW, MEDIUM, HIGH, M
AXIOMS
axm1 : $partition(ONOFF, \{ON\}, \{OFF\})$
axm2 : $partition(LEVEL, \{LOW\}, \{MEDIUM\}, \{HIGH\})$
axm3 : $partition(MINE\_SET, \{M\})$
END

Listing 2.2: Mine pump example: Initial machine

MACHINE MinePump
SEES MineContext
VARIABLES
MINE, pump, bell, methane, waterLevel
INVARIANTS
inv1 : $MINE \in \mathbb{P}(MINE\_SET)$

```
      inv2 : pump ∈ MINE → ONOFF
      inv3 : bell ∈ MINE → BOOL
      inv4 : methane ∈ MINE → BOOL
      inv5 : waterLevel ∈ MINE → LEVEL
EVENTS
Initialisation
      begin
            act1 : MINE, pump, bell, methane, waterLevel := ∅, ∅, ∅, ∅, ∅
      end
Event  updatePump ≙
      any
            m
            status
      where
            grd1 : m ∈ MINE
            grd2 : status ∈ ONOFF
      then
            act1 : pump(m) := status
      end
END
```

Starting from this, machines will be created by refining the initial machine for each agent of the KAOS model. The listing 2.3 shows the machine defined for the PumpController. This machine and all the other machines of this example can be found in annex B. The re-composed machine can also be found in listing B.7 in annex B where the update methods have been replaced by their refinements in the different sub-machines. Figures 2.6 and 2.7 shows a summary of the created machines.

Listing 2.3: Mine pump example: PumpController machine

```
MACHINE  PumpController
REFINES  MinePump
SEES  MineContext
VARIABLES
      MINE, pump, bell, methane, waterLevel
EVENTS
Initialisation
      extended
      begin
            act1 : MINE, pump, bell, methane, waterLevel := ∅, ∅, ∅, ∅, ∅
      end
Event  high_water_detected ≙
      Internal Event
refines  updatePump
      any
            m
      where
            grd2 : m ∈ MINE
            grd1 : waterLevel(m) = HIGH
            grd3 : methane(m) = FALSE
      with
            status : status = ON
      then
            act1 : pump(m) := ON
      end
Event  low_water_detected ≙
      Internal Event
```

refines *updatePump*
  any
    m
  where
    grd1 : $m \in MINE$
    grd2 : $waterLevel(m) = LOW$
  with
    status : $\texttt{status} = \texttt{OFF}$
  then
    act1 : $pump(m) := OFF$
  end
Event *updateBell* $\widehat{=}$
  External Event
extends *updateBell*
  any
    m
    status
  where
    grd1 : $\texttt{m} \in \texttt{MINE}$
    grd2 : $\texttt{status} \in \texttt{BOOL}$
  then
    act1 : $\texttt{bell}(\texttt{m}) := \texttt{status}$
  end
Event *updateMethane* $\widehat{=}$
  External Event
extends *updateMethane*
  any
    m
    status
  where
    grd1 : $\texttt{m} \in \texttt{MINE}$
    grd2 : $\texttt{status} \in \texttt{BOOL}$
  then
    act1 : $\texttt{methane}(\texttt{m}) := \texttt{status}$
  end
Event *updateWaterLevel* $\widehat{=}$
  External Event
extends *updateWaterLevel*
  any
    m
    level
  where
    grd1 : $\texttt{m} \in \texttt{MINE}$
    grd2 : $\texttt{level} \in \texttt{LEVEL}$
  then
    act1 : $\texttt{waterLevel}(\texttt{m}) := \texttt{level}$
  end
END

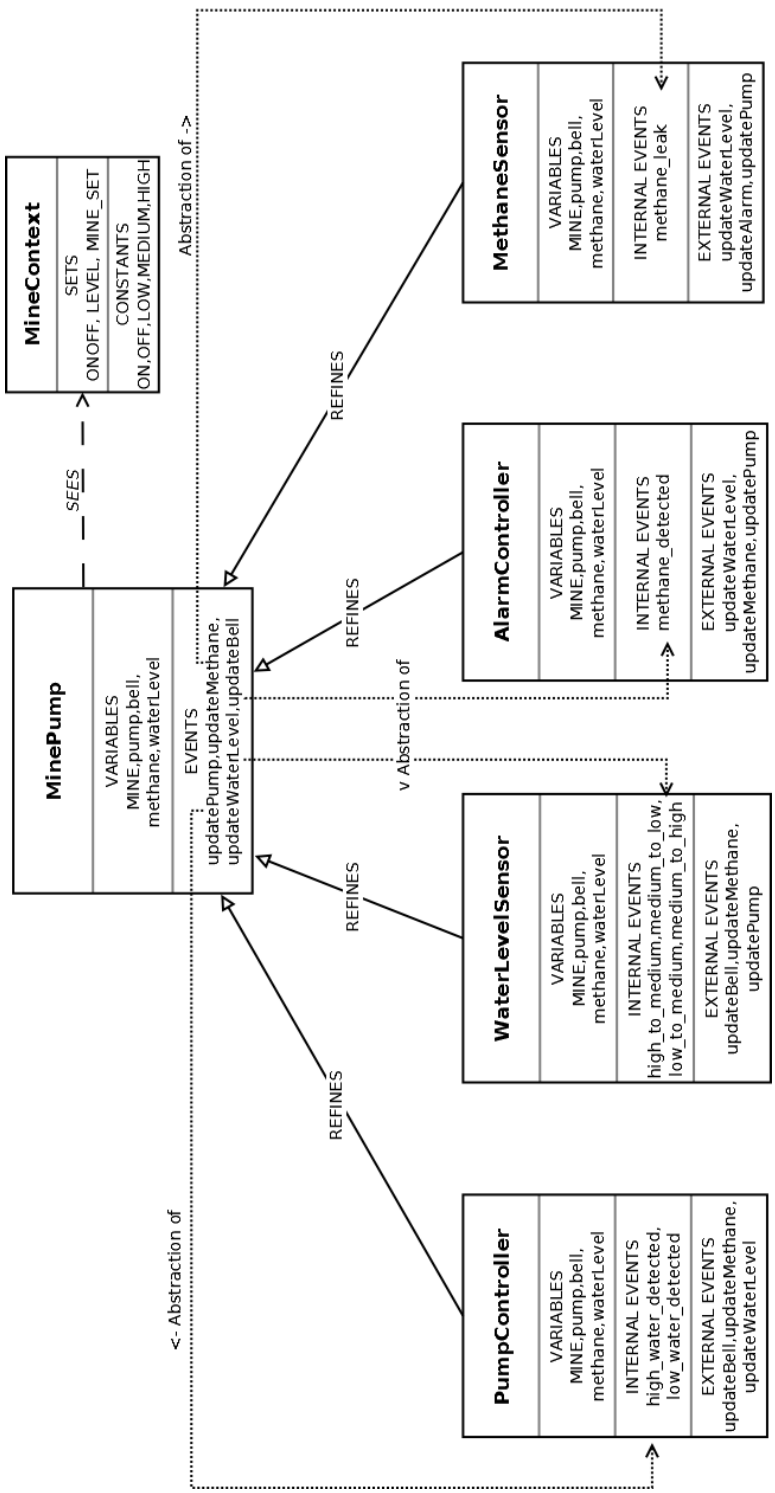

Figure 2.6: Decomposition of the initial machine

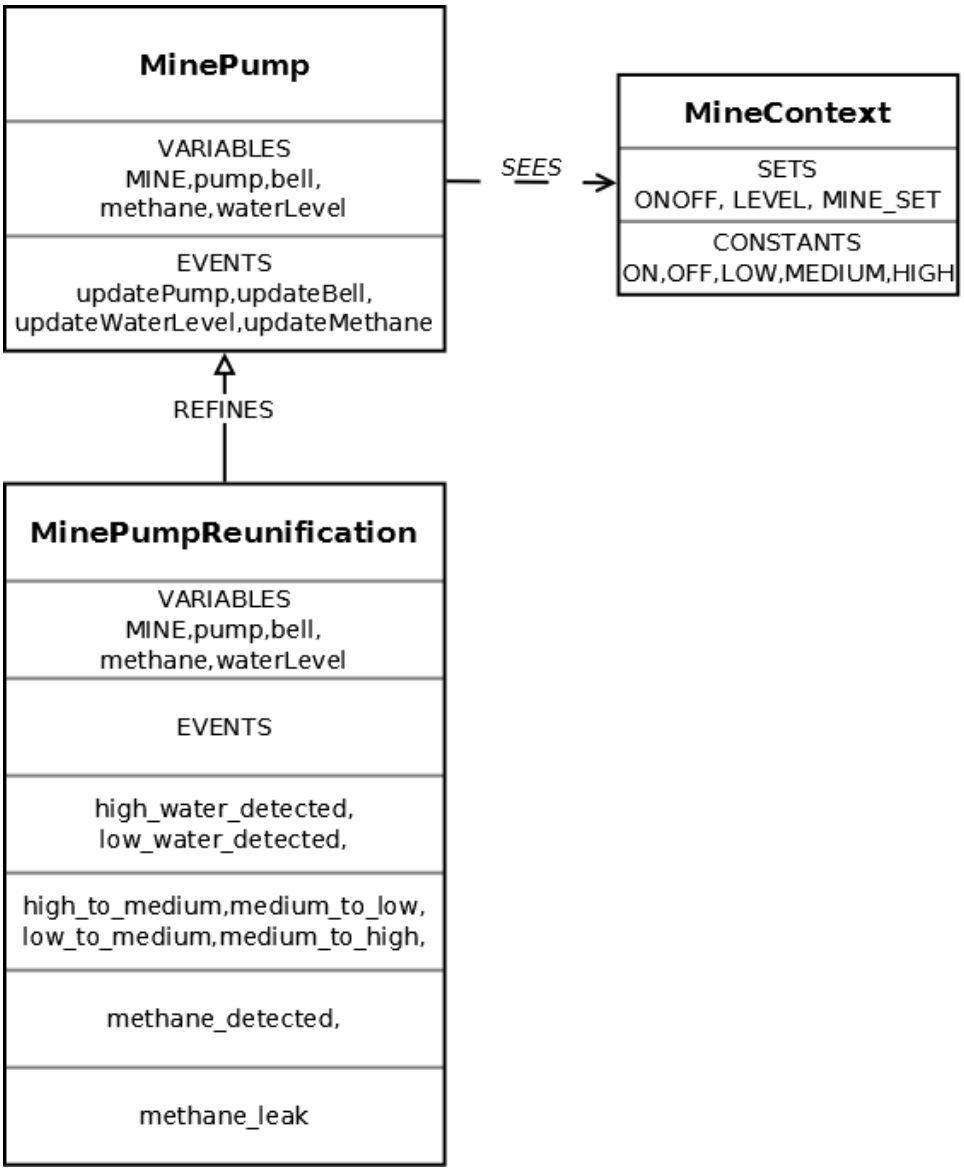

Figure 2.7: Recomposition of the initial machine

2.4 Traceability between KAOS and Event-B

The idea here is to have rules to justify every element in the Event-B model by an element coming from the requirement. The goal is to avoid over-specification and guaranty that if requirements are discovered or corrected during the elaboration of the Event-B model, the requirements documents will be adapted too.

2.4.1 Definitions

Before going further, let us introduce some definitions used to express rules hereafter :

- An abstract object in KAOS is an entity, an agent or an event. Both agents and events may, like in UML, have a "data part" with attributes.
- An attribute domain in KAOS is a domain of values defining the type of an attributes. This domain may be built-in or user defined.
- An N-Ary association in KAOS is an association with a multiplicity strictly greater than two.
- An undirected association is a bidirectional association.
- An IsA link in KAOS is a specialization link taking place between two abstract objects.
- A domain property in KAOS is a property guaranteed by the environment. This property is assumed to be always true.

We also define here what are the initial context and machine :

Definition 2.4.1. *The initial context is the context derived from the KAOS object model.*

Definition 2.4.2. *The initial machine is the machine derived from the KAOS object model with all its variables, invariants and events justified by elements of the KAOS object model.*

Now we are clear with the vocabulary, let us define criteria for the Event-B model derived from the KAOS model. The links that we are talking about are Derivation links as defined by van Lamsweerde in his hierarchy [van Lamsweerde, 2009]. A dependency links between two model A and B express the fact that changing A may require changing B. This kind of link is vertical in the sense that they take place for a single version, opposed to horizontal links, such as a variant or revision link that take place between different versions.

2.4.2 Initial model

First, we will define criteria for the initial machine and context. Those two elements are build by the transformations described in section 2.2. Those criteria are expressed must be respected to keep the Event-B model consistent with the KAOS model.

Initial context

Here are the criteria for the sets, axioms and constants that can be found in the initial context.

Criterion 2.4.1. *Each carrier set in the initial context must be linked to one abstract object, or one attribute domain or one N-Ary association.*

Criterion 2.4.2. *Each constant in the initial context must be linked to an attribute domain.*

Criterion 2.4.3. *Each axiom in the initial context must be linked to an attribute domain.*

Initial machine

Here are the criteria defined for the invariants, variables and events defined for the update of those variables.

Criterion 2.4.4. *Each variable in the initial machine must be linked to one abstract object or one attribute or one directed association or one undirected association or one N-Ary association.*

Criterion 2.4.5. *Each invariant in the initial machine must be linked to one abstract object or one directed association or one undirected association or an IsA link or an N-Ary association or a domain property.*

We will call an update event an event corresponding to the update of one KAOS element which can be an abstract object, an attribute, an N-Ary association, a directed association or an undirected association.

Criterion 2.4.6. *Each event in the initial machine must be an update event and is thus linked to one abstract object or one attribute or one directed association or one undirected association or one N-Ary association.*

Note that one element in KAOS may be translated in more than one variable in Event-B, *e.g.* the undirected association that is transformed into two sets and an additional invariant.

Criterion 2.4.7. *Each variable in the initial machine must appear in one and only one update event.*

Criterion 2.4.8. *Each KAOS element which can be an abstract object, an attribute, an N-Ary association, a directed association or an undirected association that is controlled by an agent must appear in the initial machine.*

2.4.3 Other machines in the Event-B model

Starting from the initial machine and context, other machines will be defined for agents according to our proposed approach. Those machines will then be refines independently to describe the behavior of each agent of the system under study. Here are criteria for those machines.

Machines

A machine is said as directly linked to an agent of the KAOS model if it is refining the initial machine and if a link is defined between the machine and one agent of the system. A machine directly linked to an agent is a part of the decomposition of the initial machine.

A machine is said as indirectly linked to an agent if it is refining a machine directly linked to an agent or a machine indirectly linked to an agent.

A machine will be said as linked to an agent if it is directly linked to an agent or indirectly linked to an agent.

Criterion 2.4.9. *Each machine in the Event-B model that is not the initial machine must be linked to one agent or must be a re-composition, as defined in section 2.3, of several machines.*

Events

An event is said as linked to a requirement or an expectation if it is directly linked to a requirement or an expectation or if it is refining an event linked to a requirement or an expectation.

Criterion 2.4.10. *Each event in the machines that are not the initial machine must be linked to a requirement or expectation under the responsibility of the agent linked to the machine.*

Note that a recomposed machine is implicitly linked to all the agents corresponding to the machines that are participating in the re-composition.

We will say that an event refines another event if it refines it directly or if it refines a third event that refines the other event.

Criterion 2.4.11. *If an event in a machine that is not the initial machine updates the value of variables corresponding to a KAOS element, which can be an abstract object, an attribute, an N-Ary association, a directed association or an undirected association then the event must refine the update event corresponding to this KAOS element.*

Criterion 2.4.12. *If an event in a machine that is not the initial machine refines an update event, the agent linked to the machine must control the KAOS element, which can be an abstract object, an attribute, an N-Ary association, a directed association or an undirected association in the KAOS model.*

Criterion 2.4.13. *Each update event in the initial machine may be refined by events in machines corresponding to at most one and only one agent.*

2.5 What happens if ...

2.5.1 ... an element is added in the KAOS object model

Adding an element to the KAOS object model will result in a modification of the initials machine and context. This element is added according to the rules described in section 2.2 to the initial machine and context and will be propagate to all machines and context refining them.

2.5.2 ... an element is removed from the KAOS object model

When an element is removed from the KAOS object model, the invariants, variables, update event, sets and axioms issued from its translation, according to the rules described in section 2.2, in Event-B are removed from the initial machine and context and all the machines and contexts that are refining them. Note that before deleting a piece of element, all the control and monitor links will be removed too. For recall a KAOS meta-constraint impose that all the elements used to define goals, requirements and expectations must be defined in the object model. An element will thus not be removed while at least requirements and expectations are using it and thus events linked to requirements and expectations will stay correct, even elements are deleted from the Event-B model.

2.5.3 ... an agent is added in the KAOS model

Adding an agent to the KAOS model means that a new active entity has been identified. The Event-B model will thus be enriched by a new machine, decomposing the initial machine. For recall, the decomposition link is for now informal in RODIN [RODIN, v 11] and we propose to use a refining link between the new machine and the initial machine (see section 2.3).

2.5.4 ... an agent is removed from the KAOS model

Removing an agent from the KAOS model means that an active part of the system is removed. All the responsibility links between the agent and the requirements/expectations will probably be moved to other agents before removing it. As an agent may be responsible for a requirement/expectation if and only if he can control all the data that are modified by the requirement/-expectation and monitor all the data read by the requirement/expectation, all the monitor and control links will also probably be moved before the deletion of an agent.

If an agent is removed from the KAOS model, the corresponding machine and all its sub-machines will be removed from the Event-B model. If one of those machines has been used in a re-composition, all the events coming from the machine will be removed of the decomposition. Pay attention that if the agent was still controlling a piece of data when it is removed and that one of the deleted event in the re-composition was refining the update event of this piece of data, the general update event coming from the initial machine has to be added in the re-composed machine.

2.5.5 ... a control link is added in the KAOS model

If a control link is added in the KAOS model, the update event of the controlled piece of data will become an internal event that may be refined in the machine corresponding to the agent. Of course, we suppose here that the KAOS meta-constraint saying that a piece of data can be controlled by one and only one agent is respected.

2.5.6 ... a control link is removed from the KAOS model

In the Event-B model, when an agent is controlling a piece of data, it means that the update event of this piece of data is an internal event in the machine linked to the Agent. Removing a control link will thus means that the agent can no longer modify a certain piece of data. All the events in the machine linked to the agent and its refinements that are refining the update event of the piece of data have to be removed. They will be replaced by the update event coming from the initial machine and will be marked as external.

Deleting a control link may only occur in KAOS when the agent is no longer responsible for requirements/expectations that update the previously controlled element. A more frequent situation will be to move requirements/-expectations responsibilities to another agent and in the same time, move control and monitor links needed to be responsible for those requirements/-expectations to this other agent too.

2.5.7 ... a monitor link is added in the KAOS model

When a monitor link is added to the KAOS model, it means that an agent will be notified when a certain piece of data is updated. In Event-B, it means that the update event linked to this piece of data is executed in the machine linked to the agent and all its refinements as an external event.

As we are using refinement links in place of decomposition links, all the events of the initial machine have to be refined by one or more event in the agent's machine. [Métayer et al., 2005] When the agent is monitoring a piece of data linked to the update event, this update event in the agent's machine will correspond to a copy paste of the abstract event. Note that it is the

default RODIN [RODIN, v 11] behavior when a refinement of a machine is created.

2.5.8 ... a monitor link is removed from the KAOS model

The update event, variables and invariants linked to the previously monitored piece of data may be hidden to the analyst in a tool, but in the effective Event-B model nothing happens because of the refining link between the agent's machine and the initial machine. As explained in section 2.3, this is the small cost to pay for using refinement links in place of decomposition links.

2.5.9 ... a responsibility links is moved from an agent to another

A responsibility link in KAOS is translated into an event or an invariant in the agent's machine of the Event-B model. If it is an event, it refines all the update events corresponding to the data that are modified by the requirement/expectation. The agent has thus the ability to control those data in the KAOS model. Moving a responsibility from an agent to another will thus mean that the implied control links will be moved in the same time.

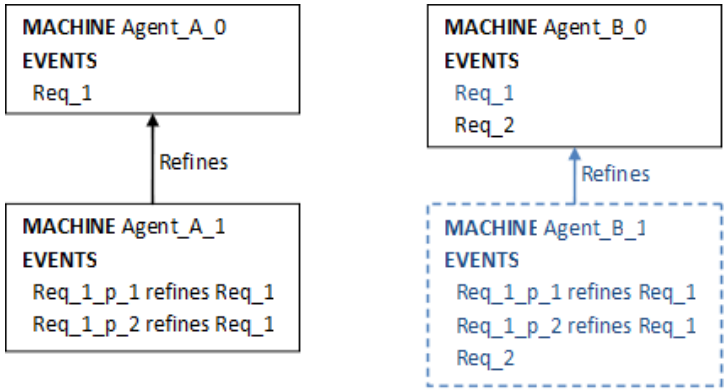

Figure 2.8: Moving responsibility link in Event-B

At the agents' machines level it means that the events linked to the requirement/expectation is moved from a machine to another. The update events of the data concerned by the moved control links will be replaced by the update event coming from the initial machine and will be marked as external. If the event linked to the requirement/expectation has already been refined in sub-machines, the refinements may be moved from the previous agent's "refinement tree" to the new one by completing the actual machines and creating new ones if the new tree is shorter than the previous one. Figure

2.8 shows an example of a refined requirement moved from the agent A to the agent B, where the agent B's machine has not yet been refined. A refinement is created, in blue dotted on the figure, to have the same refinement level as agent A's machine.

Appendix A

Linear Temporal Logic notations

This section presents the temporal operators used in KAOS [van Lamsweerde, 2009].

A history H is a function : $H : \mathbb{N} \rightarrow State(X)$ where X is the set of system variables and $State(X)$ is the set of all possible states for the corresponding variable in X.

If temporal assertion P satisfied by a history H at time position i, we say that :

$$(H, i) \models P$$

If i is the initial position 0, then the assertion P is said to be satisfied by the entire history H :

$$(H, 0) \models P$$

A.1 Time operators

The tables A.1 and A.2 summarize the time operators used with KAOS and they associated semantics.

Table A.1: Future time operators

Notation	**Informal Explanation**	**Semantic**
$\Diamond P$	Sooner or later P	$(H,i) \models \Diamond P$ **iff** $\exists j, j \geqslant i : (H,j) \models P$
$\Box P$	Always P	$(H,i) \models \Box P$ **iff** $\forall j, j \geqslant i : (H,j) \models P$
$P\ \boldsymbol{U}\ Q$	Always P until Q	$(H,i) \models P\ \boldsymbol{U}\ Q$ **iff** $(\exists j, j \geqslant i : (H,j) \models Q) \wedge (\forall k, i \leqslant k < j : (H,k) \models P)$
$P\ \boldsymbol{W}\ Q$	Always P unless Q	$(H,i) \models P\ \boldsymbol{W}\ Q$ **iff** $((H,i) \models P\ \boldsymbol{U}\ Q) \vee ((H,i) \models \Box P)$
$\circ P$	Next P	$(H,i) \models \circ P$ **iff** $(H,i+1) \models P$
$P \Rightarrow Q$	P entails Q	Equivalent to $\Box(P \rightarrow Q)$
$P \Leftrightarrow Q$	P is congruent to Q	Equivalent to $\Box(P \leftrightarrow Q)$

Table A.2: Past time operators

Notation	**Informal Explanation**	**Semantic**
$\blacklozenge P$	Some time in the past P	$(H,i) \models \blacklozenge P$ **iff** $\exists j, j \leqslant i : (H,j) \models P$
$\blacksquare P$	P has always been	$(H,i) \models \blacksquare P$ **iff** $\forall j, j \leqslant i : (H,j) \models P$
$P\ \boldsymbol{S}\ Q$	Always P in the past since Q	$(H,i) \models P\ \boldsymbol{S}\ Q$ **iff** $(\exists j, j \leqslant i : (H,j) \models Q) \wedge (\forall k, j < k \leqslant i : (H,k) \models P)$
$P\ \boldsymbol{B}\ Q$	Always P in the past back to Q	$(H,i) \models P\ \boldsymbol{B}\ Q$ **iff** $((H,i) \models P\ \boldsymbol{s}\ Q) \vee ((H,i) \models \blacksquare P)$
$\bullet P$	Previous P	$(H,i) \models \bullet P$ **iff** $(H,i-1) \models P$ with $i > 0$
$@P$	To P	Equivalent to $(\bullet \neg P) \wedge P$

Appendix B

Decomposition according to Agents: Mine pump example

This annex present the complete machines of the mine pump example described in section 2.3.

Listing B.1: Mine pump example: Initial context

CONTEXT MineContext
SETS
ONOFF
LEVEL
MINE_SET
CONSTANTS
ON
OFF
LOW
MEDIUM
HIGH
M
AXIOMS
axm1 : $partition(ONOFF, \{ON\}, \{OFF\})$
axm2 : $partition(LEVEL, \{LOW\}, \{MEDIUM\}, \{HIGH\})$
axm3 : $partition(MINE\_SET, \{M\})$
END

Listing B.2: Mine pump example: Initial machine

MACHINE MinePump
SEES MineContext
VARIABLES
MINE
pump
bell
methane
waterLevel
INVARIANTS

inv1 : $MINE \in \mathbb{P}(MINE\_SET)$
inv2 : $pump \in MINE \rightarrow ONOFF$
inv3 : $bell \in MINE \rightarrow BOOL$
inv4 : $methane \in MINE \rightarrow BOOL$
inv5 : $waterLevel \in MINE \rightarrow LEVEL$
inv6 : $dom(pump) = MINE$
inv7 : $dom(bell) = MINE$
inv8 : $dom(methane) = MINE$
inv9 : $dom(waterLevel) = MINE$

EVENTS
Initialisation
begin
act1 : $MINE := \varnothing$
act2 : $pump := \varnothing$
act3 : $bell := \varnothing$
act4 : $methane := \varnothing$
act5 : $waterLevel := \varnothing$
end
Event $updatePump \mathrel{\widehat{=}}$
any
m
$status$
where
grd1 : $m \in MINE$
grd2 : $status \in ONOFF$
then
act1 : $pump(m) := status$
end
Event $updateBell \mathrel{\widehat{=}}$
any
m
$status$
where
grd1 : $m \in MINE$
grd2 : $status \in BOOL$
then
act1 : $bell(m) := status$
end
Event $updateMethane \mathrel{\widehat{=}}$
any
m
$status$
where
grd1 : $m \in MINE$
grd2 : $status \in BOOL$
then
act1 : $methane(m) := status$
end
Event $updateWaterLevel \mathrel{\widehat{=}}$
any
m
$level$
where
grd1 : $m \in MINE$
grd2 : $level \in LEVEL$
then
act1 : $waterLevel(m) := level$
end
Event $addMine \mathrel{\widehat{=}}$
when
grd1 : $MINE = \varnothing$
then
act1 : $MINE := \{M\}$
act2 : $pump(M) := OFF$

```
            act3 : bell(M) := FALSE
            act4 : methane(M) := FALSE
            act5 : waterLevel(M) := LOW
      end
END
```

Listing B.3: Mine pump example: PumpController machine

```
MACHINE  PumpController
REFINES  MinePump
SEES  MineContext
VARIABLES
      MINE
      pump
      bell
      methane
      waterLevel
EVENTS
Initialisation
      extended
      begin
            act1 : MINE := ∅
            act2 : pump := ∅
            act3 : bell := ∅
            act4 : methane := ∅
            act5 : waterLevel := ∅
      end
Event  high_water_detected ≙
      Internal Event
refines  updatePump
      any
            m
      where
            grd2 : m ∈ MINE
            grd1 : waterLevel(m) = HIGH
            grd3 : methane(m) = FALSE
      with
            status : status = ON
      then
            act1 : pump(m) := ON
      end
Event  low_water_detected ≙
      Internal Event
refines  updatePump
      any
            m
      where
            grd1 : m ∈ MINE
            grd2 : waterLevel(m) = LOW
      with
            status : status = OFF
      then
            act1 : pump(m) := OFF
      end
Event  updateBell ≙
      External Event
extends  updateBell
      any
            m
            status
      where
            grd1 : m ∈ MINE
            grd2 : status ∈ BOOL
```

```
      then
           act1 : bell(m) := status
      end
Event  updateMethane ≙
      External Event
extends  updateMethane
      any
           m
           status
      where
           grd1 : m ∈ MINE
           grd2 : status ∈ BOOL
      then
           act1 : methane(m) := status
      end
Event  updateWaterLevel ≙
      External Event
extends  updateWaterLevel
      any
           m
           level
      where
           grd1 : m ∈ MINE
           grd2 : level ∈ LEVEL
      then
           act1 : waterLevel(m) := level
      end
END
```

Listing B.4: Mine pump example: WaterLevelSensor machine

```
MACHINE  WaterLevelSensor
REFINES  MinePump
SEES  MineContext
VARIABLES
      MINE
      pump
      bell
      methane
      waterLevel
EVENTS
Initialisation
      extended
      begin
           act1 : MINE := ∅
           act2 : pump := ∅
           act3 : bell := ∅
           act4 : methane := ∅
           act5 : waterLevel := ∅
      end
Event  high_to_medium ≙
      Internal Event
refines  updateWaterLevel
      any
           m
      where
           grd1 : m ∈ MINE
           grd2 : waterLevel(m) = HIGH
      with
           level : level = MEDIUM
      then
           act1 : waterLevel(m) := MEDIUM
      end
```

Event *medium_to_low* $\widehat{=}$
Internal Event
refines *updateWaterLevel*
any
m
where
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = MEDIUM$
with
level : level = LOW
then
act1 : $waterLevel(m) := LOW$
end
Event *low_to_medium* $\widehat{=}$
Internal Event
refines *updateWaterLevel*
any
m
where
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = LOW$
with
level : level = MEDIUM
then
act1 : $waterLevel(m) := MEDIUM$
end
Event *medium_to_high* $\widehat{=}$
Internal Event
refines *updateWaterLevel*
any
m
where
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = MEDIUM$
with
level : level = HIGH
then
act1 : $waterLevel(m) := HIGH$
end
Event *updatePump* $\widehat{=}$
External Event
extends *updatePump*
any
m
status
where
grd1 : m ∈ MINE
grd2 : status ∈ ONOFF
then
act1 : pump(m) := status
end
Event *updateBell* $\widehat{=}$
External Event
extends *updateBell*
any
m
status
where
grd1 : m ∈ MINE
grd2 : status ∈ BOOL
then
act1 : bell(m) := status
end
Event *updateMethane* $\widehat{=}$
External Event
extends *updateMethane*
any
m

```
        status
    where
        grd1 : m ∈ MINE
        grd2 : status ∈ BOOL
    then
        act1 : methane(m) := status
    end
END
```

Listing B.5: Mine pump example: AlarmController machine

```
MACHINE  AlarmController
REFINES  MinePump
SEES  MineContext
VARIABLES
    MINE
    pump
    bell
    methane
    waterLevel
EVENTS
Initialisation
    extended
    begin
        act1 : MINE := ∅
        act2 : pump := ∅
        act3 : bell := ∅
        act4 : methane := ∅
        act5 : waterLevel := ∅
    end
Event  methane_detected ≙
    Internal Event
refines  updateBell
    any
        m
    where
        grd1 : m ∈ MINE
        grd2 : methane(m) = TRUE
        grd3 : bell(m) = FALSE
    with
        status : status = TRUE
    then
        act1 : bell(m) := TRUE
    end
Event  updatePump ≙
    External Event
extends  updatePump
    any
        m
        status
    where
        grd1 : m ∈ MINE
        grd2 : status ∈ ONOFF
    then
        act1 : pump(m) := status
    end
Event  updateMethane ≙
    External Event
extends  updateMethane
    any
        m
        status
    where
        grd1 : m ∈ MINE
```

```
            grd2 : status ∈ BOOL
      then
            act1 : methane(m) := status
      end
Event  updateWaterLevel ≙
      External Event
extends  updateWaterLevel
      any
            m
            level
      where
            grd1 : m ∈ MINE
            grd2 : level ∈ LEVEL
      then
            act1 : waterLevel(m) := level
      end
END
```

Listing B.6: Mine pump example: MethaneSensor machine

```
MACHINE  MethaneSensor
REFINES  MinePump
SEES  MineContext
VARIABLES
      MINE
      pump
      bell
      methane
      waterLevel
EVENTS
Initialisation
      extended
      begin
            act1 : MINE := ∅
            act2 : pump := ∅
            act3 : bell := ∅
            act4 : methane := ∅
            act5 : waterLevel := ∅
      end
Event  methane_leak ≙
      Internal Event
refines  updateMethane
      any
            m
      where
            grd1 : m ∈ MINE
      with
            status : status = TRUE
      then
            act1 : methane(m) := TRUE
      end
Event  updatePump ≙
      External Event
extends  updatePump
      any
            m
            status
      where
            grd1 : m ∈ MINE
            grd2 : status ∈ ONOFF
      then
            act1 : pump(m) := status
      end
```

Event *updateBell* $\widehat{=}$
External Event
extends *updateBell*
any
m
status
where
grd1 : m ∈ MINE
grd2 : status ∈ BOOL
then
act1 : bell(m) := status
end
Event *updateWaterLevel* $\widehat{=}$
External Event
extends *updateWaterLevel*
any
m
level
where
grd1 : m ∈ MINE
grd2 : level ∈ LEVEL
then
act1 : waterLevel(m) := level
end
END

Listing B.7: Mine pump example: re-composed machine

MACHINE MinePumpReunification
REFINES MinePump
SEES MineContext
VARIABLES
MINE
pump
bell
methane
waterLevel
INVARIANTS
inv1 : $MINE \in \mathbb{P}(MINE\_SET)$
inv2 : $pump \in MINE \rightarrow ONOFF$
inv3 : $bell \in MINE \rightarrow BOOL$
inv4 : $methane \in MINE \rightarrow BOOL$
inv5 : $waterLevel \in MINE \rightarrow LEVEL$
EVENTS
Initialisation
begin
act1 : $MINE := \varnothing$
act2 : $pump := \varnothing$
act3 : $bell := \varnothing$
act4 : $methane := \varnothing$
act5 : $waterLevel := \varnothing$
end
Event *methane_detected* $\widehat{=}$
Internal Event
refines *updateBell*
any
m
where
grd1 : $m \in MINE$
grd2 : $methane(m) = TRUE$
grd3 : $bell(m) = FALSE$

with
status : status = TRUE
then
act1 : $bell(m) := TRUE$
end
Event $methane\_leak \widehat{=}$
Internal Event
refines *updateMethane*
any
m
where
grd1 : $m \in MINE$
with
status : status = TRUE
then
act1 : $methane(m) := TRUE$
end
Event $high\_water\_detected \widehat{=}$
Internal Event
refines *updatePump*
any
m
where
grd2 : $m \in MINE$
grd1 : $waterLevel(m) = HIGH$
grd3 : $methane(m) = FALSE$
with
status : status = ON
then
act1 : $pump(m) := ON$
end
Event $low\_water\_detected \widehat{=}$
Internal Event
refines *updatePump*
any
m
where
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = LOW$
with
status : status = OFF
then
act1 : $pump(m) := OFF$
end
Event $high\_to\_medium \widehat{=}$
Internal Event
refines *updateWaterLevel*
any
m
where
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = HIGH$
with
level : level = MEDIUM
then
act1 : $waterLevel(m) := MEDIUM$
end
Event $medium\_to\_low \widehat{=}$
Internal Event
refines *updateWaterLevel*
any
m
where
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = MEDIUM$
with
level : level = LOW
then

act1 : $waterLevel(m) := LOW$
end
Event $low\_to\_medium \widehat{=}$
Internal Event
refines $updateWaterLevel$
any
m
where
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = LOW$
with
level : $\texttt{level} = \texttt{MEDIUM}$
then
act1 : $waterLevel(m) := MEDIUM$
end
Event $medium\_to\_high \widehat{=}$
Internal Event
refines $updateWaterLevel$
any
m
where
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = MEDIUM$
with
level : $\texttt{level} = \texttt{HIGH}$
then
act1 : $waterLevel(m) := HIGH$
end
Event $addMine \widehat{=}$
extends $addMine$
when
grd1 : $\texttt{MINE} = \varnothing$
then
act1 : $\texttt{MINE} := \{\texttt{M}\}$
act2 : $\texttt{pump}(\texttt{M}) := \texttt{OFF}$
act3 : $\texttt{bell}(\texttt{M}) := \texttt{FALSE}$
act4 : $\texttt{methane}(\texttt{M}) := \texttt{FALSE}$
act5 : $\texttt{waterLevel}(\texttt{M}) := \texttt{LOW}$
end
END

Appendix C

Event-B metamodel : `simpleeventb.ecore`

This annex presents the complete metamodel used to represent an Event-B machine and its traceability links with a KAOS model. The first section describe the Event-B elements hierarchy and the traceability links hierarchy belonging to a project. The second section presents the Event-B machine and context anatomy and the third section describe the traceability links associated with those elements.

C.1 Metamodel elements hierarchy

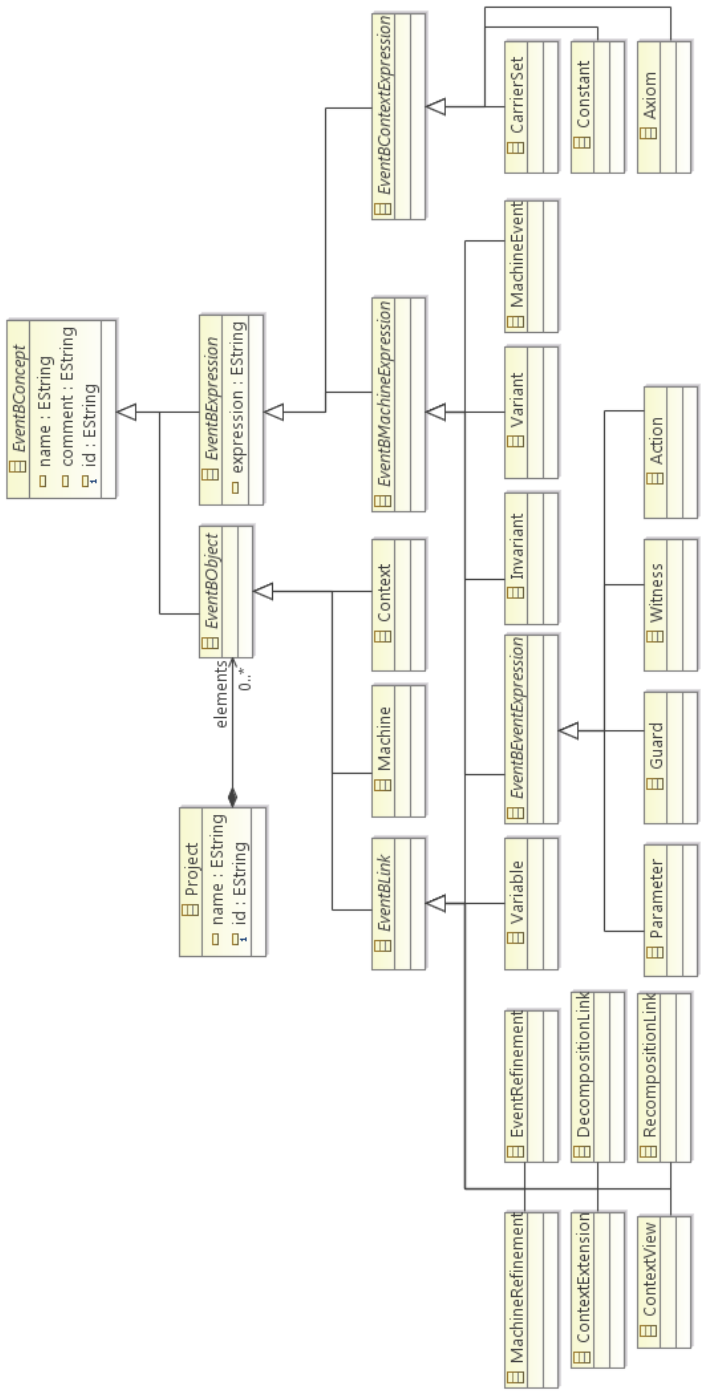

Figure C.1: Event-B concepts hierarchy

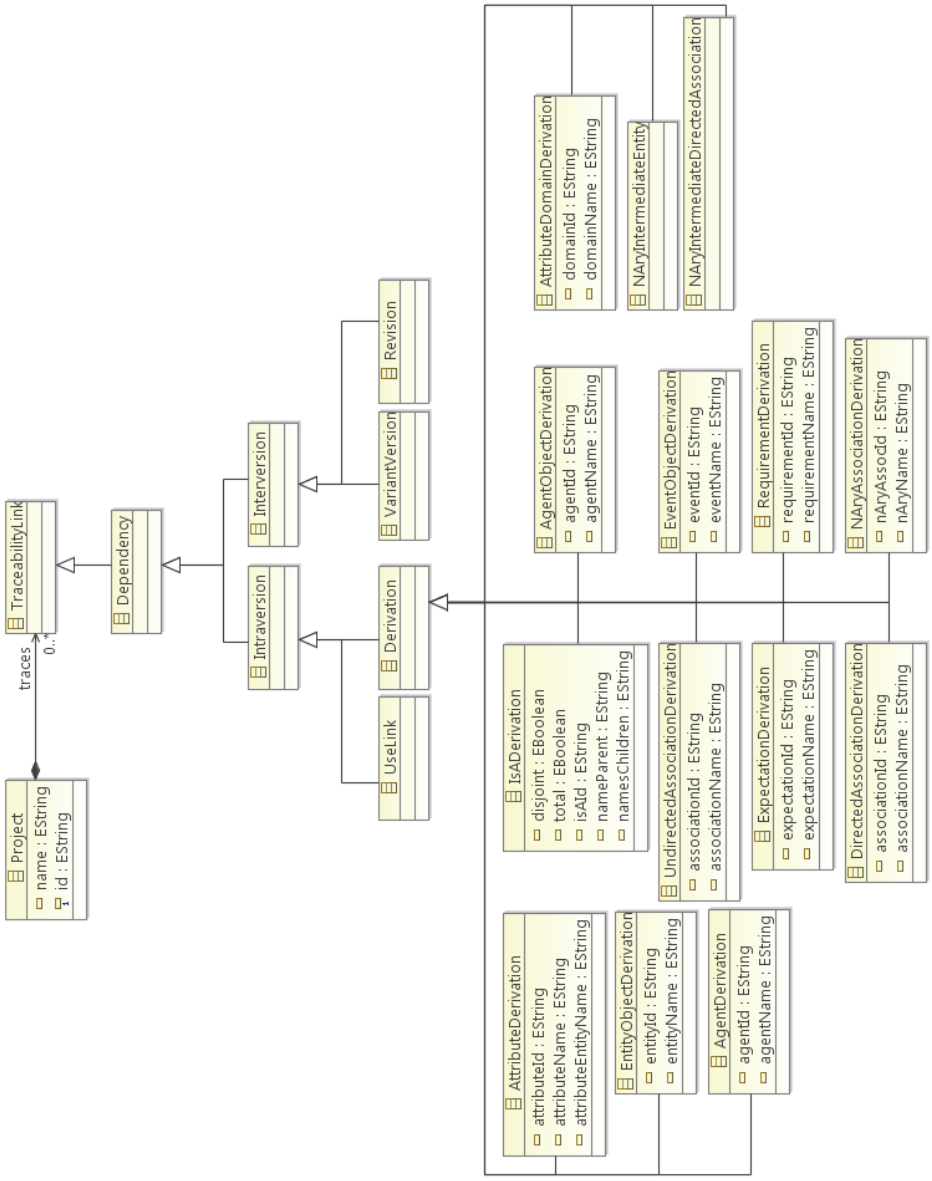

Figure C.2: Traceability links hierarchy

C.2 Event-B machine and context

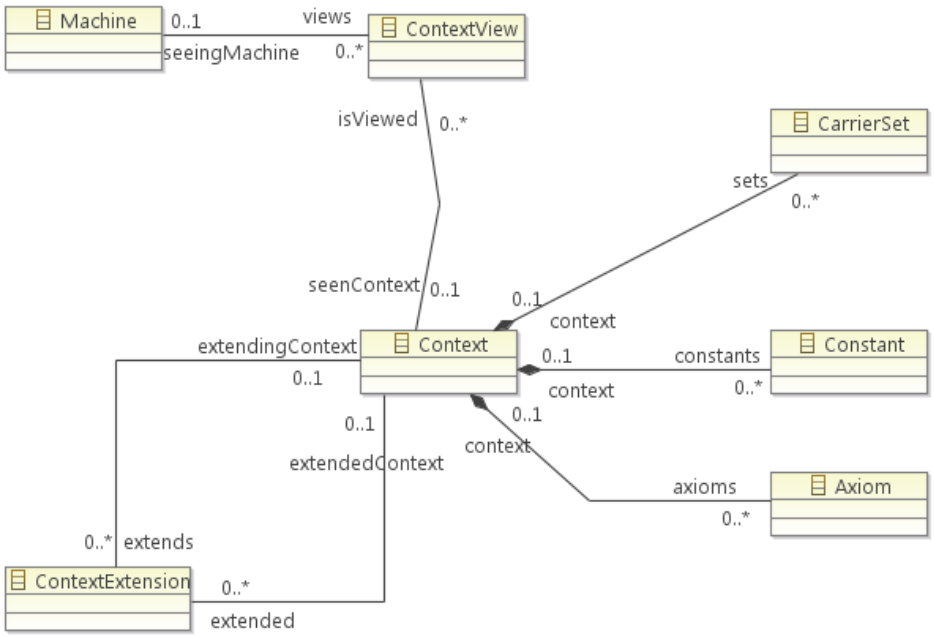

Figure C.3: Event-B context anatomy

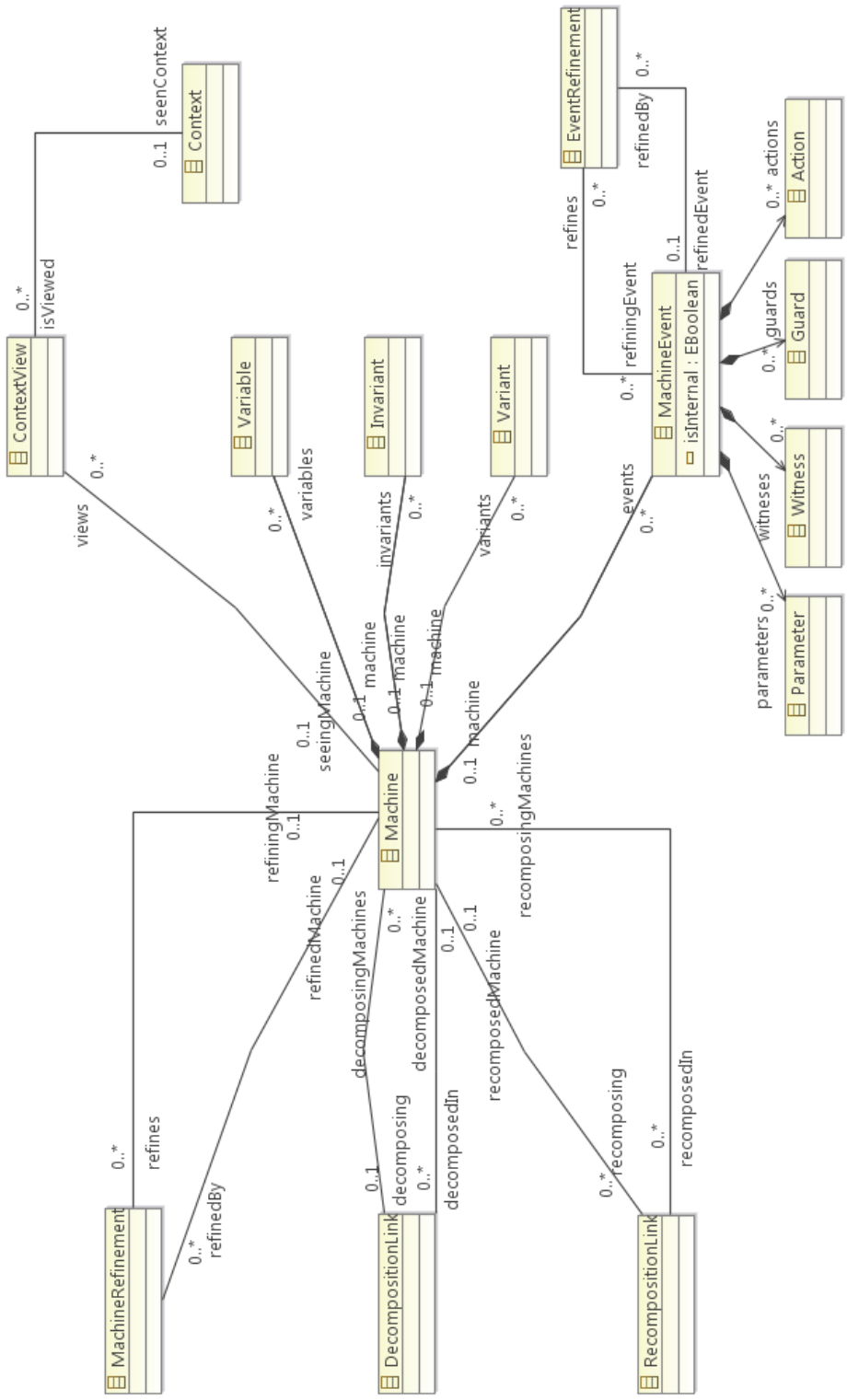

Figure C.4: Event-B machine anatomy

C.3 Traceability links

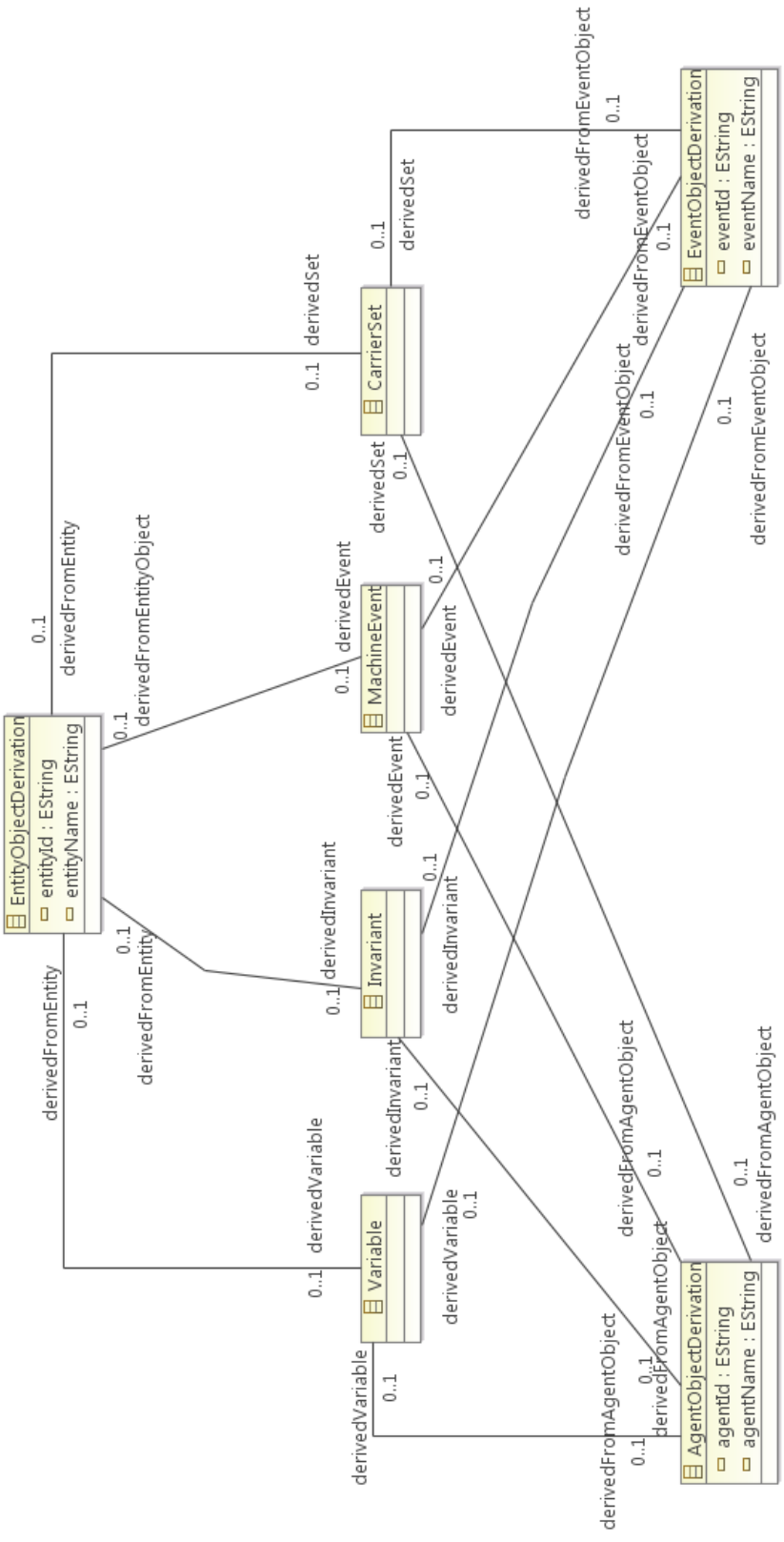

Figure C.5: Traceability between KAOS abstract objects and Event-B

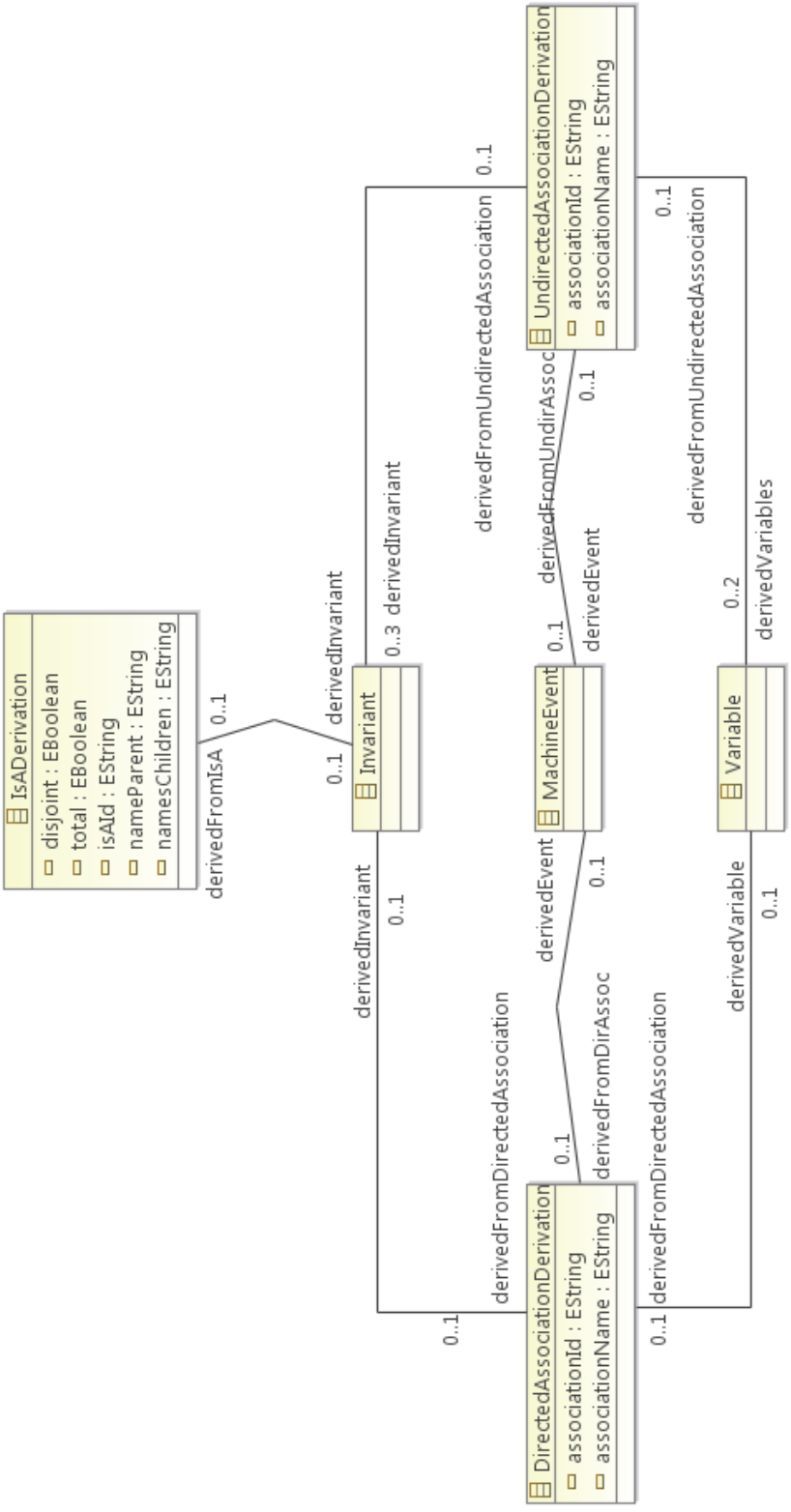

Figure C.6: Traceability between KAOS associations and Event-B

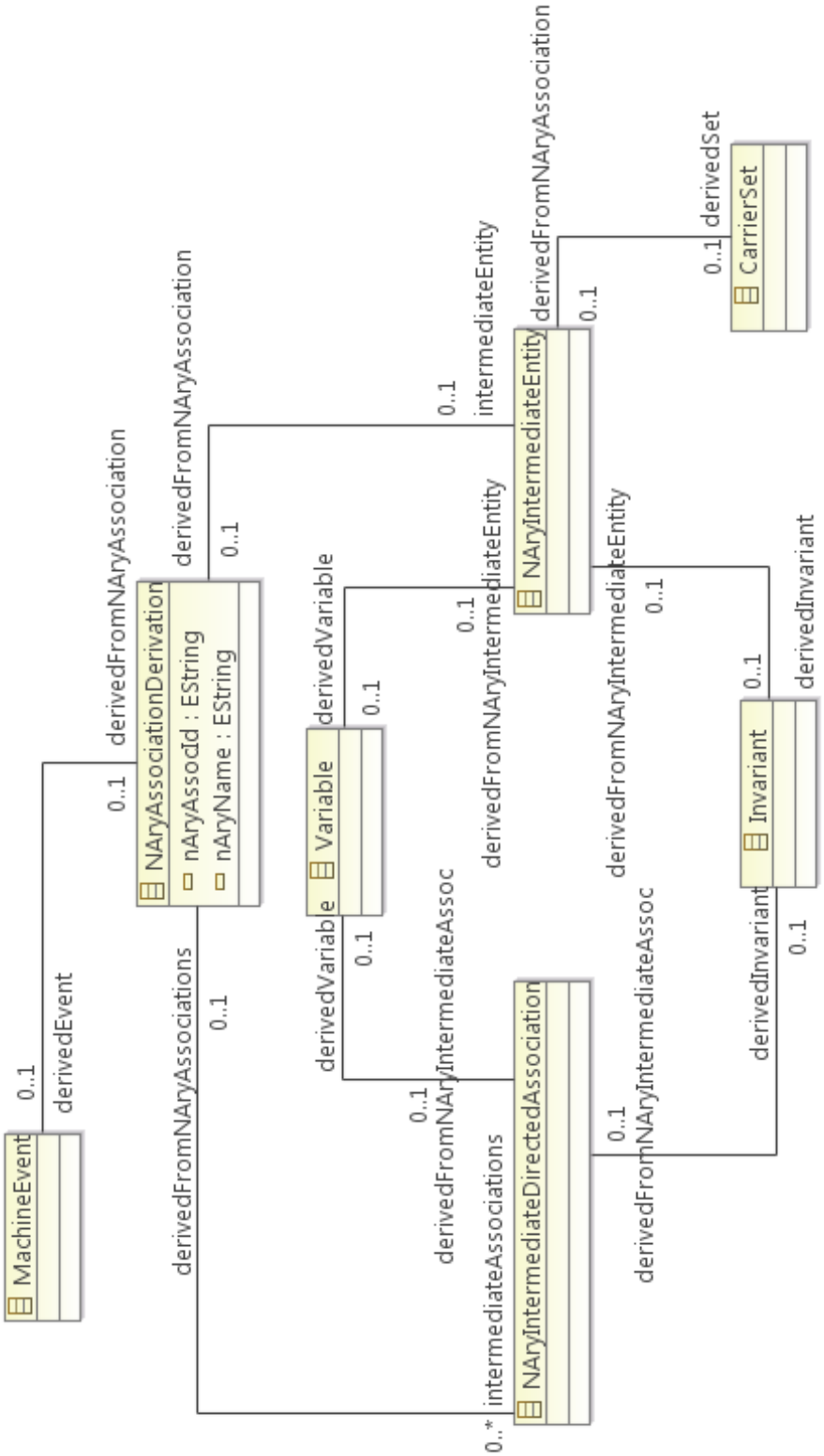

Figure C.7: Traceability between KAOS N-Ary associations and Event-B

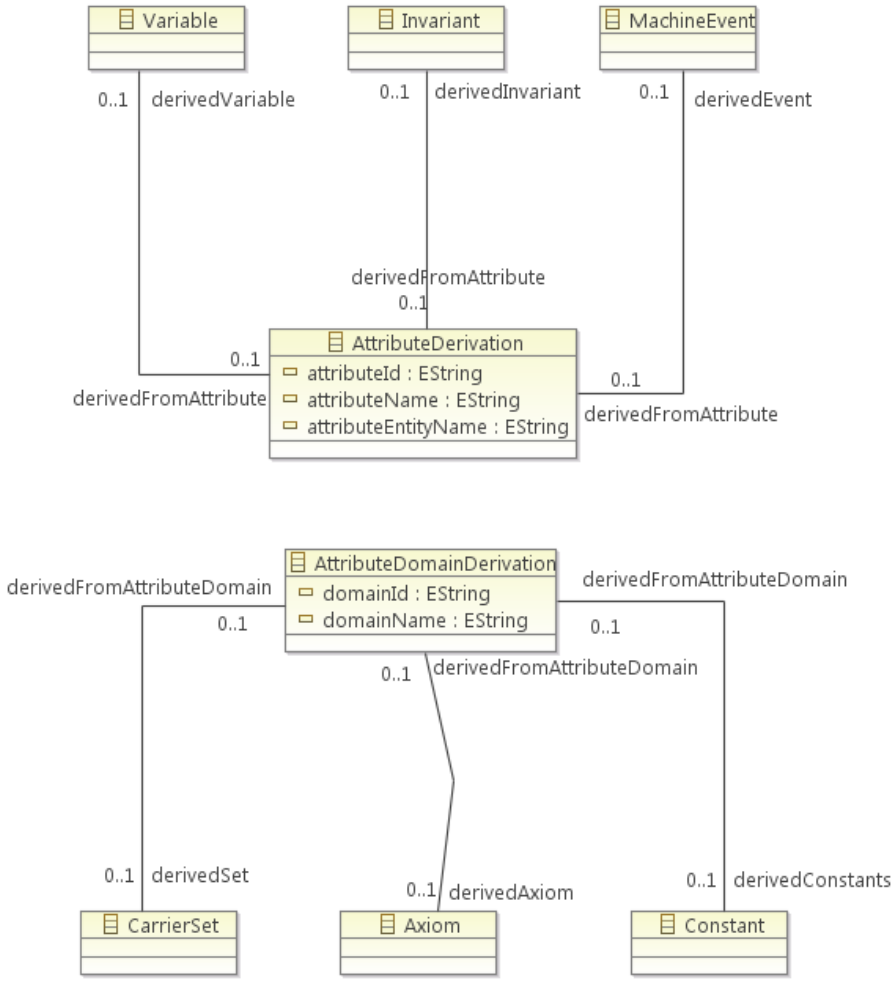

Figure C.8: Traceability between KAOS attributes and Event-B

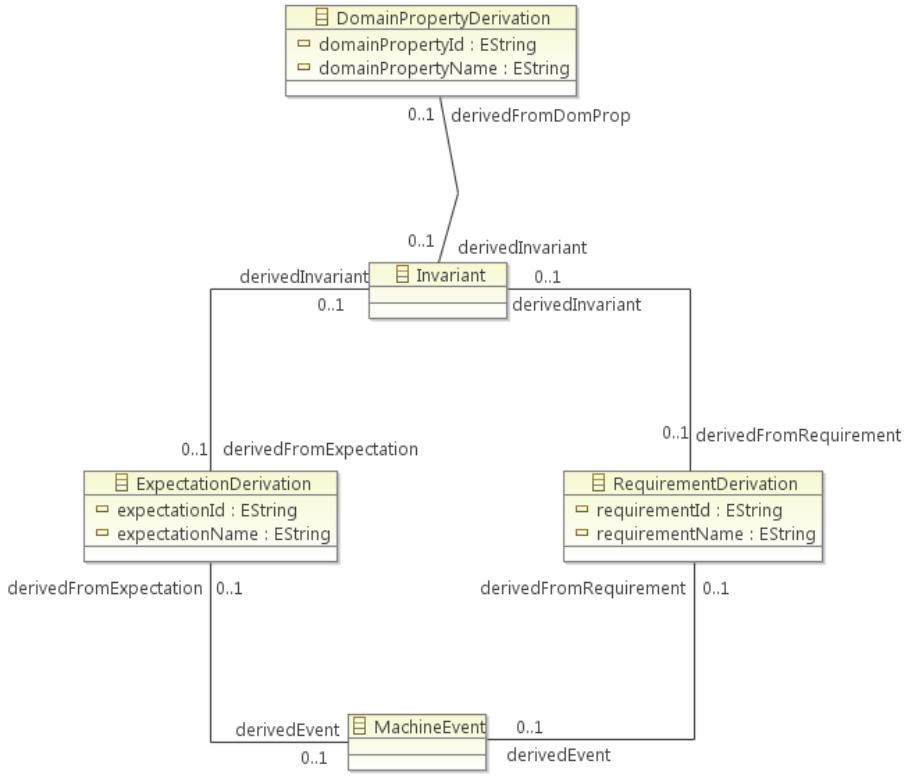

Figure C.9: Traceability between KAOS requirements, expectations and domain properties and Event-B

Figure C.10: Traceability between KAOS agents and Event-B

Appendix D

ATL transformation : `KAOS2EventB.atl`

This annex presents the ATL transformation used to process the initial transformation of the KAOS object model to the initial Event-B machine and context. It also create a machine per agent refining and decomposing the initial machine. See chapter 2 to have more details about the method.

The input model is a KAOS model serialized in a XMI [1] file with Objectiver [Respect-IT, 300]. Tools to connect to Objectiver and get a XMI version of the model are part of the FAUST project (`http://faust.cetic.be/`) and can be found at `http://sourceforge.net/projects/faust/`.

The output format of this transformation is a simplified Event-B model, also serialized in a XMI file, that has to be completed by the analyst and the traceability links with the KAOS model where it came from. The metamodel of the output model is described in annex C.

[1] XML Metadata Interchange (specification available at `http://www.omg.org/technology/documents/formal/xmi.htm`)

Listing D.1: ATL transformation code from operations to use cases

```
1  -- @atlcompiler atl2006
2  -- @path SIMPLEEVENTB=/be.cetic.kaos2eventb/metamodel/eventb/simpleeventb.ecore
3  -- @path KAOS=/be.cetic.objectiver.model/src/open/kaos.ecore
4
5  module KAOS2EventB;
6  create OUT : SIMPLEEVENTB from IN : KAOS;
7
8  helper def: project : SIMPLEEVENTB!Project = OclUndefined;
9  helper def: generalMachine : SIMPLEEVENTB!Machine = OclUndefined;
10 helper def: dataContext : SIMPLEEVENTB!Context = OclUndefined;
11 helper def: initialDecomposition : SIMPLEEVENTB!DecompositionLink = OclUndefined;
12
13 helper context KAOS!Link def : entityName : String =
14         if self.role.oclIsUndefined() then
15                 self.linksTo.name.toLower().replaceAll(' ','_')
16         else
17                 self.role.replaceAll(' ','_')
18         endif;
19
20 -- Rule called at the initialisation phase to create a project and it's
21 -- general machine, general context and traceability links collection.
22 entrypoint rule Project() {
23         to
24                 machine : SIMPLEEVENTB!Machine (
25                         id <- 'InitialMachine',
```

```
26                  name <- 'InitialMachine',
27                  comment <- 'The most general machine of the model \n'+
28                                          'created from a KAOS model',
29                  variables <- Set{},
30                  invariants <- Set{},
31                  variants <- Set{},
32                  events <- Set{},
33                  refines <- Set{},
34                  refinedBy <- Set{},
35                  views <- Set{ctxView},
36                  decomposedIn <- Set{},
37                  recomposedIn <- Set{}),
38
39              ctx : SIMPLEEVENTB!Context(
40                  id <- 'InitialContext',
41                  name <- 'DataContext',
42                  comment <- 'Data context for model data manipulation \n'+
43                                          'created from a KAOS object model',
44                  isViewed <- Set{ctxView},
45                  sets <- Set{},
46                  constants <- Set{},
47                  axioms <- Set{},
48                  extends <- Set{},
49                  extended <- Set{}),
50
51              ctxView : SIMPLEEVENTB!ContextView(
```

```
52                          seenContext <- ctx,
53                          seeingMachine <- machine),
54
55                  prj : SIMPLEEVENTB!Project (
56                          id <- 'EventBPrj',
57                          name <- 'Generated Event-B project',
58                          elements <- Set{machine,ctx,decompo,ctxView},
59                          traces <- Set{}
60                  ),
61
62                  decompo : SIMPLEEVENTB!DecompositionLink (
63                          id <- 'InitialDecomposition' ,
64                          name <- 'InitialDecomposition' ,
65                          comment <- 'Decomposition of the initial machine',
66                          decomposedMachine <- machine,
67                          decomposingMachines <- Set{}
68                  )
69          do {
70                  thisModule.project <- prj;
71                  thisModule.generalMachine <- machine;
72                  thisModule.dataContext <- ctx;
73                  thisModule.initialDecomposition <- decompo;
74          }
75  }
76
77  -- Rules concerning the KAOS Object Model :
```

```
78  -- ------------------------------------------
79
80  -- Entity is translated to a set in the initial context, a variable
81  -- in the initial machine, an updat envent in the initial machine and
82  -- an invariant in the initial machine.
83  -- The attributes of the entity are processed by this rule too by calling
84  -- the createAttributeRule lazy rule.
85  rule EntityRule {
86          from
87                  entity : KAOS!Entity
88          to
89                  set : SIMPLEEVENTB!CarrierSet(
90                          id <- 'ObjModel_'+entity.name+'_SET',
91                          name <- entity.name.replaceAll(' ','_').toUpper()+'_SET',
92                          comment<- 'Derived from Entity '+entity.name+' in KAOS Object Model
                                .',
93                          expression <- entity.name.replaceAll(' ','_').toUpper()+'_SET',
94                          context <- self.dataContext,
95                          derivedFromEntity <- link),
96
97                  variable : SIMPLEEVENTB!Variable(
98                          id <- 'ObjModel_'+entity.name+'_VARIABLE',
99                          name <- entity.name.replaceAll(' ','_').toUpper(),
100                         comment<- 'Derived from Entity '+entity.name+' in KAOS Object Model
                                .',
101                         expression <- entity.name.replaceAll(' ','_').toUpper(),
```

```
102             machine <- self.generalMachine,
103             derivedFromEntity <- link),
104
105         invariant : SIMPLEEVENTB!Invariant(
106             id <- 'ObjModel_'+entity.name+'_INVARIANT',
107             name <- entity.name.replaceAll(' ','_')+'_Type',
108             comment<- 'Derived from Entity '+entity.name+' in KAOS Object Model
                    .',
109             expression <- entity.name.replaceAll(' ','_').toUpper()+
110                     ' : POW('+entity.name.replaceAll(' ','_').toUpper()+'_SET'+
                        ')',
111             machine <- self.generalMachine,
112             derivedFromEntity <- link),
113
114         link : SIMPLEEVENTB!EntityObjectDerivation (
115             entityId <- entity.id,
116             entityName <- entity.name,
117             derivedVariable <- variable,
118             derivedInvariant <- invariant,
119             derivedEvent <- evt,
120             derivedSet <- set),
121
122         evt : SIMPLEEVENTB!MachineEvent(
123             id <- 'ObjModel_'+entity.name+'_UPDATE',
124             name <- entity.name.replaceAll(' ','_')+'_update',
125             comment<- 'Derived from Entity '+entity.name+' in KAOS Object Model
```

```
                                .',
126                             expression <- entity.name.replaceAll(' ','_')+'_update',
127                             parameters <- Set{},
128                             guards <- Set{},
129                             witneses <- Set{},
130                             actions <- Set{},
131                             refinedBy <- Set{},
132                             refines <- Set{},
133                             isInternal <- false,
134                             machine <- self.generalMachine,
135                             derivedFromEntityObject <- link)
136
137         do{
138                 self.dataContext.sets <- self.dataContext.sets.including(set);
139                 self.generalMachine.variables <- self.generalMachine.variables.including(
                        variable);
140                 self.generalMachine.invariants <- self.generalMachine.invariants.including(
                        invariant);
141                 self.generalMachine.events <- self.generalMachine.events.including(evt);
142                 self.project.traces <- self.project.traces.including(link);
143 --              self.project.elements <- self.project.elements.including(set);
144 --              self.project.elements <- self.project.elements.including(variable);
145 --              self.project.elements <- self.project.elements.including(invariant);
146 --              self.project.elements <- self.project.elements.including(evt);
147                 for(attribute in entity.attributes){
148                         thisModule.createAttribute(attribute,entity);
```

```
149                 }
150         }
151 }
152
153 -- Lazy rule called by EntityRule to create all the Attributes of an Entity.
154 rule createAttribute(attribute : KAOS!Attribute, entity : KAOS!Entity) {
155         to
156                 variable : SIMPLEEVENTB!Variable (
157                         id <- attribute.name.replaceAll(' ','_'),
158                         name <- attribute.name.replaceAll(' ','_'),
159                         comment <- 'Derived from Attribute '+attribute.name+
160                                         ' in KAOS Object Model.'+
161                                         ' -> '+attribute.domain,
162                         expression <- attribute.name.replaceAll(' ','_'),
163                         derivedFromAttribute <- link
164                 ),
165
166                 invariant : SIMPLEEVENTB!Invariant(
167                         id <- attribute.name.replaceAll(' ','_'),
168                         name <- attribute.name.replaceAll(' ','_'),
169                         comment <- 'Derived from Attribute '+attribute.name+
170                                         ' in KAOS Object Model : '+
171                                         entity.name.replaceAll(' ','_').toUpper()+
172                                         ' -> '+attribute.domain,
173                         expression <- attribute.name.replaceAll(' ','_')+' : '+
174                                                 entity.name.replaceAll(' ','_').toUpper()
```

```
175                                                    +
                                                ' --> '+attribute.domain,
176                 derivedFromAttribute <- link
177             ),
178
179             evt : SIMPLEEVENTB!MachineEvent(
180                 id <- 'ObjModel_'+entity.name+'.'+attribute.name+'_UPDATE',
181                 name <- (entity.name+'.'+attribute.name).replaceAll(' ','_')+'
                        _update',
182                 comment<- 'Derived from Entity '+entity.name+'.'+attribute.name+
183                                 ' in KAOS Object Model.',
184                 expression <- entity.name.replaceAll(' ','_')+'_update',
185                 parameters <- Set{},
186                 guards <- Set{},
187                 witneses <- Set{},
188                 actions <- Set{},
189                 refinedBy <- Set{},
190                 refines <- Set{},
191                 isInternal <- false,
192                 machine <- self.generalMachine,
193                 derivedFromEntityObject <- link),
194
195             link : SIMPLEEVENTB!AttributeDerivation(
196                 derivedVariable <- variable,
197                 derivedInvariant <- invariant,
198                 derivedUpdateEvent <- evt,
```

```
199                         attributeId <- entity.id+'.'+attribute.name,
200                         attributeName <- attribute.name,
201                         attributeEntityName <- entity.name
202                 )
203         do{
204                 self.generalMachine.variables <- self.generalMachine.variables.including(
                        variable);
205                 self.generalMachine.invariants <- self.generalMachine.invariants.including(
                        invariant);
206                 self.generalMachine.events <- self.generalMachine.events.including(evt);
207                 self.project.traces <- self.project.traces.including(link);
208 --              self.project.elements <- self.project.elements.including(variable);
209 --              self.project.elements <- self.project.elements.including(invariant);
210 --              self.project.elements <- self.project.elements.including(evt);
211         }
212 }
213
214 -- Attribute domain in translated in a set and an axiom in the
215 -- initial context.
216 --rule AttributeDomainRule {
217 --      AttributeDomain not implemented in the actual KAOS MM
218 --}
219
220 -- A directed association is translated into a variable, an
221 -- invariant and an update event in the initial machine.
222 --rule DirectedAssociationRule{
```

```
223 --        DirectedAssociation not implemented in the actual KAOS MM
224 --}
225
226 -- An undirected binary association is translated into 2 variables
227 -- and 2 invariants corresponding to 2 directed associations
228 -- and one invariant corresponding to the link between the two
229 -- directed associations and an update event in the initial machine.
230 rule BinaryUndirectedAssociationRule{
231         from association : KAOS!Relationship (
232                 -- Process only binary associations
233                 association.links->size() = 2
234         )
235         using {
236                 firstObj : KAOS!Link =
237                         association.links->asSequence()->first();
238                 secondObj : KAOS!Link =
239                         association.links->select( filteredOut | filteredOut <> firstObj )
240                         ->asSequence()->first();
241                 A : String = firstObj.linksTo.name.replaceAll(' ','_').toUpper();
242                 B : String = secondObj.linksTo.name.replaceAll(' ','_').toUpper();
243                 AtoB : String = firstObj.linksTo.name.replaceAll(' ','_')+
244                                                 '_TO_'+secondObj.linksTo.name.replaceAll('
                                                     ','_');
245                 BtoA : String = secondObj.linksTo.name.replaceAll(' ','_')+
246                                                 '_TO_'+firstObj.linksTo.name.replaceAll(' '
                                                     ,'_') ;
```

```
247
248         }
249         to
250                 -- linking 2 directed associations
251                 invariant : SIMPLEEVENTB!Invariant(
252                         id <- association.name.replaceAll(' ','_')+'_undir_assocition_inv',
253                         name <- association.name.replaceAll(' ','_')+'_undir_assocition_inv
                                ',
254                         comment <- 'Derived from undirected association '+association.name+
255                                         ' in KAOS Object Model.',
256                         expression <- '(! a.'+A+' =>( ! b.'+B+' => ( '+
257                                         AtoB+'(a) = b <=> '+BtoA+'(b) = a'
258                                         +' ) ))',
259                         derivedFromUndirectedAssociation <- trace
260                 ),
261
262                 -- update event
263                 evt : SIMPLEEVENTB!MachineEvent(
264                         id <- 'ObjModel_'+association.name+'_UPDATE',
265                         name <- (association.name).replaceAll(' ','_')+'_update',
266                         comment<- 'Derived from undirected association '+association.name+
267                                         ' in KAOS Object Model.',
268                         expression <- association.name.replaceAll(' ','_')+'_update',
269                         parameters <- Set{},
270                         guards <- Set{},
271                         witneses <- Set{},
```

```
272                     actions <- Set{},
273                     refinedBy <- Set{},
274                     refines <- Set{},
275                     isInternal <- false,
276                     machine <- self.generalMachine,
277                     derivedFromUndirAssoc <- trace),
278
279             -- link
280             trace : SIMPLEEVENTB!UndirectedAssociationDerivation(
281                     associationId <- association.id,
282                     associationName <- association.name,
283                     derivedInvariant <- Set{invariant},
284                     derivedEvent <- evt,
285                     derivedVariables <- Set{}
286             )
287
288     do{
289             thisModule.createDirectedAssociationFromUndirected(firstObj,firstObj.
                linksTo,
290                     secondObj.linksTo,trace,association);
291             thisModule.createDirectedAssociationFromUndirected(secondObj,secondObj.
                linksTo,
292                     firstObj.linksTo,trace,association);
293             self.generalMachine.invariants <- self.generalMachine.invariants.including(
                invariant);
294             self.generalMachine.events <- self.generalMachine.events.including(evt);
```

```
295                 self.project.traces <- self.project.traces.including(trace);
296 --                self.project.elements <- self.project.elements.including(invariant);
297 --                self.project.elements <- self.project.elements.including(evt);
298         }
299 }
300 -- Called by BinaryUndirectedAssociationRule
301 rule createDirectedAssociationFromUndirected(link : KAOS!Link,
302                                                source : KAOS!AbstractObject,
303                                                destination : KAOS!
                                                    AbstractObject,
304                                                trace : SIMPLEEVENTB!
                                                    UndirectedAssociationDerivation
                                                    ,
305                                                association : KAOS!Relationship)
                                                    {
306         to
307                 variable : SIMPLEEVENTB!Variable(
308                         id <- link.entityName+'_undir_assocition_'+
309                                 association.name.replaceAll(' ','_')+'_var',
310                         name <- link.entityName+'_undir_assocition_'+
311                                 association.name.replaceAll(' ','_')+'_var',
312                         comment <- 'Derived from undirected association '+association.name+
313                                 ', link '+link.entityName+' in KAOS Object Model.
                                    ',
314                         expression <- source.name.replaceAll(' ','_')+'_TO_'+
315                                 destination.name.replaceAll(' ','_'),
```

```
316                     machine <- self.generalMachine,
317                     derivedFromUndirectedAssociation <- trace
318             ),
319
320             invariant : SIMPLEEVENTB!Invariant(
321                     id <- link.entityName+'_undir_assocition_'+
322                             association.name.replaceAll(' ','_')+'_inv',
323                     name <- link.entityName+'_undir_assocition_'+
324                             association.name.replaceAll(' ','_')+'_inv',
325                     comment <- 'Derived from undirected association '+association.name+
326                                     ', link '+link.entityName+' in KAOS Object Model.
                                        ',
327                     -- Could be improved by using comparisons on link.multiplicity
328                     -- (typed as EString)
329                     expression <- 'TO DO',
330                     derivedFromAttribute <- link
331             )
332     do{
333             self.generalMachine.variables <- self.generalMachine.variables.including(
                    variable);
334             self.generalMachine.invariants <- self.generalMachine.invariants.including(
                    invariant);
335 --          self.project.elements <- self.project.elements.including(variable);
336 --          self.project.elements <- self.project.elements.including(invariant);
337             trace.derivedInvariant <- trace.derivedInvariant.including(invariant);
338             trace.derivedVariables <- trace.derivedVariables.including(variable);
```

```
339         }
340 }
341
342 -- A N-Ary association is transflated into (Intermediate entity part)
343 -- a set in the initial context, a variable in the initial machine,
344 -- an updat envent in the initial machine, an invariant in the
345 -- initial machine, (Directed associations part) n variables and n
346 -- invariants in the initial machine.
347 rule NAryUndirectedAssociationRule{
348         from association : KAOS!Relationship (
349                 -- Process only binary associations
350                 association.links->size() > 2
351         )
352         to
353                 -- intermediate entity
354                 entitySet : SIMPLEEVENTB!CarrierSet(
355                         id <- 'ObjModel_'+association.name+'_SET',
356                         name <- association.name.replaceAll(' ','_').toUpper()+'_SET',
357                         comment<- 'Derived from N-Ary association '+association.name+
358                                         ' in KAOS Object Model.',
359                         expression <- association.name.replaceAll(' ','_').toUpper()+'_SET'
                                ,
360                         context <- self.dataContext,
361                         derivedFromNAryAssociation <- intermediateEntity),
362
363                 entityVariable : SIMPLEEVENTB!Variable(
```

```
364                 id <- 'ObjModel_'+association.name+'_VARIABLE',
365                 name <- association.name.replaceAll(' ','_').toUpper(),
366                 comment<- 'Derived from N-Ary association '+association.name+
367                                 ' in KAOS Object Model.',
368                 expression <- association.name.replaceAll(' ','_').toUpper(),
369                 machine <- self.generalMachine,
370                 derivedFromNAryIntermediateEntity <- intermediateEntity),
371
372         entityInvariant : SIMPLEEVENTB!Invariant(
373                 id <- 'ObjModel_'+association.name+'_INVARIANT',
374                 name <- association.name.replaceAll(' ','_')+'_Type',
375                 comment<- 'Derived from N-Ary association '+association.name+
376                                 ' in KAOS Object Model.',
377                 expression <- association.name.replaceAll(' ','_').toUpper()+
378                         ' : POW('+association.name.replaceAll(' ','_').toUpper()+'
                            _SET'+')',
379                 machine <- self.generalMachine,
380                 derivedFromNAryIntermediateEntity <- intermediateEntity),
381
382         -- update event
383         evt : SIMPLEEVENTB!MachineEvent(
384                 id <- 'ObjModel_'+association.name+'_UPDATE',
385                 name <- association.name.replaceAll(' ','_')+'_update',
386                 comment<- 'Derived from N-Ary association '+association.name+
387                                 ' in KAOS Object Model.',
388                 expression <- association.name.replaceAll(' ','_')+'_update',
```

```
389                         parameters <- Set{},
390                         guards <- Set{},
391                         witneses <- Set{},
392                         actions <- Set{},
393                         refinedBy <- Set{},
394                         refines <- Set{},
395                         machine <- self.generalMachine,
396                         isInternal <- false,
397                         derivedFromNAryAssociation <- trace),
398
399                 -- links
400                 trace : SIMPLEEVENTB!NAryAssociationDerivation(
401                         nAryAssocId <- association.id,
402                         nAryName <- association.name,
403                         intermediateAssociations <- Set{},
404                         intermediateEntity <- intermediateEntity
405                 ),
406
407                 intermediateEntity : SIMPLEEVENTB!NAryIntermediateEntity(
408                         derivedSet <- entitySet,
409                         derivedInvariant <- entityInvariant,
410                         derivedVariable <- entityVariable
411                 )
412
413         do{
414                 -- IntermediateDirectedAssociations
```

```
415                 for(lnk in association.links){
416                         thisModule.createNAryIntermediateDirectedAssociation(lnk,trace,
                                association);
417                 }
418
419                 self.generalMachine.variables <- self.generalMachine.variables.including(
                        entityVariable);
420                 self.generalMachine.invariants <- self.generalMachine.invariants.including(
                        entityInvariant);
421                 self.generalMachine.events <- self.generalMachine.events.including(evt);
422
423                 self.dataContext.sets <- self.dataContext.sets.including(entitySet);
424
425 --                  self.project.elements <- self.project.elements.including(entityVariable);
426 --                  self.project.elements <- self.project.elements.including(entityInvariant);
427 --                  self.project.elements <- self.project.elements.including(entitySet);
428 --                  self.project.elements <- self.project.elements.including(evt);
429
430                 self.project.traces <- self.project.traces.including(trace);
431                 self.project.traces <- self.project.traces.including(intermediateEntity);
432         }
433 }
434
435 -- Called by NAryUndirectedAssociationRule
436 rule createNAryIntermediateDirectedAssociation(link : KAOS!Link,
437                                                trace : SIMPLEEVENTB!
```

```
                                                    NAryAssociationDerivation,
438                                                 association : KAOS!Relationship){
439         to
440                 intermediateAssoc : SIMPLEEVENTB!NAryIntermediateDirectedAssociation(
441                         derivedFromNAryAssociations <- trace,
442                         derivedInvariant <- invariant,
443                         derivedVariable <- variable
444                 ),
445
446                 variable : SIMPLEEVENTB!Variable(
447                         id <- 'ObjModel_'+link.entityName+
448                                 '_nary_assoc'+association.name.replaceAll(' ','_')+'_var'
                                    ,
449                         name <- link.entityName+
450                                 '_nary_assoc'+association.name.replaceAll(' ','_')+'_var'
                                    ,
451                         comment<- 'Derived from link '+link.entityName+' of N-Ary
                            association '+
452                                         association.name+' in KAOS Object Model.',
453                         expression <- association.name.replaceAll(' ','_')+'_TO_'+
454                                             link.entityName.replaceAll(' ','_'),
455                         machine <- self.generalMachine,
456                         derivedFromNAryIntermediateAssoc <- intermediateAssoc
457                 ),
458
459                 invariant : SIMPLEEVENTB!Invariant(
```

```
460                         id <- 'ObjModel_'+link.entityName+
461                                 '_nary_assoc'+association.name.replaceAll(' ','_')+'_inv'
                                    ,
462                         name <- link.entityName+
463                                 '_nary_assoc'+association.name.replaceAll(' ','_')+'_inv'
                                    ,
464                         comment <- 'Derived from link '+link.entityName+' of N-Ary
                                    association '+
465                                         association.name+' in KAOS Object Model.',
466                         -- Could be improved by using comparisons on link.multiplicity
467                         -- (typed as EString)
468                         expression <- 'TO DO',
469                         derivedFromNAryIntermediateAssoc <- intermediateAssoc
470                 )
471
472         do{
473                 trace.intermediateAssociations <-
474                         trace.intermediateAssociations.including(intermediateAssoc);
475
476                 self.generalMachine.variables <- self.generalMachine.variables.including(
                    variable);
477                 self.generalMachine.invariants <- self.generalMachine.invariants.including(
                    invariant);
478
479 --              self.project.elements <- self.project.elements.including(variable);
480 --              self.project.elements <- self.project.elements.including(invariant);
```

```
481
482                     self.project.traces <- self.project.traces.including(intermediateAssoc);
483             }
484
485 }
486
487 -- IsA link between 2 entities is translated an invariant
488 -- according to the total and the disjoint creteria of the
489 -- IsA link.
490 --rule IsARule{
491 -- IsA not implemented in the actual KAOS MM
492 --}
493
494 -- Domain property defined in the goal model becomes an invariant
495 -- in the initial machine.
496 --rule DomainPropertyRule{
497 -- DomProp is not implemented in the actual KAOS MM
498 --}
499
500 -- An agent is translated into a machine refining the initial machine.
501 rule AgentRule{
502         from
503                 agent : KAOS!Agent
504         to
505                 refinement : SIMPLEEVENTB!MachineRefinement(
506                         id <- agent.name+'_MACHINE_refines_InitialMachine',
```

```
507                 name <- agent.name+'_refines_InitialMachine',
508                 comment <- 'Create from the KAOS Agent :'+agent.name,
509                 refinedMachine <- self.generalMachine,
510                 refiningMachine <- machine
511         ),
512
513         machine : SIMPLEEVENTB!Machine (
514                 id <- agent.name+'_MACHINE',
515                 name <- agent.name,
516                 comment <- 'Create from the KAOS Agent :'+agent.name,
517                 variables <- Set{},
518                 invariants <- Set{},
519                 variants <- Set{},
520                 events <- Set{},
521                 refines <- Set{refinement},
522                 refinedBy <- Set{},
523                 views <- Set{},
524                 decomposedIn <- Set{},
525                 recomposedIn <- Set{},
526                 derivedFromAgent <- link),
527
528         link : SIMPLEEVENTB!AgentDerivation(
529                 derivedMachine <- machine,
530                 agentId <- agent.id,
531                 agentName <- agent.name
532         )
```

```
533 do{
534         self.project.traces <- self.project.traces.including(link);
535         self.project.elements <- self.project.elements.including(machine);
536         self.project.elements <- self.project.elements.including(refinement);
537         self.initialDecomposition.decomposingMachines <-
538                 self.initialDecomposition.decomposingMachines.including(machine);
539         self.generalMachine.refinedBy <-
540                 self.generalMachine.refinedBy.including(refinement);
541     }
542 }
```

Bibliography

[Abrial, 2009a] Abrial, J.-R. (2009a). Event model decomposition. `http://deploy-eprints.ecs.soton.ac.uk/109/`.

[Abrial, 2009b] Abrial, J.-R. (2009b). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.

[Aziz et al., 2009] Aziz, B., Arenas, A., Bicarregui, J., Ponsard, C., and Massonet, P. (2009). From goal-oriented requirements to event-b specifications. In *First Nasa Formal Method Symposium*, pages 96–105.

[Ball, 2008] Ball, E. (2008). *An Incremental Process for the Development of Multi-agent Systems in Event-B*. PhD thesis, University of Southampton. `http://eprints.ecs.soton.ac.uk/16575/`.

[Butler, 2009] Butler, M. (2009). Decomposition structures for event-b. *Integrated Formal Methods iFM2009, Springer, LNCS*, 5423:20–38.

[Gervais et al., 2009] Gervais, F., Gnaho, C., Laleau, R., Matoussi, A., and Semmak, F. (2009). Tacos livrable l1.2 : Kaos extension with non-functional properties. `http://tacos.loria.fr/drupal/?q=node/74`. Projet TACOS : Trustworthy Assembling of Components: frOm requirements to Specification ANR-06-SETI-017 Janvier 2007 - D´ecembre 2009.

[Landtsheer, 2007a] Landtsheer, R. D. (2007a). Deriving event-based security policy from declarative security requirements.

[Landtsheer, 2007b] Landtsheer, R. D. (2007b). *Elaborating Complete and Consistent Requirements for Security-Critical Systems*. PhD thesis, Université Catholique de Louvain. `http://www.info.ucl.ac.be/~rdl/thesis/`.

[Letier, 2001] Letier, E. (2001). *Reasoning about Agents in Goal-Oriented Requirements Engineering*. PhD thesis, Université Catholique de Louvain.

[Matoussi, 2009] Matoussi, A. (2009). Expressing kaos goal models with event-b. LACL, Université Paris-Est.

[Matoussi et al., 2008] Matoussi, A., Gervais, F., and Laleau, R. (2008). A first attempt to express kaos refinement patterns with event b. In *Proc. of the Int. Conf. on ASM, B and Z (ABZ). Lecture Notes in Computer Science, Springer-Verlag*, pages 12–14. Springer.

[Matoussi et al., 2009] Matoussi, A., Laleau, R., and Petit, D. (2009). Bridging the gap between kaos requirements models and b specifications. Technical Report TR-LACL-2009-5, LACL (Laboratory of Algorithms, Complexity and Logic), University of Paris-Est (Paris 12).

[Métayer et al., 2005] Métayer, C., Abrial, J.-R., and Voisin, L. (2005). Rodin deliverable 3.2: Event-b language. `http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf`. `http://rodin-b-sharp.sourceforge.net`.

[Pascal and Silva, 2009] Pascal, C. and Silva, R. (2009). Event-b model decomposition: A-style vs. b-style.

[Respect-IT, 300] Respect-IT (v 3.0.0). Objectiver. `http://www.objectiver.com/`.

[REVER, 901] REVER (v 9.0.1). Db-main. `http://www.db-main.be`.

[RODIN, v 11] RODIN (v 1.1). Rodin platform. `http://www.event-b.org/`.

[Snook and Butler, 2006] Snook, C. and Butler, M. (2006). Uml-b: Formal modeling and design aided by uml. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122.

[van Lamsweerde, 2009] van Lamsweerde, A. (2009). *Requirements Engineering: From System Goals to UML Models to Software Specifications.* Wiley.

[yah Said et al., 2009] yah Said, M., Butler, M., and Snook, C. (2009). Language and tool support for class and state machine refinement in uml-b. In *FM2009 - 16th International Symposium on Formal Methods*, number LNCS 5, pages 579–595. Springer.