Bridging KAOS and Event B: Intermediate report

Xavier Devroey

December, 11 2009

Contents

1 Expressing KAOS Goal Models with Event-B: A. Matoussi 1
1.1 First phase 4
1.1.1 Milestone-driven refinement 4
1.1.2 Or-refinement 5
1.2 Second phase 5
1.2.1 Milestone-driven refinement 6
1.2.2 Or-refinement 7

2 KAOS Object model to Event-B Context and Machine 8
2.1 Object types and Attributes 8
2.2 Associations and Specializations 9

3 Decomposition of the initial model according to Agents 12
3.1 State-Based Decomposition 14
3.1.1 Example 15

A Decomposition according to Agents: Mine pump example 18

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.

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 "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

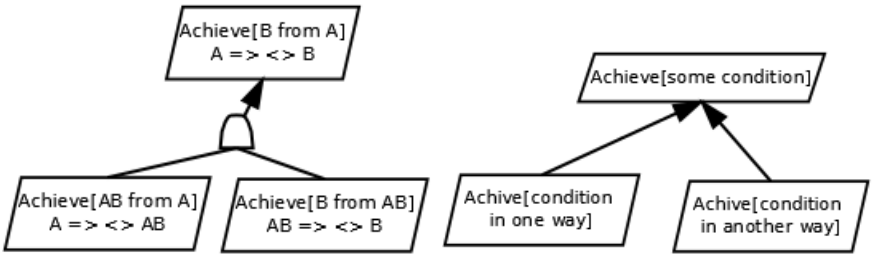

Figure 1: Milestone-driven refinement and Or-refinement

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.

The process in figure 2 has two phases: the first one creates an Event-B representation of the goal model. The initial 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" of 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.

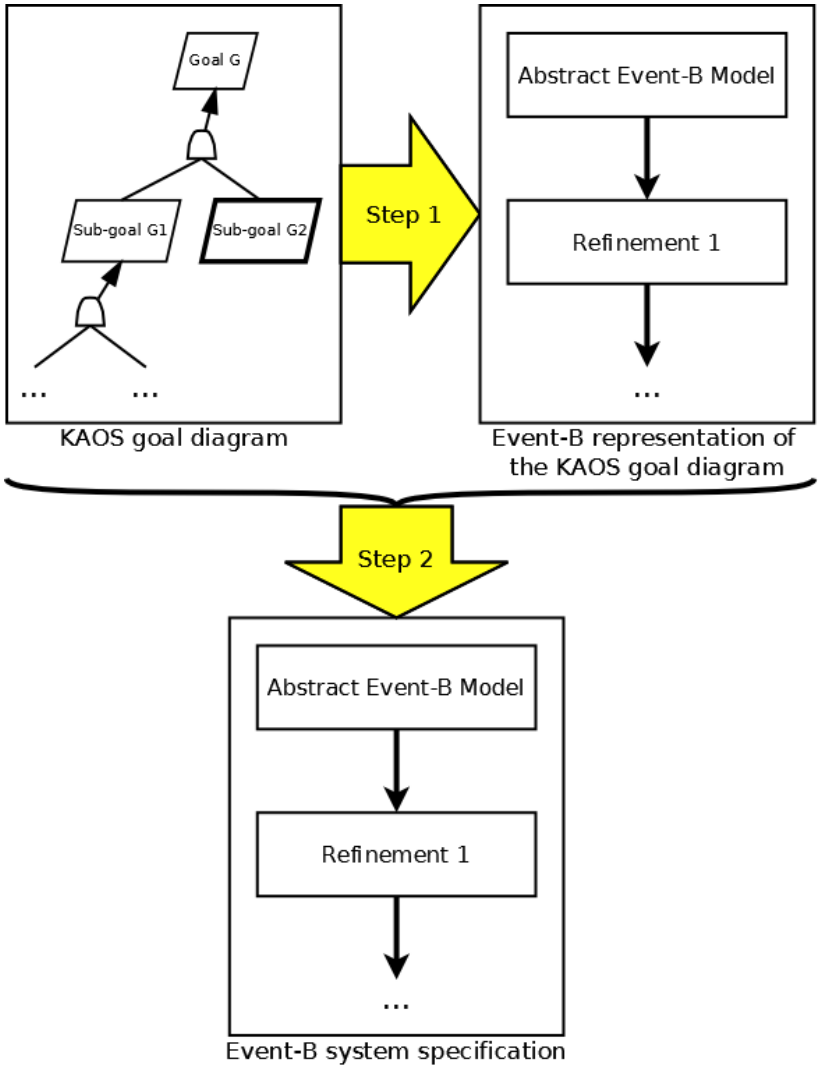

Figure 2: Expressing KAOS Goal Models with Event-B: process overview

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 could always be achieved.

Listing 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

1.1.1 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 will give a machine:

Listing 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 : $A \wedge AB$
end
Event *EvtG1* $\widehat{=}$
where
grd : TRUE
then
act : AB
end
Event *EvtG2* $\widehat{=}$
where
grd : TRUE
then
act : B
end
Event *EvtG* $\widehat{=}$
refines *EvtG*
where
grd : $AB \wedge B$
then
act : B
end
END

1.1.2 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:

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

1.2 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 fallowing 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 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.

1.2.1 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 `EvtOpG3` will refine the `EvtOpG` event of the parent machine. The refinement of goal G following the pattern described in figure 1 will give a machine:

Listing 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

1.2.2 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.

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, 2007]. 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.

2.1 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 1 gives the transformation rules for the different Multiplicities of an attribute of Object type `T`.

Table 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 Associations and Specializations

Associations may be directed or not and will be represented in the Machine by functions. Table 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
A set BtoA according to the rules in table 2
An additional invariant:
$$\forall x, y \cdot (x \in A \wedge y \in B) \Leftrightarrow (AtoB(x) = y \Leftrightarrow BtoA(y) = x)$$

As show in figure 3, 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 :

$$SON1 \cup SON2 = PARENT$$

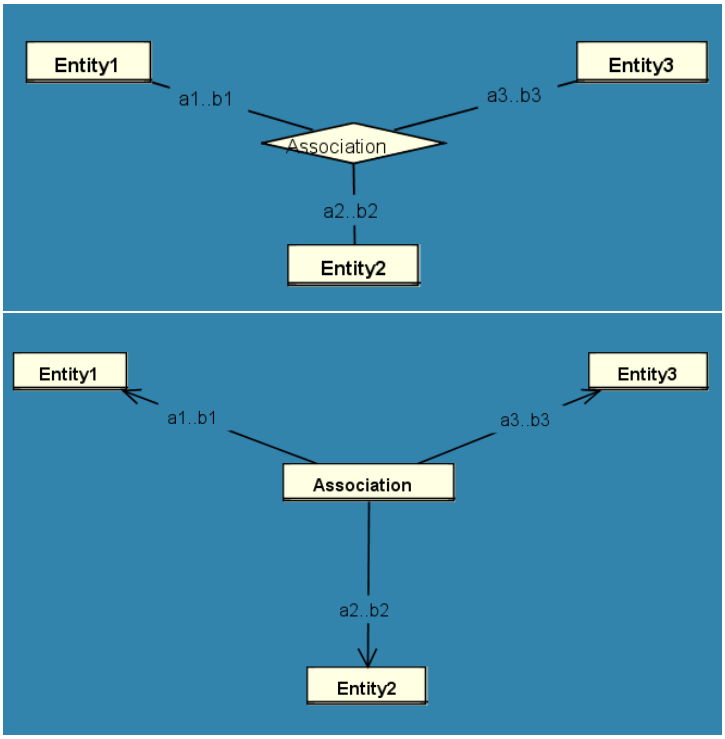

Figure 3: N-Ary Association are seen as an Entity with N directed Associations

Table 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`	$AtoB \in A \pfun B$
$0..* \rightarrow 1..1$	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`	$AtoB \in A \pinj B$
$0..1 \rightarrow 1..1$	Total injection to `B`	$AtoB \in A \inj B$
$0..1 \rightarrow 0..*$	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`	$AtoB \in A \psur B$
$1..* \rightarrow 1..1$	Total surjection to `B`	$AtoB \in A \surj B$
$1..* \rightarrow 0..*$	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`)	$AtoB \in A \pinj B \wedge \forall b \cdot (b \in B \Rightarrow (\exists a \cdot (a \in A \wedge (a \mapsto b) \in AtoB)))$
$1..1 \rightarrow 1..1$	Total bijection to `B`	$AtoB \in A \bij B$
$1..1 \rightarrow 0..*$	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	$AtoB \in A \rightarrow \mathbb{P}_1(B) \wedge \text{union}(\text{ran}(AtoB)) = B \wedge disjoint$

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, 2007,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 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 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 for 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 4 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.

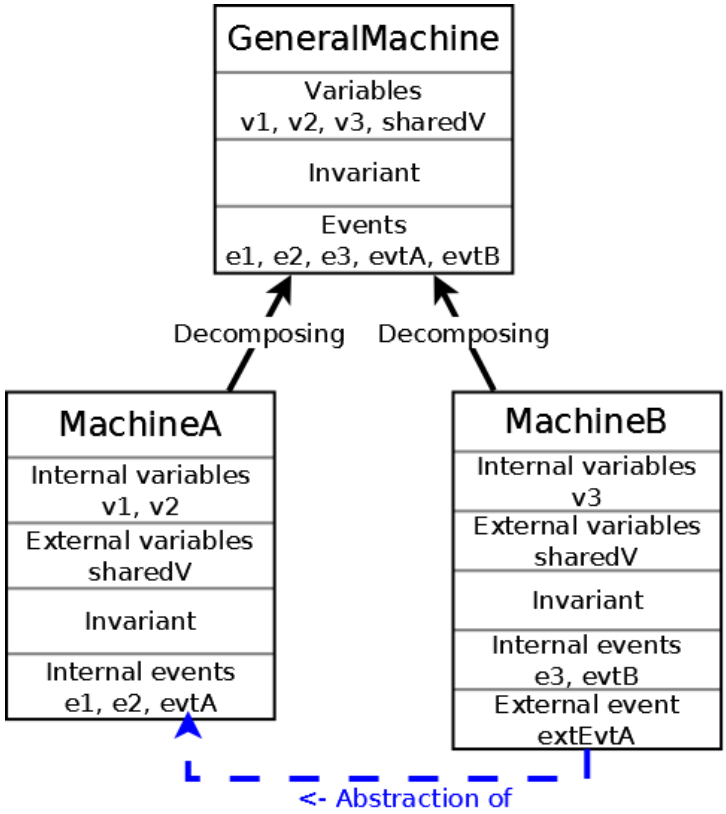

Figure 4: Decomposition of a general machine into two sub-machines

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.

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, 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, 2007]. 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 refinement of the general update event defined in the general machine. The re-composition of sub-machines will be simply 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 simply 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 could be simply 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.

3.1.1 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.

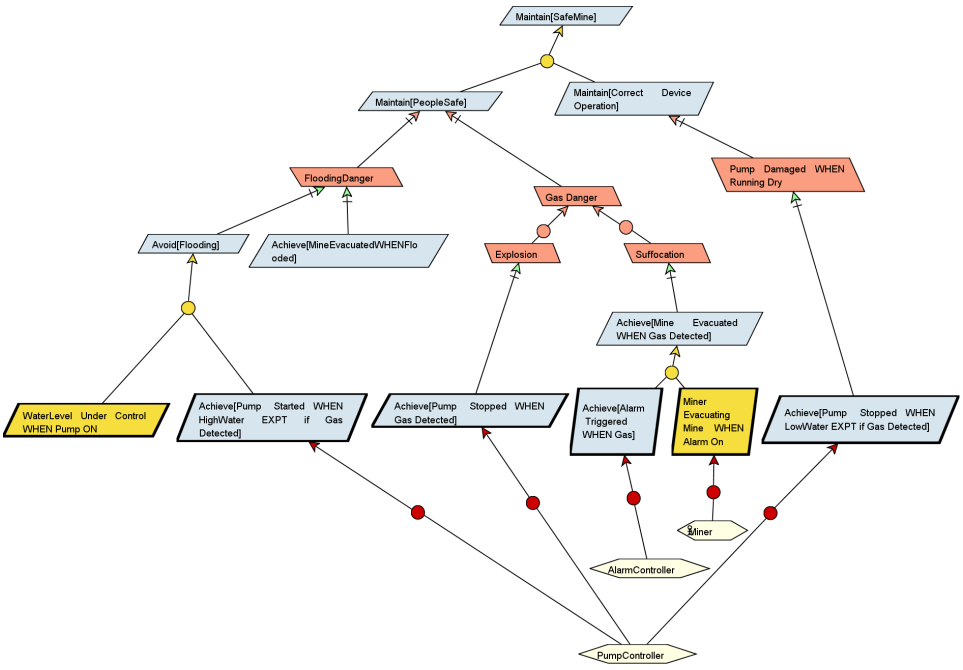

Figure 5: Mine pump goal model

Figure 6 presents the goal model and the different agents responsible for the requirements and expectations. Figure 6 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.

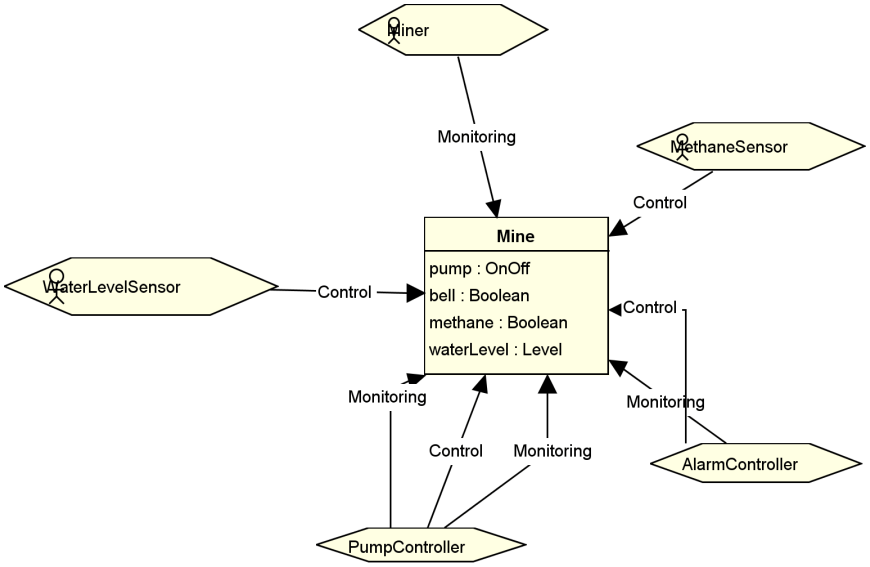

Figure 6: Mine pump agent model

By applying the procedure described in section 2, we get an initial Context in listing 5 and an initial machine in listing 6 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 6 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 A.

Listing 5: 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 6: Mine pump example: Initial machine

MACHINE MinePump
SEES MineContext
VARIABLES

```
    MINE, pump, bell, methane, waterLevel
INVARIANTS
    inv1 : MINE ∈ 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 7 shows the machine defined for the PumpController. This machine and all the other machines of this example can be found in annex A. The re-composed machine can also be found in listing 14 in annex A where the update methods have been replaced by their refinements in the different sub-machines.

Listing 7: 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* $\widehat{=}$
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 *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
Event *updateWaterLevel* $\widehat{=}$
External Event
extends *updateWaterLevel*
any
m
level
where
grd1 : m ∈ MINE
grd2 : level ∈ LEVEL
then
act1 : waterLevel(m) := level
end
END

A Decomposition according to Agents: Mine pump example

This annex present the complete machines of the mine pump example described in section 3.

Listing 8: 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 9: 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 ∈ MINE
        grd2 : status ∈ BOOL
    then
        act1 : bell(m) := status
    end
Event updateMethane ≙
    any
        m
        status
    where
        grd1 : m ∈ MINE
        grd2 : status ∈ BOOL
    then
        act1 : methane(m) := status
    end
Event updateWaterLevel ≙
    any
        m
        level
    where
        grd1 : m ∈ MINE
        grd2 : level ∈ LEVEL
    then
        act1 : waterLevel(m) := level
    end
Event addMine ≙
    when
        grd1 : MINE = ∅
    then
        act1 : MINE := {M}
        act2 : pump(M) := OFF
        act3 : bell(M) := FALSE
        act4 : methane(M) := FALSE
        act5 : waterLevel(M) := LOW
    end
END
```

Listing 10: 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* $\widehat{=}$
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 *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
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 11: 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 : $m \in MINE$
grd2 : $waterLevel(m) = HIGH$
with
level : level = MEDIUM
then
act1 : $waterLevel(m) := MEDIUM$
end
Event *medium_to_low* ≙
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* ≙
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* ≙
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 ≙
    External Event
extends updatePump
    any
        m
        status
    where
        grd1 : m ∈ MINE
        grd2 : status ∈ ONOFF
    then
        act1 : pump(m) := status
    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
END
```

Listing 12: 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 13: 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 : $m \in MINE$
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* ≙
External Event
extends *updateBell*
any
m
status
where
grd1 : m ∈ MINE
grd2 : status ∈ BOOL
then
act1 : bell(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 14: 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 $methane\_detected \mathrel{\widehat{=}}$
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 $methane\_leak \mathrel{\widehat{=}}$
Internal Event
refines *updateMethane*
any
m
where
grd1 : $m \in MINE$
with
status : status = TRUE
then
act1 : $methane(m) := TRUE$
end
Event $high\_water\_detected \mathrel{\widehat{=}}$
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 $low\_water\_detected \mathrel{\widehat{=}}$
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 $high\_to\_medium \widehat{=}$

Internal Event

refines $updateWaterLevel$

any

m

where

grd1 : $m \in MINE$

grd2 : $waterLevel(m) = HIGH$

with

level : $\texttt{level} = \texttt{MEDIUM}$

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 : $\texttt{level} = \texttt{LOW}$

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

References

[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, 2007] Landtsheer, R. D. (2007). *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.

[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`.

[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.