# From Goal-Oriented Requirements to Event-B Specifications

Benjamin Aziz, Alvaro E. Arenas, Juan Bicarregui
e-Science Centre, STFC Rutherford Appleton Laboratory
Oxford OX11 0QX, United Kingdom
{benjamin.aziz,alvaro.arenas,juan.bicarregui}@stfc.ac.uk
Christophe Ponsard, Philippe Massonet
Centre d'Excellence en Technologies de l'Information et de la Communication (CETIC)
B-6041 Charleroi, Belgium
{christophe.ponsard,philippe.massonet}@cetic.be

## Abstract

In goal-oriented requirements engineering methodologies, goals are structured into refinement trees from high-level system-wide goals down to fine-grained requirements assigned to specific software/hardware/human agents that can realise them. Functional goals assigned to software agents need to be operationalised into specification of services that the agent should provide to realise those requirements. In this paper, we propose an approach for operationalising requirements into specifications expressed in the Event-B formalism. Our approach has the benefit of aiding software designers by bridging the gap between declarative requirements and operational system specifications in a rigorous manner, enabling powerful correctness proofs and allowing further refinements down to the implementation level. Our solution is based on verifying that a consistent Event-B machine exhibits properties corresponding to requirements.

## 1 Introduction

Goal-driven approaches focus on why systems are constructed, providing the motivation and rationale for justifying software requirements. Examples of goal-oriented requirements methodologies include KAOS [24] and $i^*$/Tropos [10], among others. In these methodologies, a goal is an objective, which the system under consideration and its environment must achieve. Hence, goals are *operationalised* into specifications of operations to achieve them. For instance, the KAOS language supports the specification of operations defined as constraints on state transitions. The language relies on a temporal state-based logic, where a global clock generates ticks regularly, creating new states and transitions. These constraints, described as pre/post/trigger conditions, restrict the possible values in pairs of successive states.

One of the main aspects of goal-oriented requirements engineering methodologies is that they are best at refining goals and capturing intentions and expectations about the system-to-be. However, they lack the ability to extend this refinement process to the design specification level. Instead, they are usually limited to the definition of high-level declarative specifications of the system design. One would like to continue the refinement process though these high-level specifications down to levels of detail that facilitate the implementation of such specifications.

In this paper, we define an approach for operationalising goal-oriented requirements into Event-B specifications [4], using model-checking techniques to demonstrate that an Event-B machine is a model of the system requirements expressed as linear temporal logic formualae. The approach is general, however, we propose a few operationalisation patterns that assist designers in the derivation of an Event-B machine from the system requirements. This can be considered a small step towards achieving an automatic construction of these machines.

Combining goal-orientation and Event-B has several benefits. First, they have a common scope in that they target the modelling of the system as a whole. Second, they are complementary in that goals help in identifying key properties and reasoning on them while Event-B helps in designing rigorously a more operational system by introducing more and more design details using the refinement approach. Finally, at tool level, the benefit is mutual: requirements level tools[22] help in ensuring consistent/complete requirements and guide the elaboration of the initial Event-B specification. Event-B industrial-level tools can then be used to perform more powerful verification, especially automated proofs [3].

The rest of the paper is structured as follows. Section 2 gives an overview of goal-oriented requirements engineering and introduces an example of a safety-critical system. Section 3 gives an overview of the Event-B language. Section 4 presents our approach for defining a correct operationalisation of requirements into Event-B machines. Section 5 compares our work with related literature and finally, Section 6 concludes our work and highlights future research directions.

## 2    Goal-Oriented Requirements Engineering

Requirements engineering involves eliciting, analysing and specifying the requirements of a system. The precise understanding of these requirements serves as a foundation to assess and manage the subsequent development phases. Goal-oriented requirements engineering methodologies, such as KAOS [24] and $i^*$/Tropos [10], focus on justifying why a system is needed through the specification of its high-level goals. These goals then drive the requirements elaboration process, which results in the definition of domain-specific requirements that can be *implemented* by the system components under development.

Goals may be organised in an AND-refinement hierarchy [24], where higher-level goals are in general strategic and coarse-grained whereas lower-level goals are technical and fine-grained. In such hierarchies, AND-links (represented here by circles) relate a goal to a set of sub-goals possibly conjoined with domain properties or environment assumptions; this means that satisfying all the subgoals in the refinement is a sufficient condition in the domain for satisfying the goal. Goal refinement ends when every sub-goal is realisable by some individual component assigned to it in the software-to-be.

Formally, goals at all levels can be represented in real-time linear temporal logic, which in addition to the usual first order logic operators ($\land \lor \lnot \rightarrow \leftrightarrow$), it provides a number of temporal operators for the future: $\circ$ (next), $\square$ (always), $\diamond$ (eventually) and $\diamond_{\leq d}$ (bounded eventually). We do not consider past LTL in the scope of this paper. Furthermore, we write $P \Rightarrow Q$ to mean $\square(P \rightarrow Q)$.

### 2.1    Goals in Action: A Mine Sump

In order to demonstrate the goal-refinement methodology and introduce the requirements that will be operationalised, we consider the example of a mine sump [12]. In this system, water seeps into the sump from the mine and the level of water is kept within bounds by operating a pump. Additionally, a bell alarm must be immediately sounded if methane is detected in the sump and the pump must be shut down.

Figure 1 shows the goal model for such a mine sump system. The top goal of the system is to keep
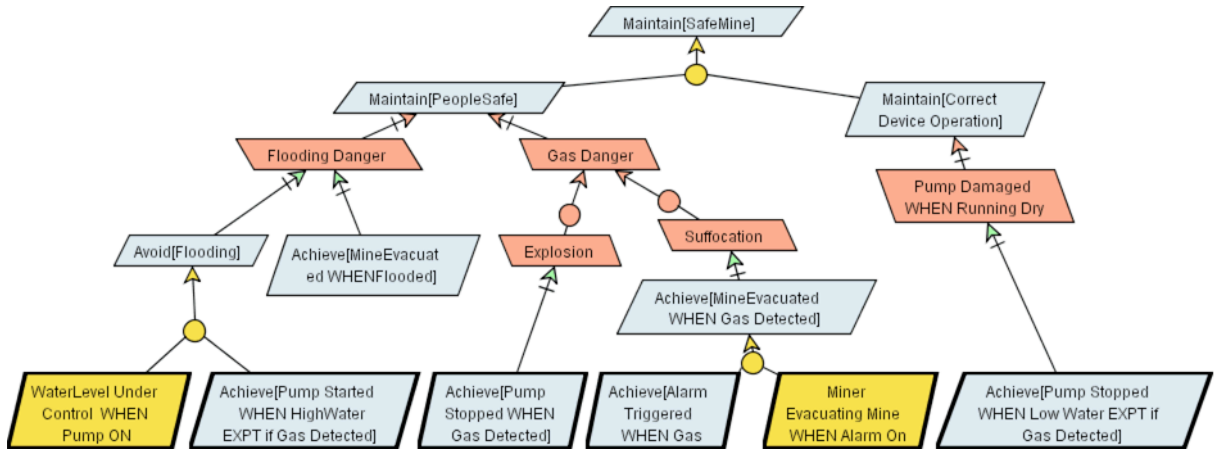


Figure 1: Goal/Obstacle Model of the Mine Sump System

the mine safe and the main refinement strategy is to avoid obstacles to safety. These obstacles (in red) are summarised by the flooding and methane dangers and the risk of damaging the pump when the sump runs dry. The methane danger is further refined to the dangers of explosion and suffocation. Mitigating those dangers yield four low level requirements under the responsibility of the system (in blue with thick outline at bottom line of the figure) with some necessary domain conditions and expectations (in yellow).

For sake of simplicity and because mapping of class-like model on Event-B has already been addressed [23], the domain is not structured in a complex object model but simply represented by five attributes of the *mine* system: *highWater*, *lowWater*, *pump*, *methane* and *bell*. Note that *highWater* and *lowWater* cannot hold at the same time. Based on this, our requirements can be formalised as follows.

**Requirement** Achieve[PumpStoppedWHENGasDetected]
  **Refines**  Avoid[Explosion]
  **FormalDef**  $(\forall m : Mine)\ m.methane = True \Rightarrow\ \circ\ m.pump = Off$

**Requirement** Achieve[AlarmTriggeredWHENGasDetected]
  **Refines**  Avoid[Suffocation]
  **FormalDef**  $(\forall m : Mine)\ m.methane = True \Rightarrow\ \circ\ m.bell = On$

**Requirement** Achieve[PumpStartedWHENHighWaterEXPTmethanePresent]
  **Refines**  Avoid[MineFlooded], Avoid[Explosion]
  **FormalDef**  $(\forall m : Mine)\ m.highWater = True \Rightarrow\ \Diamond_{\leq d_1} (m.methane \neq True \Rightarrow\ m.pump = On)$

**Requirement** Achieve[PumpStoppedWHENLowWaterEXPTmethanePresent]
  **Refines**  Avoid[MineFlooded]
  **FormalDef**  $(\forall m : Mine)\ m.lowWater = True \Rightarrow\ \Diamond_{\leq d_2} (m.methane \neq True \Rightarrow\ m.pump = Off)$

The dangers arising from the presence of methane have higher priority than flooding (people can still be evacuated when flooding occurs) and pump damage (not people). Therefore, the first two (gas related) requirements will have priority on the last two. This is why the pump is not started in the presence of gas (conflict resolution) and why *next* is used instead of bounded eventuality in the first two requirements.

The deadlines, $d_i$, represent the amount of time within which the predicates following $\Diamond_{\leq d_i}$ must become true. When $d_i$ is equal to 0, then bounded eventually $\Diamond_{\leq d_i}$ becomes the next operator. Due to the above safety priority, both $d_1$ and $d_2$ must be more than 0.

It is worth noting that a number of verifications can already be addressed at this level: goal and obstacle refinements, conflicts resolution. Some tool support is available [22], mostly based on model-checking. The Event-B mapping will bridge the gap with the next development step but will also give access to a larger set of tools, including proof-based ones.

## 3   Event-B with Obligations

Event-B is a specification language for developing discrete systems [4]. Behavioural aspects of Event-B models are expressed by means of *machines*. A machine is defined in terms of a global *state* consisting of a set of *variables*, and some *events* that cause the state to change by updating the values of the variables as defined by the *generalised substitution* of the event. Events are guarded by a *condition*, which when satisfied implies that the event is permitted to execute by applying its generalised substitution in the current state of the machine. Event-B also incorporates a refinement methodology, which can be used by software architects to incrementally develop a model of a system starting from the initial most abstract specification and following gradually through layers of detail until the model is close to the implementation. *Invariants* denoting desirable behaviour can be specified at each layer of detail as well as across

different layers.

In Event-B, an event is defined by the following syntax:

$$ev ::= \quad \texttt{EVENT } e \texttt{ WHEN } G \texttt{ THEN } S \texttt{ END}$$

where $G$ is the guard, expressed as a first-order logical formula in the state variables, and $S$ is the generalised substitution, defined by the following syntax:

| $S$ | ::= | SKIP | Do nothing |
|---|---|---|---|
| | | $\mid x := E(v)$ | Deterministic substitution |
| | | $\mid$ ANY $t$ WHERE $P(t, v)$ THEN $x := F(t, v)$ END | Non-deterministic substitution |
| | | $\mid S \parallel S'$ | Parallel substitution |

SKIP is a do-nothing substitution, which does not affect the machine's state. The deterministic substitution, $x := E(v)$, assigns to variable $x$ the value of expression $E(v)$, defined over set of state variables $v$. In a non-deterministic substitution, ANY $t$ WHERE $P(t, v)$ THEN $x := F(t, v)$ END, it is possible to choose non-deterministically local variables, $t$, that will render the logical guard $P(t, v)$ true. If this is the case, then the substitution, $x := F(t, v)$, can be applied, otherwise nothing happens. Finally, substitutions can be composed in parallel, $S \parallel S'$. For a comprehensive description of the Event-B language, we refer the reader to more detailed references such as [4, 18].

The operationalisation of requirements into Event-B machines requires extending the machinery of Event-B to incorporate the notion of obligations. The extra machinery has already been formally defined in [6]. Obligations are needed since the linear temporal logic-based specifications of requirements usually define not only a maximal set of permitted behaviours but also a minimal set of obliged ones. When the guard of an event is true, there is no obligation to perform the event and its execution may be delayed as a result of, for example, interleaving it with other permitted events. The choice of scheduling permitted events is made non-deterministically. In [6], we describe how obligations can be modelled in Event-B as events with *triggers*. The trigger of an event is a first-order logical formula in the state variables expressing an obligation on when the event must be executed.

We introduce here three types of triggered events:

- WITHIN events. These represent the general case of triggered events and are written as follows:

  $$\texttt{EVENT } e \texttt{ WHEN } T \texttt{ WITHIN } n \texttt{ NEXT } S \texttt{ END}$$

  where $T$ is the trigger condition such that when $T$ becomes true, the event must be executed within at most $n + 1$ number of events, provided the trigger remains true. If the trigger changes to false within that number, the obligation to execute the event is canceled. This type of event represents a bounded version of the leads-to modality, represented by the obligation $(T \Rightarrow \Diamond_{\leq n}(T \rightarrow e))$.

- NEXT events. These events are a special case of the WITHIN events where $n = 0$. Their syntax is:

  $$ev ::= \quad \texttt{EVENT } e \texttt{ WHEN } T \texttt{ NEXT } S \texttt{ END}$$

  Whenever $T$ becomes true, then the event will be the next one to be executed, which fulfills the obligation $(T \Rightarrow \circ e)$.

- EVENTUALLY events. These are also a special case of the WITHIN events, but where the value of $n$ is unbounded and non-deterministically chosen. The syntax of the event is:

  $$ev ::= \quad \texttt{EVENT } e \texttt{ WHEN } T \texttt{ EVENTUALLY } S \texttt{ END}$$

  such that when $T$ becomes true, then the event will eventually be executed. The choice of $n$ (i.e. the deadline for executing the event) is made when the trigger becomes true and the value of $n$ is known only internally. This event is modelling the obligation $(T \Rightarrow \Diamond(T \rightarrow e))$.

Here, our triggered events do not include a guard; we are interpreting the guard as being the same as the trigger, that is the event is triggered when it is permitted. All of the above triggered events are syntactic sugar and can be encoded and refined in the standard Event-B language, as described in [6]. In the rest of the paper, we adopt Event-B with obligations as our system specification language.

### 3.1 Machine Consistency

The semantics of Event-B machines is expressed via proof obligations, which must be proved in order for the machine to be well defined.

**Definition 1** (Machine Consistency). *Let M be a Event-B machine with, obligations, state variables v and invariant $I(v)$. Machine M is consistent if:*

1. *(Feasibility)*

   - *For each un-triggered event* `EVENT e WHEN G THEN S END` *in M the following property holds:* $I(v) \land G(v) \rightarrow \exists v' \cdot S(v, v')$ .
   - *For each triggered event* `EVENT e WHEN T WITHIN n NEXT S END` *in M the following property holds:* $I(v) \land T(v) \rightarrow \exists v' \cdot S(v, v')$ .

2. *(Invariant Preservation)*

   - *For each un-triggered event* `EVENT e WHEN G THEN S END` *in M the following property holds:* $I(v) \land G(v) \land S(v, v') \rightarrow I(v')$ .
   - *For each triggered event* `EVENT e WHEN T WITHIN n NEXT S END` *in M the following property holds:* $I(v) \land T(v) \land S(v, v') \rightarrow I(v')$ .

3. *(Deadlock Freeness for Triggered Events)*

   - *Every triggered event* `EVENT e WHEN T WITHIN n NEXT S END` *in M will be executed within at most $n+1$ events, provided trigger T remains true.*

Deadlock freeness is not easy to prove in general. In [6], we introduced strategies to prove such a property. For instance, if the machine consists only of `NEXT` triggered events, it will be deadlock-free if all triggers are mutually disjoint.

## 4 Operationalising Requirements into Event-B Specifications

The essence of goal refinement is to decompose a goal into sub-goals that can be implemented by the components of the software-to-be. Such sub-goals are called *requirements*. The process of assigning requirements (declarative property specifications) to their systems components (operational specifications) is called *operationalisation*. Our approach to operationalisation is to propose an Event-B-based specification, which represents a *consistent* machine, and then verify that the machine meets the requirements. In this sense, the machine is considered to be a *model* of the requirements.

We do not define a notion of correct operationalisation at the level of individual events, instead we develop a notion of correctness associated to an Event-B machine in relation to a set of requirements. This is formalised by the following.

**Definition 2** (Correct Operationalisation). *Given KAOS requirements $R_1, \cdots, R_k$, an Event-B machine M is a correct operationalisation of the requirements if the following conditions hold: 1) Machine M is consistent and 2) Machine M is a model for all requirements, i.e. $M \models R_i$ for $i = 1, \cdots, R_k$*

This definition provides a general solution to the problem of operationalisation, which could be achieved either through program verification or program construction methods. We use the notation $M \models P$ to indicate that the Event-B machine $M$ is a model for the linear temporal logic property $P$ according to the classical definition of a model checking problem [7].

Next, we demonstrate how patterns can assist the system designer in obtaining, in a constructive manner, a high-level system design in Event-B from system requirements expressed as real-time linear temporal logic formulae.

## 4.1 Operationalisation Patterns

In software engineering, patterns are defined as general reusable solutions to commonly recurring problems in software design. They define templates on how to solve a problem. Table 1 provides operationalisation patterns for three of the most frequently used goal patterns: *immediate generation* when using the next temporal operator, *eventually generation* when using the eventually temporal operator and *bounded generation* when using the eventually bounded operator. Here, we assume that $C$ and $S$ denote

| Requirement | Formal Definition | Event-B Operationalisation |
|---|---|---|
| Immediate Generation | $C \Rightarrow \circ S$ | EVENT $e$ WHEN $\overline{C}$ NEXT $\overline{S}$ END |
| Bounded Generation | $C \Rightarrow \Diamond_{\leq d} S$ | EVENT $e$ WHEN $\overline{C}$ WITHIN $d$ NEXT $\overline{S}$ END |
| Eventually Generation | $C \Rightarrow \Diamond S$ | EVENT $e$ WHEN $\overline{C}$ EVENTUALLY $\overline{S}$ END |

Table 1: Patterns for Operationalising Requirements into Event-B

first-order logical formulae defined over the space of objects of the specification-to-be. In the Event-B machine, these objects are defined as variables and so $\overline{C}$ is the corresponding formula over Event-B state and $\overline{S}$ is the generalised substitution derived from predicate $S$, where $S$ is seen as the post-condition of the substitution.

In summary, our method to derive an Event-B machine from system requirements comprises the following steps:

1. Transform the object state (e.g. KAOS Object Model) into the Event-B state. UML-B defines it for Event-B and proposes a related tool [23].

2. Derive Event-B events from KAOS requirements following the patterns presented in Table 1.

3. Complete the Event-B machine with the initialisation part and other events.

4. Verify that the Event-B machine is a correct operationalisation of the KAOS requirements.

The resulting Event-B machine is an abstract specification of a system meeting the KAOS requirements, which can be refined down to the implementation following the Event-B refinement method [4].

One important issue to note here is the relationship between time in the definition of requirements and duration of events in Event-B specifications. In goal-oriented methodologies, real-time temporal logic is used to formally specify goals and requirements. The main advantage of this logic is that real time temporal properties can be expressed simply using temporal operators without the explicit use of time variables. A linear temporal structure is used and time is related to states using a history function. Given a current state and a time unit, the "next" temporal operator refers to the next state in the linear temporal structure. We have modelled the temporal structure in Event-B with obligations by defining each event as having a duration of one time unit and associating each triggered event with a counter that controls that the event must be executed within the associated time constraint [6]. An abstract scheduler enforces

then the execution of an event when its associated counter is zero. For instance, we are interpreting the requirement of a "next" operation to indicate the next event in the machine operationalising the requirement, and this is enforced by making its counter zero when its trigger condition is true.

## 4.2 Operationalising the Mine Sump Requirements

In order to operationalise the requirements of the mine sump example introduced earlier, we need to represent the system's objects that are mentioned in those requirements as Event-B variables. This is done by mapping the objects to the variables as well as their corresponding value spaces using some bijection function. For simplicity, we assume this to be the identity function; each Event-B variable and its value space is named after its corresponding object and value space. After this step, it is possible to consider requirements as Event-B requirements that can be verified. We show in Table 2 the operationalisation of the requirements in the case of our mine sump example following the patterns suggested in Table 1.

| Requirement | Event-B Event |
|---|---|
| Achieve[PumpStartedWHENHighWaterEXPTmethanePresent] | *high_water_detected* |
| Achieve[PumpStoppedWHENLowWaterEXPTmethanePresent] | *low_water_detected* |
| Achieve[PumpStoppedWHENmethaneDetected] | *methane_detected* |
| Achieve[AlarmTriggeredWHENmethaneDetected] | *methane_detected* |

Table 2: Mapping the Mine Sump Requirements to Event-B Events

The derived definitions of these events are shown in the machine of Figure 2 modelled in Event-B with obligations [6]. The machine consists of events for detecting high and low water levels, detecting

```
INVARIANTS
  lowwater, highwater:  Bool
  methane:  Bool                pump, bell :  {ON, OFF}        highwater ∧ lowwater = false


EVENTS
Initialisation                  methane_detected              normal_to_low
  BEGIN                           WHEN methane = true            WHEN highwater = false
    highwater := false ||         NEXT pump := OFF ||                & lowwater = false
    lowwater := false ||              bell := ON                     & pump = ON
    methane := false ||         END                              THEN lowwater := true
    pump := OFF ||                                               END
    bell := OFF                 methane_leak
  END                             WHEN methane = false         low_to_normal
                                  THEN methane := true            WHEN highwater = false
high_water_detected             END                                & lowwater = true
  WHEN highwater = true                                             & pump = OFF
  WITHIN d1 NEXT pump := ON    high_to_normal                   THEN lowwater := false
  END                            WHEN highwater = true          END
                                     & lowwater = false
low_water_detected                   & pump = ON              normal_to_high
  WHEN lowwater = true           THEN highwater := false         WHEN highwater = false
  WITHIN d2 NEXT pump := OFF   END                                  & lowwater = false
  END                                                              & pump = OFF
                                                                 THEN lowwater := true
                                                                 END
```

Figure 2: The Mine Sump Machine in Event-B with Obligations

methane leak and changing the state of the high/normal/low water levels and the methane level. The machine is initialised in the normal state where the water level is between high and low levels, there is no

7

methane leak and both the pump and the alarm bell are off. The machine then makes transitions across the high, normal and low water levels as long as no methane is leaked. Whenever the high (low) water level is sensed, the machine obliges the high (low) water detection event to fire. This obligation must be fulfilled within $d_1 + 1$ $(d_2 + 1)$ number of events. If methane is leaked, the machine obliges the methane detection event to execute, which in turn shuts down the system by turning off the water sensors and the pump and sounds the alarm bell to evacuate the area. This event must be executed immediately in the next state following the methane leak since it is of a higher priority than any other event.

Finally, we establish the correctnes of our requirements operationalisation by the following theorem.

**Theorem 1.** *The Event-B machine presented in Figure 2 is a correct operationalisation of the requirements described in Section 2.1.*

**Proof**: The proof relies on showing that properties 1 and 2 of Definition 2 hold.

*Property 1.1*: the feasibility of the machine of Figure 2 can be proved trivially according to [18, Page 7, Figure 13], since all the events in the machine have deterministic substitutions.

*Property 1.2*: the invariant preservation property was expressed as five proof obligations for the (unsugared) machine when it was encoded in the Rodin tool (`http://www.event-b.org/platform.html`), all of which were discharged by the tool.

*Property 1.3*: to prove the deadlock freeness of the machine of Figure 2, we need to adopt a notion of schedulability as in [6] and then prove that the scheduler does not allow any one active counter (from the set $\{d1, d2\}$) to have the value of zero at any one computational slot. This implies that both counters must be always initialised with values above zero, i.e. bounded eventuality cannot be refined to next.

*Property 2*: for this property, we need to show that the machine of Figure 2 is a model of the requirements of Section 2.1. This was established by applying the ProB LTL model checker [15, 16], which verified that the machine is indeed a model of all the four requirements.

## 5   Related Work

There is a body of work on relating requirements and specifications. The standard operationalisation of KAOS requirements is presented in [14], where formal derivation rules map KAOS goal specifications, represented as real-time specifications, into specification of software operations, represented as sets of pre-, post- and trigger-conditions. This work has inspired us, but there are some differences. First, [14] requires a true-concurrency semantics for the operationalisation; by contrast, ours follows Event-B interleaving semantics. We consider the interleaving semantics to be more natural to developers. However it is more difficult for specifying timing constrains which required the Event-B extension presented previously [6]. Second, our definition does not require full equivalence between the specification and the Event-B machine. Third, our trigger condition is a state predicate while [14] is more general and allows for past formulaes. Finally, the use of well-established specification languages like Event-B allows the modeller to continue the development incrementally with a stepwise refinement method.

An early attempt to bridge requirements to specification in the context of the B method is presented in [21], which relates KAOS operations with B operations, and suggests to keep maintain properties as B invariants. The mapping is however totally informal and mainly focusing on deriving traceability links. In contrast our work is anchored in Event-B with a semantic link allowing both derivation and verification.

In [11], the authors propose a method for generating B operations from KAOS requirements that is driven by the aim of analysing desirable security properties in the system requirement and then preserving those properties when generating the system specification. However, their method is limited and lacks formality in that it assumes a syntactic relationship between KAOS and B operations. For example, the

authors do not demonstrate how triggering conditions in KAOS requirements, which represent obliged behaviour, are transformed and preserved throughout the B specification and refinement process.

In [13], the authors propose an operationalisation of KAOS requirements into event-based transition systems specified in the language of Labeled Transition Systems (LTS) of [17]. However, their main aim is to be able to analyse, such as checking for consistency and implicit requirements, and animate the KAOS operation models rather than bridge the requirements to another operational model, such as Event-B, which has allowed us to continue the refinement process throughout the system specification.

Finally, Tropos and i* are alternative modeling framework supporting the goal-oriented paradigm [25, 9]. These frameworks are dedicated to the analysis of dependencies in socio-technical systems. i* is centered on the structural modeling and does not support formal layer. On the other hand, FormalTropos supports a formal language similar to the one of the KAOS method [9], supporting linear temporal logic for goals, first order logic for pre/post conditions on tasks (called creation/fulfillment conditions) and for invariants on various kinds of objects (such as tasks and resources). This framework also supports formal verification through model checking techniques [10]. However those checks are limited to global checks respectively addressing the detection contradiction, overspecification and underspecification. There is no direct operationalisation relation of goals on other model element and related model-checking capabilities.

# 6   Conclusion and Future Work

This paper presented a constructive verification-based approach to linking high-level system requirements, expressed as linear temporal logic formulae, to a system specification expressed as an Event-B machine extended with the notion of obligations. The source requirements are included as verification assertions that can be model-checked by tools like ProB, showing that the proposed specification indeed meets the system requirements. The significance of this work is that it integrates goal-driven requirements engineering methodologies with refinement-driven system specification and design methodologies. Such a technique helps in bridging the gap between requirements and formal specifications as well as providing the means for building system designs based on rigorous formal grounds.

The work presented here is part of an on-going effort to help in the industrial adoption of Event-B and of a more specific effort to model security requirements of large-scale distributed systems like Grids [19, 8], to derive rigorously security policies from requirements [20], and to exploit well-established techniques, such as refinement, in the development of these types of systems [5].

Future work will further elaborate the pattern library and explicit proofs of operationalisation correctness. It will consider larger problems, with more complex object and agent models. This would allow us to address the lack of structure in Event-B and to capture the agent model in the specification by tracing agent-goal responsibility relationships. At the tool level, industrial level tools already exist both for goal-oriented requirements engineering [1] and Event-B [2] however integration at the formal level still needs to be developed.

# Acknowledgement

# References

[1] Objectiver. `http://www.objectiver.com`.

[2] Rodin. `http://sourceforge.net/projects/rodin-b-sharp/`.

[3] J-R. Abrial, M. J. Butler, S. Hallerstede, and L. Voisin. An Open Extensible Tool Environment for Event-B. In *Formal Methods and Software Engineering, ICFEM 2006*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer, 2006.

[4] J-R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[5] A. E. Arenas, B. Aziz, J. C. Bicarregui, and B. Matthews. Managing Conflicts of Interest in Virtual Organisations. In *3rd International Workshop on Security and Trust Management, STM 2007*. Elsevier, 2007.

[6] J. Bicarregui, A. E. Arenas, B. Aziz, P. Massonet, and C. Ponsard. Toward Modelling Obligations in Event-B. In E. Borger, M. Butler, and J. P. Bowen, editors, *International Conference of ASM, B and Z Users*, volume 5238 of *Lecture Notes in Computer Science*, pages 181–194. Springer, 2008.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[8] G. Dallons, P. Massonet, J. F. Molderez, C. Ponsard, and A. E. Arenas. An Analysis of the Chinese Wall Pattern for Guaranteeing Confidentiality in Grid-Based Virtual Organisations. In *International Workshop on Security, Trust and Privacy in Grid Systems, Grid-STP 2007*. IEEE, 2007.

[9] A. Fuxman, R. Kazhamiakin, M. Pistore, and M. Roveri. Formal tropos: language and semantics, 2003.

[10] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and Analyzing Early Requirements in Tropos. *Requirements Engineering*, 9(2):132–150, 2004.

[11] R. Hassan, S. Bohner, S. El-Kassas, and M. Eltoweissy. Goal-Oriented, B-Based Formal Derivation of Security Design Specifications from Security Requirements. In *ARES '08: Proc. of the 2008 Third Int. Conf. on Availability, Reliability and Security*, Washington, DC, USA, 2008. IEEE Computer Society.

[12] M. Joseph. *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall International, 1996.

[13] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Deriving event-based transition systems from goal-oriented requirements models. *Automated Software Engg.*, 15(2):175–206, 2008.

[14] E. Letier and A. van Lamsweerde. Deriving Operational Software Specifications from System Goals. In *FSE'10: 10th ACM S1GSOFT Symp. on the Foundations of Software Engineering*, 2002.

[15] M. Leuschel and M. Butler. ProB: A Model Checker for B. pages 855–. Springer-Verlag, LNCS, 2003.

[16] M. Leuschel and D. Plagge. Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more. Technical report, 2007.

[17] J. Magee and J. Kramer. *Concurrency - State Models and Java Programs*. John Wiley and Sons, 1999.

[18] C. Métayer, J. R. Abrial, and L. Voisin. Event-B Language. Rodin Deliverable D3.2, 2005.

[19] S. Naqvi, P. Massonet, and A. E. Arenas. Security Requirements Model for Grid Data Management Systems. In *Critical Information Infrastructures Security*, volume 4347 of *LNCS*. Springer, 2007.

[20] S. Naqvi, C. Ponsard, P. Massonet, and A. E. Arenas. Security Requirements Elaborations for Grid Data Management Systems. *International Journal of System of Systems Engineering*, 2009. To appear.

[21] C. Ponsard and E. Dieul. From Requirements Models to Formal Specifications in B. In *Proc; of CAISE Workshops, Regulations Modelling and their Validation and Verification, Luxembourg*, June 2006.

[22] C. Ponsard, P. Massonet, J-F. Molderez, A. Rifaut, A. van Lamsweerde, and H. T. Van. Early verification and validation of mission critical systems. *Formal Methods in System Design*, 30(3):233–247, 2007.

[23] C. Snook and M. Butler. UML-B and Event-B: an integration of languages and tools. In *SE2008*, 2008.

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

[25] E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, Department of Computer Science, University of Toronto, 1995.