Linking Modelling in Event-B with Safety Cases

Yuliya Prokhorova[1,2] and Elena Troubitsyna[2]

[1] TUCS – Turku Centre for Computer Science
[2] Department of Information Technologies, Åbo Akademi University
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
{Yuliya.Prokhorova,Elena.Troubitsyna}@abo.fi

Abstract. Safety cases are adopted in the certification process of many safety-critical systems. They justify why a system is safe and whether the design adequately incorporates safety requirements defined in a system requirement specification. The use of formal methods facilitates modelling and verification of safety-critical systems. In our work, we aim at establishing a link between formal modelling in Event-B and constructing a safety case. We propose an approach to incorporating safety requirements in a formal specification in such a way that it allows the developers to derive a safety case sufficient to demonstrate safety. We present a small case study illustrating the proposed approach.

Keywords: Event-B, formal specification, safety case, safety requirements, safety-critical systems.

1 Introduction

The use of formal methods in specification and verification of safety-critical systems has increased during the last decade. However, Habli and Kelly [1] point out that the use of an evidence generated from formal analysis is still an open issue in the system certification process. Basir et al. in [2] also state that formal methods, specifically formal proofs, provide justification for the validity of claims and widely deployed in software development. Nevertheless, the formal proofs are often too complex which causes uncertainties about trustworthiness of using formal proofs as the evidence in safety cases of safety-critical systems.

Another open issue related to the formal modelling process is whether the obtained formal model adequately represents safety requirements described in a system requirement specification. Several works address the question of requirements elicitation and traceability into a formal model [3,4,5]. Formal proofs as the evidence are only reasonable if those proofs are demonstrated to support incorporated safety requirements.

In this paper, we propose an approach to linking formal modelling in Event-B [6,7] with safety cases. We give the classification of safety requirements and define how each class can be represented in a formal specification. Additionally, we propose to split up a safety case into two main branches: argumentation over safety requirements and argumentation over the whole design. We define a number of invariants and theorems to support the argumentation. We use the

P. Avgeriou (Ed.): SERENE 2012, LNCS 7527, pp. 47–62, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Event-B framework to automatically generate the respective proof obligations which can be used in the safety case as the evidence that requirements have been met. The approach permits the developers to obtain a consistent system specification that allows for deriving a "sufficient" safety case. The problem what amount or what types of evidence to consider as "sufficient" is addressed in [8]. The authors determine the sufficiency of the evidence as *"its capability to address specific explicit safety assurance claims in a safety argument"*.

The paper is structured as follows. The safety case concept and modelling in Event-B principles are described in Section 2. In Section 3 we present our approach. In particular, in Subsection 3.1 we classify safety requirements and show the corresponding elements of an Event-B model. The link between Event-B and the main parts of a system safety case (represented using GSN) is given in Subsection 3.2. The verification support provided by the Event-B formalism for safety case arguments is discussed in Subsection 3.3. We illustrate our approach by an example – the sluice gate control system in Section 4. Finally, in Section 5 we give concluding remarks as well as discuss future and related work.

2 Background

2.1 Safety Cases

A safety case is *"a structured argument, supported by a body of evidence that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given operating environment"* [9,10].

A claim or a requirement, a safety argument and an evidence are the main elements of a safety case. Bishop and Bloomfield [10] define a *claim* to be the property of the system or some subsystem, while Kelly [11] defines a *requirement* as the safety objectives that must be addressed to assure safety. An *evidence* is the information extracted from analysis, testing or simulation of the system. The evidence makes the basis of the safety argument. An *argument* is a link between the evidence and the claim or requirements.

To represent the elements of a safety case and the relationships that exist between these elements, a graphical argumentation notation called *Goal Structuring Notation (GSN)* has been proposed by Kelly [11]. The principal elements of the notation are shown in Fig. 1.

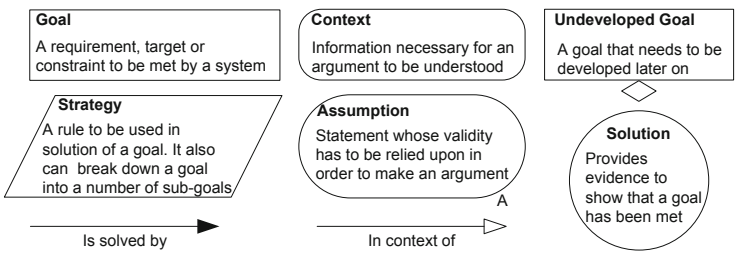

Fig. 1. Principal elements of GSN (detailed description is given in [11,12])

The safety case constructed in terms of GSN shows how goals (claims) are decomposed into sub-goals until the claim can be supported by the direct evidence (solution). It also defines the argument strategies and the context in which goals are declared. GSN has been adopted by a wide range of European companies from different domains: avionics, submarines, railways, etc. [1].

2.2 Modelling in Event-B

Event-B [6,7] is a state-based formal method for system level modelling and analysis. It is an extension of the B Method [13] that aims at facilitating modelling of parallel, distributed and reactive systems. Automated support for modelling and verification in Event-B is provided by the Rodin Platform [6].

In Event-B system models are defined using the notion of an *abstract state machine*. An abstract machine encapsulates the state (the variables) of a model and defines operations (events) on its state. The machine is uniquely identified by its name *MachineName*. The state variables of the machine are declared in the **VARIABLES** clause and initialised in the *INITIALISATION* event. The variables are strongly typed by the constraining predicates in terms of invariants given in the **INVARIANTS** clause. The data types and constants of the model are stated in a separate component called **CONTEXT**, where their properties are postulated as axioms. The behaviour of the system is determined by a number of atomic **EVENTS**. An event can be defined as follows:

$$e \mathrel{\hat{=}} \textbf{ANY } lv \textbf{ WHERE } g \textbf{ THEN } R \textbf{ END}$$

where lv is a list of local variables, the guard g is the conjunction of predicates defined over model variables, and the action R is a composition of assignments on the variables executed simultaneously.

The guard denotes when an event is enabled. If several events are enabled simultaneously then any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks. An assignment to a variable can be either deterministic or non-deterministic.

The semantics of Event-B events is defined using before-after predicates [14]. A before-after predicate (BA) describes a relationship between the system states before and after execution of an event. To verify correctness of a specification, one needs to prove that the model preserves its invariants, i.e., each event e_i of the model preserves the given invariant:

$$A(d,c),\ I(d,c,v),\ g_i(d,c,v),\ BA_i(d,c,v,v') \ \vdash\ I(d,c,v') \qquad \text{(INV)}$$

where A stands for the conjunction of the axioms, I is the conjunction of the invariants, g_i is the guard of the event e_i, BA_i is the before-after predicate of this event, d stands for the sets, c are the constants, and v, v' are the variable values before and after event execution.

2.3 Refinement and Verification in Event-B

Event-B employs a top-down refinement-based approach to formal development of a system. The development starts from an abstract specification of the system and continues with stepwise unfolding of system properties by introducing

new variables and events into the model. We call such kind of a refinement a *superposition refinement*. Moreover, Event-B formal development supports data refinement, allowing us to replace some abstract variables with their concrete counterparts. In this case, the invariant of a refined model formally defines the relationship between the abstract and concrete variables; this type of invariants is called a *gluing invariant*.

To verify correctness of a refinement step, one needs to discard a number of *proof obligations (PO)* for a refined model. For brevity, here we show only a few essential ones. The full list of proof obligations can be found in [7].

Let us introduce a shorthand $H(d, c, v, w)$ which stands for the hypotheses while $I(d, c, v)$ and $I'(d, c, v, w)$ are respectively the abstract and refined invariants and v, w are respectively the abstract and concrete variables.

The event guards in a refined model can only be strengthened in refinement:

$$H(d, c, v, w),\ g'_i(d, c, w)\ \vdash\ g_i(d, c, v) \tag{GRD}$$

where g_i, g'_i are respectively the abstract and concrete guards of the event e_i.

The *simulation* proof obligation (SIM) requires to show that the action (i.e., assignment on the state variables) of a refined event is not contradictory to its abstract version:

$$H(d, c, v, w),\ g'_i(d, c, w),\ BA'_i(d, c, w, w')\ \vdash\ \exists v'.BA_i(d, c, v, v')\ \wedge\ I'(d, c, v', w') \tag{SIM}$$

where BA_i, BA'_i are respectively the abstract and concrete before-after predicates of the same event e_i.

Finally, the Event-B formalism allows us to define theorems either in the context $T(d, c)$ or in the machine $T(d, c, v)$. The *theorem* proof obligation (THM) ensures that a proposed theorem is indeed provable. The first variant is defined for a theorem in a context:

$$A(d, c)\ \vdash\ T(d, c) \tag{THM}$$

The second variant is defined for a theorem in a machine:

$$A(d, c), I(d, c, v)\ \vdash\ T(d, c, v) \tag{THM}$$

The described proof obligations are automatically generated by the Rodin Platform [6] that supports Event-B. Additionally, the tool attempts to automatically prove them. Sometimes it requires user assistance by invoking its interactive prover. However, in general the tool achieves high level of automation (usually over 80%) in proving.

3 An Approach to Linking Modelling in Event-B with Safety Cases

In this section, we present an approach that establishes a link between formal modelling of a system in Event-B and deriving a safety case for this system. We aim at obtaining the safety case where the argumentation is based on formal

reasoning and supported by discharging proof obligations. To achieve this goal, we firstly give the classification of the safety requirements and show how each class is treated within the Event-B framework. Secondly, we show how the safety requirements, invariants and theorems as well as proofs for them correspond to the elements of the safety case. Finally, we provide verification support for safety case arguments by defining safety invariants and theorems formally.

3.1 Requirements Classification

Let us now give a classification of safety requirements. To provide the reader with the classification of safety requirements, we adopt and modify the taxonomy proposed by Bitsch [15]. Following his approach, we divide safety requirements into two groups: *Static Safety Requirements* and *Dynamic Safety Requirements*. The former are those properties that must hold for the whole formal model. The latter are those properties that must be true only in certain model states.

The *Dynamic Safety Requirements (DSRs)* cover a large group of safety requirements. To simplify the task of mapping them on the Event-B framework, we decompose the *Dynamic Safety Requirements* class into two sub-classes: *DSRs about General Access Guarantee* and *DSRs about Chronological Succession*.

To ensure safety of a certain class of control systems, we should prove that these systems are deadlock free. Therefore, DSRs about General Access Guarantee are defined as requirements which describe the necessity to provide an access to some property or to reach some state. While developing a formal model of a safety-critical control system, we also might deal with requirements that are dependent on the chronological occurrences of some properties, e.g., requirements that define fault tolerance procedures. Since fault detection, isolation and recovery actions are strictly ordered, we also need to preserve their sequence in a formal model of a system. The DSRs about Chronological Succession reflect such requirements. Hence, these sub-classes of DSRs are addressed differently in a formal model. Furthermore, the proposed classification can be extended with respect to, for example, timing properties. However, we leave such sub-classes out of the scope of this paper.

To define how each class of safety requirements can be treated in Event-B, we have analysed several works that aim at tracing requirements in a formal specification in Event-B [3,4,5]. For instance, Méry and Singh [3] propose to represent safety requirements as invariants or theorems, while Jastram et al. [4] incorporate them as invariants and before-after predicates of events. Additionally, in [5] Yeganefard and Butler state that requirements can be modelled in Event-B as guards or actions of events. All these works show the mapping between particular requirements and the Event-B structure. However, they do not consider the classification of the safety requirements. In contrast, we create the link between the requirements classification and their representation in Event-B as shown in Table 1. We propose to model *Static Safety Requirements* in the Event-B framework as invariants or incorporate them in the process of guards strengthening in refinement while *Dynamic Safety Requirements*, in general, can be related to after predicates or actions simulation in refinement. *DSRs about General access*

Table 1. Mapping of the safety requirements classification on the Event-B framework

Safety requirements class	Event-B framework
Static Safety Requirements	Invariants; Guards strengthening in refinement
Dynamic Safety Requirements	After predicates; Actions simulation in refinement
Dynamic Safety Requirements about General access guarantee	Deadlock freedom
Dynamic Safety Requirements about Chronological succession	Events order

guarantee are represented as the deadlock freedom condition, while *DSRs about Chronological succession* are defined as events order in Event-B.

As soon as all safety requirements are assigned to respective classes and the mapping on Event-B is done, we can argue that the system is safe.

3.2 Linking Safety Cases with the Event-B Framework

To provide an argument that the system is safe, we build a system safety case using GSN (Fig. 2). We introduce the main **Goal** *"**(G1)**: System is safe"* that is considered in the **Context** of formal modelling in Event-B ***(C1)***. Additionally, we propose to split up the overall process of the safety case derivation into two main parts: argumentation over each safety requirement and argumentation that the design is satisfactory. To represent this in terms of GSN, we introduce two respective **Strategies**: *"**(S1)**: Argument over each safety requirement"* and *"**(S2)**: Argument that design is satisfactory"*.

The complete safety case with respect to the system safety requirements is obtained by providing an evidence that the safety requirements listed in the Requirements Document (RD) are derived and adequately formalised. For this purpose, we introduce an **Assumption** *"**(A1)**: All safety requirements are derived from RD"*. We support this assumption by performing hazard analysis. In this case, our goal is to obtain the safety requirements list ***(G2)***. The **Solution** (or the evidence) is to conduct one or a combination of well-known hazard analysis techniques such as Failure Modes and Effects Analysis (FMEA), HAZard and OPerability analysis (HAZOP), Preliminary Hazard Analysis (PHA), etc.

We arrange the derived safety requirements according to the classification proposed in Section 3.1. Therefore, we introduce several **Sub-goals** (or goals) that correspond to the given safety requirements classes and their mapping on Event-B (proposed in Table 1):

***(G3)**: Static Safety Requirements (invariants)*
***(G4)**: Static Safety Requirements (guards strengthening in refinement)*
***(G5)**: Dynamic Safety Requirements (after predicates)*
***(G6)**: Dynamic Safety Requirements (actions simulation in refinement)*
***(G7)**: Dynamic Safety Requirements. General access guarantee (deadlock freedom)*
***(G8)**: Dynamic Safety Requirements. Chronological succession (events order)*

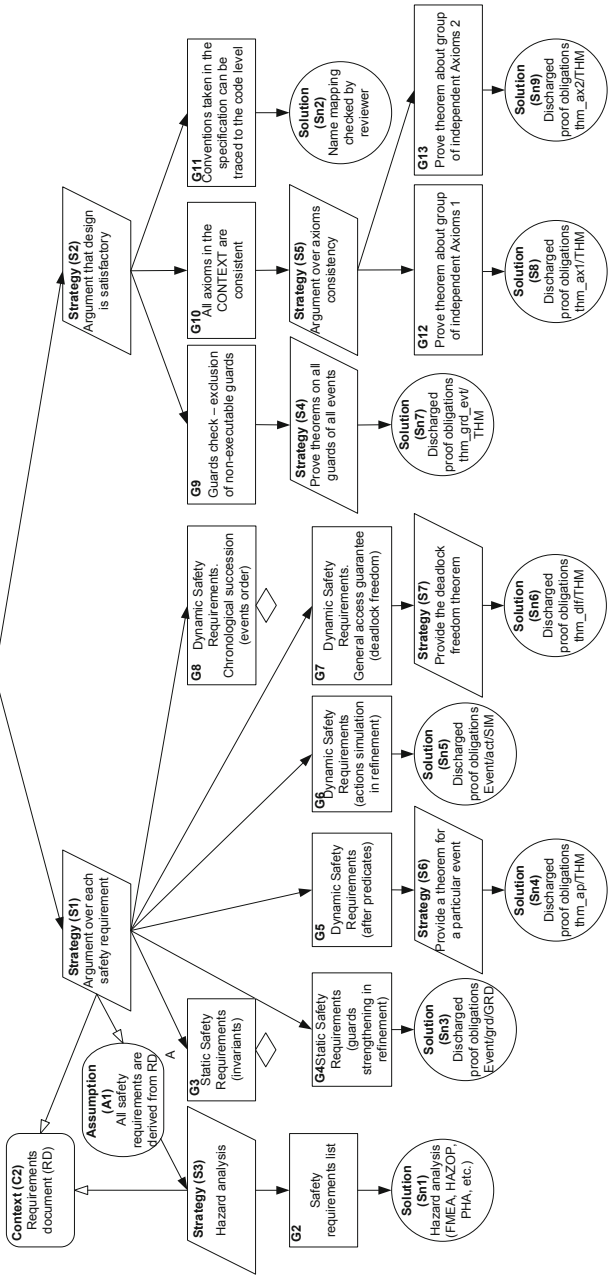

Fig. 2. System Safety Case

The goals ***(G4)*** and ***(G6)*** simply require a solution that can be provided in terms of Event-B by discharged proof obligations ***(Sn3)*** and ***(Sn5)***, respectively. However, the goals ***(G3)***, ***(G5)***, ***(G7)*** and ***(G8)*** need to be decomposed before deriving the evidence for them.

The safety requirements given as invariants ***(G3)*** must be supported by an argumentation concerning their proper formalisation: assumption ***(A3.1)*** and respective strategy ***(S3.3)*** as shown in Fig. 3. We also need to prove that the invariant holds for all events by providing an argument over each event independently, e.g., ***(G3.1)***. The discharged proof obligations for each event serve as solutions for this branch of the safety case, e.g., ***(Sn3.1)***, ***(Sn3.2)***, etc.

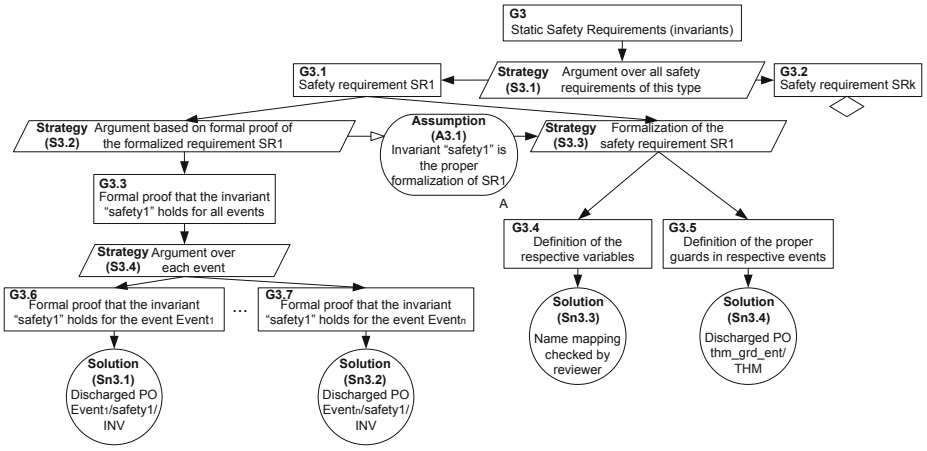

Fig. 3. Safety Case for Static Safety Requirements of Invariant Type

To decompose the goal ***(G5)***, we introduce a theorem stating that an after predicate holds for a particular event ***(S6)***. The proof obligation ***(Sn4)*** supports this claim. Respectively, for ***(G7)*** we provide a deadlock freedom theorem ***(S7)*** and the evidence ***(Sn6)*** as shown in Fig. 2.

To produce the evidence for the dynamic safety requirements about chronological succession ***(G8)***, we propose to adopt the flow approach introduced by Iliasov [16]. However, due to the lack of space, we omit the detailed description of the respective safety case.

Despite the fact that safety requirements might be adequately represented in a formal specification, the goal *"**(G1)**: System is safe"* can still be unreachable. The design (the formal specification) can contain inconsistency and improper representation of physical components in terms of variables, constants and sets. Therefore, we need to show that our design is consistent and adequate as well. We consider such **Sub-goals** as *"**(G9)**: Guards check – exclusion of non-executable guards"*, *"**(G10)**: All axioms in the CONTEXT are consistent"* and *"**(G11)**: Conventions taken in the specification can be traced to the code level"*. To support the first sub-goal, theorems on all guards of all events have to be introduced in the specification ***(S4)*** and the derived proof obligations can serve as solutions ***(Sn7)***. The second

sub-goal requires argumentation over axioms consistency which leads to the decomposition of this goal into sub-goals ***(G12)***, ***(G13)***. This level sub-goals represent different groups of independent axioms. To provide the evidence, we propose to introduce and prove theorems for consistency of each group of axioms, e.g., ***(Sn8)*** and ***(Sn9)***. The solution to the third sub-goal is the name mapping checked by a reviewer ***(Sn2)***. The reader can consult Fig. 2 for recalling the precise meaning of abbreviations.

3.3 Verification Support for Safety Case Arguments

In the previous subsection, we created the link between safety cases and the Event-B framework. Now, we show the formalisation of this link, i.e., how the proposed invariants and theorems can be postulated in Event-B and which proof obligations support them.

First, let us consider the classes of safety requirements that simply require an evidence and can be implemented in a refinement step, i.e., *"**(G4)**: Static Safety Requirements (guards strengthening in refinement)"* and *"**(G6)**: Dynamic Safety Requirements (actions simulation in refinement)"*.

In a refinement step an abstract variable, e.g., abs_var_k, can be replaced by some new concrete variables $conc\_var$. For instance, the variable that represents a failure of a system might be replaced by variables representing failures of concrete system units (sensors, actuators, etc.). For such a data refinement, we introduce a gluing invariant of the form: $abs\_var_k = VALUE \Leftrightarrow P(conc\_var)$, where P is a predicate on new concrete variables.

To illustrate the considered classes of requirements, we introduce the abstract event *Event_Name* that contains one local variable and its refinement *New_Evt_Name* as shown in Fig. 4. The label $@grd_i$ stands for the *i-th* guard of an event whilst $@act_j$ represents the *j-th* action of an event. Since we replace the abstract local variable n by a concrete value TRUE in the refinement, we

// Event in the abstract machine	*// Event in the refined machine*
EVENT Event_Name **ANY** *n* **WHERE** @grd1 *n* ∈ BOOL @grd2 abs_var = FALSE @grd3 var = FALSE **THEN** @act1 abs_var ≔ n @act2 var ≔ n **END**	**EVENT** New_Evt_Name **REFINES** Event_Name **WHERE** @grd2 $conc\_var_1$ = FALSE ∧ $conc\_var_2$ = FALSE @grd3 var = FALSE **WITH** @n n = TRUE **THEN** @act2 var ≔ TRUE **END**
New_Evt_Name/grd2/GRD *// Proof obligation of a guard strengthening in the refinement* $conc\_var_1$ = FALSE ∧ $conc\_var_2$ = FALSE var = FALSE |- abs_var = FALSE	
New_Evt_Name/act2/SIM *// Proof obligation of an action simulation in the refinement* $conc\_var_1$ = FALSE ∧ $conc\_var_2$ = FALSE var = FALSE |- TRUE = TRUE	

Fig. 4. The example of abstract and concrete events with proof obligations

need to provide a *witness* for it (the **WITH** clause). The witness substitutes the disappearing local variable with a new variable or a concrete value.

To support these safety requirements in the safety case, we provide the proofs that are instances of (GRD) and (SIM), respectively for ***(Sn3)*** and ***(Sn5)***.

Next, we state the Static Safety Requirements ***(G3)*** as safety invariants of the model, i.e., $I(d, c, v)$. The respective proof obligations (INV) serve as the evidence for the safety case and are automatically generated for all model events.

Now, we formalise the Dynamic Safety Requirements implemented as after predicates *"**(G5)**: Dynamic Safety Requirements (after predicates)"*. The simplification of the event definition by omitting local variables does not affect the generality of the proposed approach since any event containing local variables can always be rewritten in a simpler form. Hence, the definition of an event presented in Section 2.2 can be given as the relation: $e(v, v') = g_e(v) \wedge BA_e(v, v')$. Then, *before(e)* represents a set of all possible pre-states defined by the guard of an event e while *after(e)* is a set of all possible post-states of the event e [16]:

$$before(e) = \{v \in \Sigma \mid I(v) \wedge g_e(v)\}$$
$$after(e) = \{v' \in \Sigma \mid I(v') \wedge (\exists v \in \Sigma \cdot I(v) \wedge g_e(v) \wedge BA_e(v, v'))\}$$

where Σ corresponds to a model state space defined by all possible values of the vector v (i.e., a set of system variables).

Thereafter, we assume that $q(v')$ is a certain desired post-state. We can verify that this post-state is always established by proving the following theorem:

$$\forall v' \ \cdot \ v' \in after(e) \ \Rightarrow \ q(v')$$

This theorem serves as the strategy ***(S6)*** in the respective branch of the safety case while the discharged proof obligation of the type (THM) defined for a machine provides the solution ***(Sn4)***.

Another class of the Dynamic Safety Requirements is the requirements about general access guarantee ***(G7)***. We represent this class within an Event-B model as the deadlock freedom theorem ***(S7)***. This theorem is postulated as the disjunction of guards of all model events $g_1(d, c, v) \vee ... \vee g_m(d, c, v)$. The instance of the (THM) proof obligation for a machine given in Section 2.3 provides the evidence for the safety case ***(Sn6)*** and is shown below:

$$A(d, c), \ I(d, c, v) \ \vdash \ g_1(d, c, v) \vee ... \vee g_m(d, c, v)$$

Please note that the application of this rule is not compulsory, since not all systems need to be deadlock free. For example, if the system allows the manual operation or shutdown when failure occurs, the model can be deadlocked.

To verify that our formal specification (the design) is satisfactory and there are no non-executable guards ***(G9)***, we introduce theorems for checking guards ***(S4)***. We base our reasoning on the notion of post- and pre-states as well. The following theorem guarantees that there are no non-executable guards in the model:

$$\forall i \ \exists j \ \cdot \ i \neq j \ \wedge \ after(e_j) \ \Rightarrow \ before(e_i)$$

This theorem states that for each event e_i there exists an event e_j that enables it (i.e., e_j enables e_i). The exceptions are the initialisation event, which is always enabled at the beginning of the simulation independently of other events, and events without guards, i.e., their guards are always true. The provided theorem and the respective proof obligation ensure that each event of the model is enabled at least once, i.e., the model does not include events with non-executable guards.

The inconsistency in the model axioms has an impact on the whole model ***(G10)***. If axioms contradict to each other, the model is falsifiable, i.e., we cannot guarantee its correctness any more. To avoid this, we define theorems for all groups of independent axioms in the model CONTEXT ***(S5)***:

$$\exists\, d, c \;\cdot\; A_1(d, c) \;\wedge\; \ldots \;\wedge\; A_n(d, c)$$

The generated proof obligation, e.g., ***(Sn8)***, is an instantiation of the (THM) proof obligation for a context given in Section 2.3.

At this point, we have classified safety requirements and have shown how each class can be treated in the formal specification. Additionally, we have defined the link between the Event-B framework and safety cases. In the next section we illustrate our approach by a realistic system – the sluice gate control system.

4 Case Study – Sluice Gate Control System

The sluice gate control system shown in Fig. 5 is a typical representative of a safety-critical control system. This system controls a sluice that connects areas with dramatically different pressures [17]. The purpose of the system is to adjust the pressure in the sluice area. The system consists of two doors – *door1* and *door2* that can be operated independently of each other and a pressure chamber pump that changes the pressure in the sluice area (i.e., the room). To guarantee safety, a door can be opened only if the pressure in the locations it connects is equalized. Moreover, at most one door can be opened at any moment and the pressure chamber pump can only be switched on when both doors are closed. The sluice gate control system is equipped with the sensors and actuators (motors) as shown in Fig. 5.

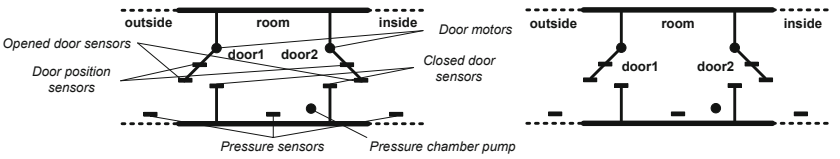

Fig. 5. Sluice Gate System

For the sake of brevity, here we show only those parts of the specification that are relevant to the approach we present. More details on the specification can be found in [17].

Fig. 6 gives an example of a static safety requirement formalisation that represents guards strengthening in the refinement for *"SR1: The system failure occurs*

// Event in the abstract machine	*// Event in the refined machine*
EVENT Prediction	**EVENT** Prediction **REFINES** Prediction
WHERE	**WHERE**
@grd1 flag = PRED	@grd1 flag = PRED
@grd2 Failure = FALSE	@grd2 door1_fail = FALSE ∧ door2_fail = FALSE ∧ pressure_fail = FALSE
@grd3 Stop = FALSE	@grd3 Stop = FALSE
THEN	**THEN**
@act1 flag ≔ ENV	@act2 d1_exp_min ≔ min_door(door1_position↦door1_motor)
	...
END	**END**

Prediction/grd2/GRD *// Proof obligation of a guard strengthening in the refinement*
flag = PRED
door1_fail = FALSE ∧ door2_fail = FALSE ∧ pressure_fail = FALSE
Stop = FALSE
|- Failure = FALSE

Fig. 6. The event *Prediction* and the respective proof obligation

if and only if either the door1 component fails, or the door2 component fails, or the pressure pump fails".

The event *Prediction* taken as an example models the prediction of the expected values of sensors based on the current state of the system and physical laws of components operation. Later, the obtained information permits the system to detect faults of components by the comparison between expected values and the received ones.

The dynamic safety requirement *"SR2: To handle a system failure, the systems should stop its operation"* is formalised as an action simulation in the refinement step. The event *SafeStop* depicted in Fig. 7 represents the refinement of the abstract event *ErrorHandling*. The deterministic assignment to the variable *Stop* substitutes the non-deterministic assignment to this variable with value TRUE. Additionally, the refined event is supported with the respective witness (@res res = TRUE). The generated proof obligation of the type (SIM)

// Event in the abstract machine	*// Event in the refined machine*
EVENT ErrorHandling	**EVENT** SafeStop **REFINES** ErrorHandling
ANY *res* **WHERE**	**WHERE**
@grd1 flag = CONT	@grd1 flag = CONT
@grd2 Failure = TRUE	@grd2 door1_fail = TRUE ∨ door2_fail = TRUE
@grd3 Stop = FALSE	∨ pressure_fail = TRUE
@grdres *res* ∈ BOOL	@grd3 Stop = FALSE
THEN	**WITH** @res res = TRUE
@act1 flag ≔ PRED	**THEN**
@act2 Failure ≔ *res*	@act1 flag ≔ PRED
@act3 Stop ≔ *res*	@act3 Stop ≔ TRUE
END	**END**

SafeStop/act3/SIM *// Proof obligation of an action simulation in the refinement*
flag = CONT
door1_fail = TRUE ∨ door2_fail = TRUE ∨ pressure_fail = TRUE
Stop = FALSE
|- TRUE = TRUE

Fig. 7. The event *SafeStop* and the respective proof obligation

shows that the assignment to the state variable *Stop* of the refined event is not contradictory to its abstract version.

Such safety requirements as: *"SR3: Both doors cannot be opened simultaneously"*, *"SR4: If the pressure inside the room is not equal neither to the pressure of the outside area nor to the pressure of the inside area, the doors must remain closed"*, *"SR5: The pressure pump can be switched on only if both doors are closed"*, etc. can be formalised as invariants. Therefore, let us show only the safety case for SR5 (Fig. 8). We introduce the invariant *"safety5"* as a formalisation of SR5: $failure = FALSE \wedge pump \neq PUMP\_OFF \Rightarrow (door1\_position = 0 \wedge door2\_position = 0)$.

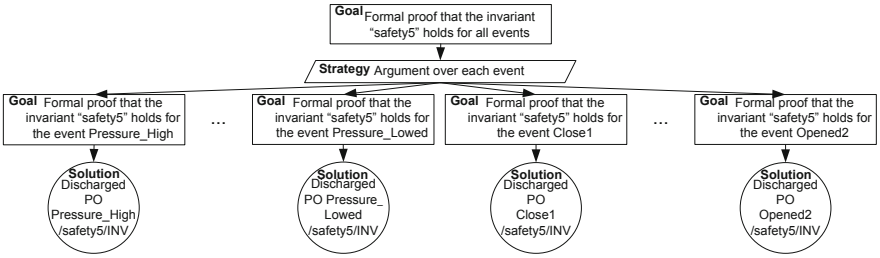

Fig. 8. Safety Case for the Safety Requirement SR5

To support the claim that *"safety5"* holds for all events of the model, we state an argument over each event and discard the proof obligations of the type (INV). For brevity, here we show only the event *Pressure_High* that models the increase of the pressure inside the room and display the proof obligation ensuring that "safety5" holds for this event (Fig. 9).

EVENT Pressure_High **REFINES** NormalSkip **WHERE** @grd1 door1_position = 0 @grd2 door2_position = 0 @grd3 pressure_value = PRESSURE_OUTSIDE @grd0_1 flag = CONT @grd0_2 failure = FALSE @grd0_3 Stop = FALSE **THEN** @act1 flag ≔ PRED @act2 pump ≔ PUMP_INC **END**	**Pressure_High/safety5/INV** failure = FALSE ∧ pump ≠ PUMP_OFF ⇒ door1_position = 0 ∧ door2_position = 0 door1_position = 0 door2_position = 0 pressure_value = PRESSURE_OUTSIDE flag = CONT failure = FALSE Stop = FALSE |- failure = FALSE ∧ PUMP_INC≠PUMP_OFF ⇒ door1_position = 0 ∧ door2_position = 0

Fig. 9. The event *Pressure_High* and the respective proof obligation

Let us consider the dynamic safety requirement formalised as an after predicate: *"SR6: If the pressure value in the room is equalised to the pressure value of the inside area, the pump should be switched off"*. The desired post-state q for this safety requirement is $pump = PUMP\_OFF$. The event *Pressure_Highed* models this case and has the set of all possible post-states $after(Pressure\_Highed)$ as shown in Fig. 10. Here we omit showing other model variables since the event *Pressure_Highed* does not modify them, i.e., they remain the same. The given

EVENT Pressure_Highed **REFINES** NormalSkip **WHERE** @grd2 pump = PUMP_INC @grd3 pressure_value = PRESSURE_INSIDE @grd0_1 flag = CONT @grd0_2 failure = FALSE @grd0_3 Stop = FALSE **THEN** @act1 flag ≔ PRED @act2 pump ≔ PUMP_OFF **END**	*after(Pressure_Highed)* = { (pump, flag, pressure_value, failure, Stop, ...) | pump = PUMP_OFF /\ flag = PRED /\ pressure_value = PRESSURE_INSIDE /\ failure = FALSE /\ Stop = FALSE } q ≙ pump = PUMP_OFF

Fig. 10. The event *Pressure_Highed*

$after(Pressure\_Highed)$ implies that q is true. The proof obligation of the type (THM) provides the evidence for the respective branch of the safety case.

Due to manual handling of a system failure, the sluice gate control system has a deadlock. There is an event that sets the variable *Stop* to TRUE (the event *SafeStop* in Fig. 7), while none of the events assigns FALSE to it. To reset the variable *Stop*, the system should be restarted. Therefore, we do not include the deadlock freedom branch in the system safety case.

Finally, we support our modelling of the sluice gate control system with argumentation over the whole design. To exclude non-executable guards, all events are supported by the respective theorems. The generated proof obligations of the type (THM) serve as the evidence. The consistency of axioms is also checked and proved for the sluice gate system through discarding the corresponding theorems in the model *CONTEXT*.

Since we have defined one-to-one mapping between a safety case and a formal model, we can use this mapping to generate the safety case from the model.

5 Related Work and Conclusions

5.1 Related Work

The work presented by Basir et al. [2,12] is dedicated to formal program verification using a safety case construction. It establishes a link between automatically generated program code and a formal analysis, based on automated theorem proving. The authors focus on natural deduction (ND) style proofs and explain how to construct the safety cases by converting the ND proof tree into corresponding safety case elements. Unlike [2,12], in our work we do not go so deeply in the proof obligations semantics provided by Event-B and do not introduce inference rules [7] as elements of the safety case. We rather guide the modelling of safety-critical systems in Event-B and use the proofs as an evidence for a safety case.

Habli and Kelly [1] consider two standards, DO-178B and the UK Defence Standard 00-56, to analyse how formal analysis facilitates achieving the certification goals. They also present a generic safety case for presentation and justification of formal analysis that shows feasibility of applying formal methods

within a specific development. In contrast, our approach not only focuses on the use of formal methods in a safety case but also covers the design stage where the decisions how requirements can be implemented in a formal model are made.

5.2 Conclusions

In this paper, we have proposed the approach to support a system safety case with an evidence derived from a formal specification in Event-B. The safety case construction has been divided into two main parts: argumentation over safety requirements incorporation and argumentation over the whole model. To help the developers in the structured argumentation over safety requirements, we have provided the classification of safety requirements and have shown the link between informal requirements description and their formalisation within Event-B. We also defined mapping between the Event-B model and the elements of the safety case.

Obviously, the larger and more complex a safety-critical system, the larger and more complex its safety case. To increase effectiveness of safety case construction, availability of tools is essential. Therefore, as a part of our future work we plan to provide a tool support for automatic generation of safety cases as well as validate the proposed approach on large-scale case studies.

Acknowledgments. The authors would like to thank Linas Laibinis for fruitful discussions as well as Ilya Lopatkin, Alexei Iliasov and Alexander Romanovsky for their valuable feedback on the case study.

References

1. Habli, I., Kelly, T.: A Generic Goal-Based Certification Argument for the Justification of Formal Analysis. Electronic Notes in Theoretical Computer Science 238(4), 27–39 (2009)
2. Basir, N., Denney, E., Fischer, B.: Deriving Safety Cases from Machine-Generated Proofs. In: Proceedings of the Workshop on Proof-Carrying Code and Software Certification (PCC 2009), Los Angeles, California, USA (2009)
3. Méry, D., Singh, N.K.: Technical Report on Interpretation of the Electrocardiogram (ECG) Signal using Formal Methods. Technical Report inria-00584177 (2011)
4. Jastram, M., Hallerstede, S., Ladenberger, L.: Mixing Formal and Informal Model Elements for Tracing Requirements. ECEASST 46 (2011)
5. Yeganefard, S., Butler, M.: Structuring Functional Requirements of Control Systems to Facilitate Refinement-based Formalisation. ECEASST 46 (2011)
6. Event-B and the Rodin Platform (2012), `http://www.event-b.org/`
7. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
8. Hawkins, R., Kelly, T.: A Structured Approach to Selecting and Justifying Software Safety Evidence. In: Proceedings of the 5th IET International Conference on System Safety, pp. 1–6 (2010)
9. UK Ministry of Defence. 00-56 Safety Management Requirements for Defence Systems (2007)

10. Bishop, P., Bloomfield, R.: A Methodology for Safety Case Development. In: Safety-Critical Systems Symposium. Springer, Birmingham (1998)
11. Kelly, T.P.: Arguing Safety – A Systematic Approach to Managing Safety Cases. Doctoral Thesis (1998)
12. Basir, N.: Safety Cases for the Formal Verification of Automatically Generated Code. University of Southampton, Dependable Systems and Software Engineering, ECS. Doctoral Thesis (2010)
13. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
14. Metayer, C., Abrial, J.-R., Voisin, L.: Rigorous Open Development Environment for Complex Systems (RODIN). Event-B (2005), `http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf`
15. Bitsch, F.: Safety Patterns - The Key to Formal Specification of Safety Requirements. In: Voges, U. (ed.) SAFECOMP 2001. LNCS, vol. 2187, pp. 176–189. Springer, Heidelberg (2001)
16. Iliasov, A.: Use Case Scenarios as Verification Conditions: Event-B/Flow Approach. In: Troubitsyna, E.A. (ed.) SERENE 2011. LNCS, vol. 6968, pp. 9–23. Springer, Heidelberg (2011)
17. Lopatkin, I., Prokhorova, Y., Troubitsyna, E., Iliasov, A., Romanovsky, A.: Patterns for Representing FMEA in Formal Specification of Control Systems. TUCS Technical Reports 1003, Turku Centre for Computer Science (2011)