Yuliya Prokhorova | Linas Laibinis | Elena Troubitsyna

Towards Rigorous Construction of Safety Cases

Turku Centre *for* Computer Science

TUCS Technical Report
No 1110, May 2014

Towards Rigorous Construction of Safety Cases

Yuliya Prokhorova
TUCS – Turku Centre for Computer Science,
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
yuliya.prokhorova@abo.fi
Linas Laibinis
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
linas.laibinis@abo.fi
Elena Troubitsyna
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
elena.troubitsyna@abo.fi

TUCS Technical Report
No 1110, May 2014

Abstract

Certification of safety-critical software systems requires submission of safety assurance documents, e.g., in the form of safety cases. A safety case is a justification argument used to show that a system is safe for a particular application in a particular environment. Different argumentation strategies are applied to determine the evidence for a safety case. They allow us to support a safety case with such evidence as results of hazard analysis, testing, simulation, etc. On the other hand, application of formal methods for development and verification of critical software systems is highly recommended for their certification. In this paper, we propose a methodology that combines these two activities. Firstly, it allows us to map the given system safety requirements into elements of the formal model to be constructed, which is then used for verification of these requirements. Secondly, it guides the construction of a safety case demonstrating that the safety requirements are indeed met. Consequently, the argumentation used in such a safety case allows us to support the safety case with formal proofs and model checking results as the safety evidence. Moreover, we propose a set of argument patterns that aim at facilitating the construction of (a part of) a safety case from a formal model. In this work, we utilise the Event-B formalism due to its scalability and mature tool support. We illustrate the proposed methodology by numerous small examples as well as validate it by a larger case study – a steam boiler control system.

Keywords: safety-critical software systems, safety requirements, formal development, formal verification, Event-B, safety cases, argument patterns.

TUCS Laboratory
Embedded Systems Laboratory

1 Introduction

Safety-critical software systems are subject to certification. More and more standards in different domains require construction of *safety cases* as a part of the safety assurance process of such systems, e.g., ISO 26262 [40], EN 50128 [25], and UK Defence Standard [19]. Safety cases are justification arguments for safety. They justify why a system is safe and whether the design adequately incorporates the safety requirements defined in a system requirement specification to comply with the safety standards. To facilitate the construction of safety cases, two main graphical notations have been proposed: *Claims, Arguments and Evidence (CAE)* notation [17] and *Goal Structuring Notation (GSN)* [44]. In our work, we rely on the latter one due to its support for *argument patterns*, i.e., common structures capturing successful argument approaches that can be reused within a safety case [45]. To demonstrate the compliance with the safety standards, different types of evidence can be used [51]. Among them are results of hazard analysis, testing, simulation, formal verification, manual inspection, etc.

At the same time, the use of *formal methods* is highly recommended for certification of safety-critical software systems [36]. Safety cases constructed using formal methods give us extra assurance that the desired safety requirements are satisfied. There are several works dedicated to show how formal proofs can contribute to a safety case, e.g., [8–10, 21, 43]. For instance, such approaches as [8, 10] apply formal methods to ensure that different types of safety properties of critical systems hold while focusing on particular blocks of software system implementation (C code). The authors of [21] propose a generic approach to automatic transformation of the formal verification output into a software safety assurance case. Similarly to [8, 10], a formalised safety requirement in [21] is verified to hold at a specific location (a specific line number for code, a file, etc.).

In our work, we deal with formal system models rather than the code. A high level of abstraction allows us to cope with complexity of systems yet ensuring the desired safety properties. We rely on formal modelling techniques, including external tools that can be used together, that are scalable to analyse the entire system. Our chosen formal framework is Event-B [4] – a state-based formal method for system level modelling and verification. Event-B aims at facilitating modelling of parallel, distributed and reactive systems. Scalability in Event-B can be achieved via abstraction, proof and decomposition. Moreover, this formalism has strictly defined semantics and mature tool support – the Rodin platform [26] accompanied by various plug-ins, including the ones for program code generation, e.g, C, Java, etc. This allows us to model and verify a wide range of different safety-related properties stipulated by the given system safety requirements. Those requirements may include the safety requirements about global and local system properties, the absence of system deadlocks, temporal and timing properties, etc.

In this paper, we significantly extend and exemplify with a large case study our approach to linking modelling in Event-B with safety cases presented in [54]. More specifically, we further elaborate on the classification of safety requirements and define how each class can be treated formally to allow for verification of the given safety requirements, i.e., we define *mapping* of the classified safety requirements into the corresponding elements of Event-B.

The Event-B semantics then allows us to associate them with particular theorems (proof obligations) to be proved when verifying the system. The employed formal framework assists the developers in automatic generation of the respective proof obligations. This allows us to use the obtained proofs as the evidence in safety cases, demonstrating that the given safety requirements have been met. Finally, to facilitate the construction of safety cases, we define a set of argument patterns where the argumentation and goal decomposition in safety cases are based on the results obtained from the associated formal reasoning.

Therefore, the overall contribution of this paper is a developed methodology that covers two main processes: (1) integration of formalised safety requirements into formal models of software systems, and (2) construction of structured safety cases [1] from such formal models.

The remainder of the paper is organised as follows. In Section 2, we briefly introduce our modelling framework – Event-B, its refinement-based approach to modelling software systems as well as the Event-B verification capabilities based on theorem proving. Additionally, we overview the notion of safety cases and their supporting graphical notation. In Section 3, we describe our methodology and provide the proposed classification of safety requirements. We elaborate on the proposed methodology in Section 4, where we define a set of argument patterns and their verification support. In Section 5, we illustrate application of the proposed patterns on a larger case study – a steam boiler control system. In Section 6, we overview the related work. Finally, in Section 7, we give concluding remarks as well as discuss our future work.

2 Preliminaries

In this section, we briefly outline the Event-B formalism that we use to derive models of safety-critical systems. In addition, we briefly describe the notion of safety cases and their supporting notation that we will rely on in this paper.

2.1 Overview of Event-B

Event-B language. Event-B [4, 26] is a state-based formal method for system level modelling and verification. It is a variation of the B Method [2]. Automated support for modelling and verification in Event-B is provided by the Rodin platform [26].

Formally, an Event-B model is defined by a tuple $(d, c, A, v, \Sigma, I, Init, E)$, where d stands for sets (data types), c are constants, v is a vector of model variables, Σ corresponds to a model state space defined by all possible values of the vector v. $A(d, c)$ is a conjunction of axioms defining properties of model data structures, while $I(d, c, v)$ is a conjunction of invariants defining model properties to be preserved. $Init$ is an non-empty set of model initial states, $Init \subseteq \Sigma$. Finally, E is a set of model *events* where each event e is a relation of the form $e \subseteq \Sigma \times \Sigma$.

The sets and constants of the model are stated in a separate component called CONTEXT, where their properties are postulated as axioms. The model variables, invariants and events,

[1] From now on, by safety cases we mean structured safety cases.

including initialisation event, are introduced in the component called MACHINE. The model variables are strongly typed by the constraining predicates in terms of invariants.

In general, an event e has the following form:

$$e \mathrel{\hat{=}} \textbf{any } lv \textbf{ where } g \textbf{ then } R \textbf{ end},$$

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

The event guard defines 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. In general, the action of an event is a composition of assignments executed simultaneously. Variable assignments can be either deterministic or non-deterministic. The deterministic assignment is denoted as $x := Expr(v)$, where x is a state variable and $Expr(v)$ is an expression over the state variables v. The non-deterministic assignment can be denoted as $x :\in S$ or $x :| \ Q(v, x')$, where S is a set of values and $Q(v, x')$ is a predicate. As a result of the non-deterministic assignment, x gets any value from S or it obtains a value x' such that $Q(v, x')$ is satisfied.

The Event-B language can also be extended by different kinds of attributes attached to model events, guards, variables, etc. We will use Event-B attributes to contain formulas or expressions to be used by external tools or Rodin plug-ins, e.g., *Linear Temporal Logic (LTL)* formulas to be checked.

Event-B semantics. The semantics of Event-B events is defined using before-after predicates [50]. A *before-after predicate (BA)* describes a relationship between the system states before and after execution of an event. Hence, the definition of an event presented above can be given as the relation describing the corresponding state transformation from v to v', such that:

$$e(v, v') = g_e(v) \wedge I(v) \wedge BA_e(v, v'),$$

where g_e is the guard of the event e, BA_e is the before-after predicate of this event, and v, v' are the system states before and after event execution respectively.

Sometimes, we need to explicitly reason about possible model states before or after some particular event. For this purpose, we introduce two sets – *before*(e) and *after*(e). Specifically, *before*(e) represents a set of all possible pre-states defined by the guard of the event *e*, while *after*(e) is a set of all possible post-states of the event *e*, i.e., *before*$(e) \subseteq \Sigma$ and *after*$(e) \subseteq \Sigma$ denote the domain and range of the relation e [37]:

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

To verify correctness of an Event-B model, we generate a number of *proof obligations* (*POs*). More precisely, for an initial (i.e., abstract) model, we prove that its initialisation and all events preserve the invariant:

$$A(d, c),\ I(d, c, v),\ g_e(d, c, v),\ BA_e(d, c, v, v') \ \vdash \ I(d, c, v'). \qquad \text{(INV)}$$

Since the initialisation event has no initial state and guard, its proof obligation is simpler:

$$A(d,c),\ BA_{Init}(d,c,v')\ \vdash\ I(d,c,v'). \tag{INIT}$$

On the other hand, we verify event feasibility. Formally, for each event e of the model, its feasibility means that, whenever the event is enabled, its before-after predicate is well-defined, i.e., there exists some reachable after-state:

$$A(d,c),\ I(d,c,v),\ g_e(d,c,v)\ \vdash\ \exists\, v' \cdot BA_e(d,c,v,v'). \tag{FIS}$$

Refinement 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 (i.e., an abstract machine) and continues with stepwise unfolding of system properties by introducing new variables and events into the model (i.e., refinements). This type of a refinement is known as 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 discharge a number of POs for a refined model. For brevity, here we show only essential ones. The full list of POs can be found in [4].

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

When refining an event, its guard can only be strengthened:

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

where g_e, g'_e are respectively the abstract and concrete guards of the event e.

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

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

where BA_e, BA'_e are respectively the abstract and concrete before-after predicates of the same event e, w and w' are the concrete variable values before and after this event execution.

All the described above proof obligations are automatically generated by the Rodin platform [26] 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 in proving (usually over 80% of POs are proved automatically).

Verification via theorem proving. Additionally, the Event-B formalism allows the developers to formulate theorems either in the model CONTEXT or MACHINE components. In the

first case, theorems are logical statements about model static data structures that are provable (derivable) from the model axioms given in the CONTEXT component. In the latter case, these are logical statements about model dynamic properties that follow from the given formal definitions of the model events and invariants.

The *theorem* proof obligation (THM) indicates that this is a theorem proposed by the developers. Depending whether a theorem is defined in the CONTEXT or MACHINE components, it has a slightly different form. To highlight this difference, we use indexes C and M in this paper. The first variant of a proof obligation is defined for a theorem $T(d, c)$ in the CONTEXT component:

$$A(d, c) \ \vdash \ T(d, c). \qquad (\text{THM}_C)$$

The second variant is defined for a theorem $T(d, c, v)$ in the MACHINE component:

$$A(d, c), I(d, c, v) \ \vdash \ T(d, c, v). \qquad (\text{THM}_M)$$

2.2 Safety cases

A *safety case* is *"a structured argument, supported by a body of evidence that provides a convincing and valid case that a system is safe for a given application in a given operating environment"* [13, 19]. The construction, review and acceptance of safety cases are the valuable steps in safety assurance process of critical software systems. Several standards, e.g., ISO 26262 [40] for the automotive domain, EN 50128 [25] for the railway domain, and the UK Defence Standard [19], prescribe production and evaluation of safety (or more generally assurance) cases for certification of such critical systems [31].

In general, safety cases can be documented either textually or graphically. However, a growing number of industrial companies working with safety-critical systems adopt a graphical notation, namely *Goal Structuring Notation (GSN)* proposed by Kelly [44], in order to present safety arguments within safety cases [30]. GSN aims at graphical representation of safety case elements as well as the relationships that exist between these elements. The principal building blocks of the GSN notation are shown in Figure 1. Essentially, a safety case constructed using GSN consists of *goals*, *strategies* and *solutions*. Here *goals* are propositions in an argument that can be said to be true or false (e.g., claims of requirements to be met by a system). *Solutions* contain the information extracted from analysis, testing or simulation of a system (i.e., evidence) to show that the goals have been met. Finally, *strategies* are reasoning steps describing how goals are decomposed and addressed by sub-goals.

Thus, a safety case constructed in the GSN notation presents decomposition of the given safety case goals into sub-goals until they can be supported by the direct evidence (a solution). It also explicitly defines the argument strategies, relied assumptions, the context in which goals are declared, as well as justification for the use of a particular goal or strategy. If the contextual information contains a model, a special GSN symbol called *model* can be used instead of a regular GSN context element.

The elements of a safety case can be in two types of relationships: *"Is solved by"* and *"In context of"*. The former is used between goals, strategies and solutions, while the latter

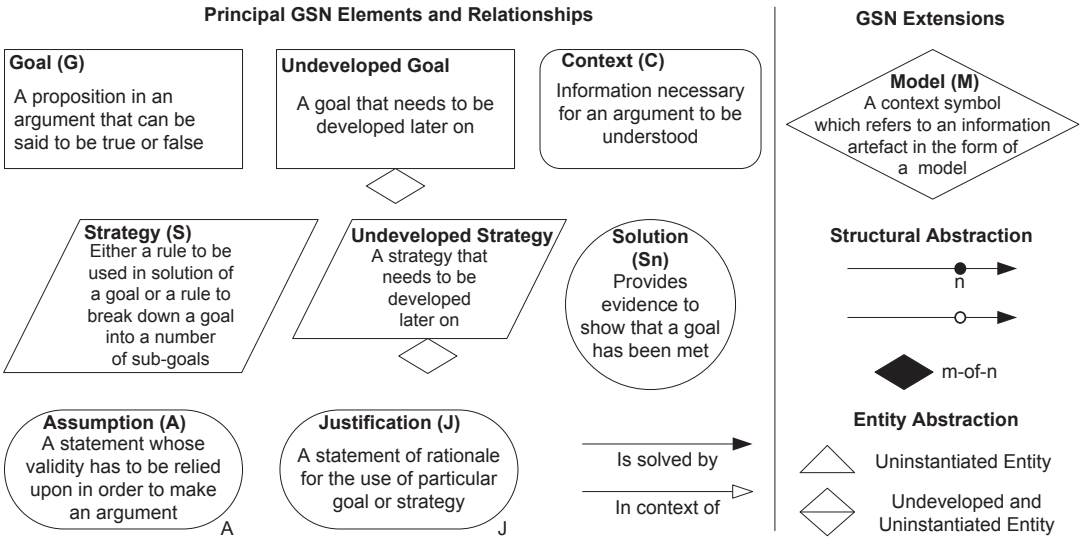

Figure 1: Elements of GSN (detailed description is given in [7, 28, 44, 45])

links a goal to a context, a goal to an assumption, a goal to a justification, a strategy to a context, a strategy to an assumption, a strategy to a justification.

To allow for construction of argument patterns, GSN has been extended to represent generalised elements [44, 45]. We utilise the following elements from the extended GSN for structural abstraction of our argument patterns: *multiplicity* and *optionality*. Multiplicity is a generalised n-ary relationship between the GSN elements, while optionality stands for optional and alternative relationship between the GSN elements. Graphically, the former is represented as a solid ball or a hollow ball on an arrow *"Is solved by"* shown in Figure 1, where the label *n* indicates the cardinality of a relationship, while a hollow ball means zero or one. The latter is depicted as a solid diamond in Figure 1, where *m-of-n* denotes a possible number of alternatives. The multiplicity and the optionality relationships can be combined. If a multiplicity symbol is placed in front of the optionality symbol, this stands for a multiplicity over all the options.

There are two extensions for entity abstraction in GSN: (1) *uninstantiated entity*, and (2) *undeveloped and uninstantiated entity*. The former one specifies that the entity requires to be instantiated, i.e., the "abstract" entity needs to be replaced with a more concrete instance later on. In Figure 1, the corresponding annotation is depicted as a hollow triangle. It can be used with any GSN element. The latter one indicates that the entity needs both further development and instantiation. In Figure 1, it is shown as a hollow diamond with a line in the middle. This annotation can be applied to GSN goals and strategies only.

3 Methodology

In this section, we describe our methodology that aims at establishing a link between formal verification of safety requirements in Event-B and the construction of safety cases.

3.1 General methodology

In this work, we contribute to the process of development, verification and certification of software systems by showing how to proceed from the given safety requirements to safety cases via formal modelling and verification in Event-B (Figure 2). We distinguish two main processes: (1) representation of formalised safety requirements in Event-B models, and (2) derivation of safety cases from the associated Event-B specifications. Let us point out that these activities are tightly connected to each other. Accuracy of the safety requirements formalisation influences whether we are able to construct a safety case sufficient to demonstrate safety of a system. This dependence is highlighted in Figure 2 as a dashed line. If a formal specification is not good enough, we need to return and improve it.

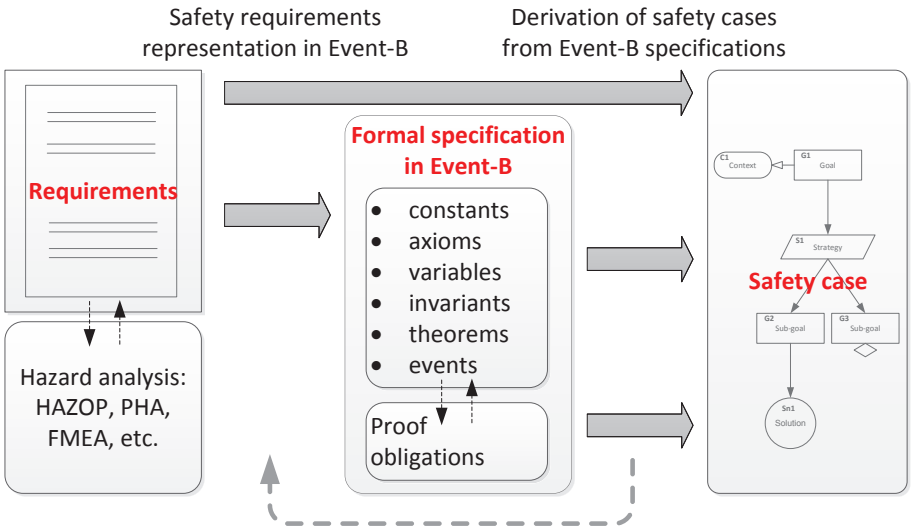

Figure 2: High-level representation of the overall approach

We connect these two processes via classification of safety requirements. On the one hand, we propose a specific classification associated with particular ways these requirements can be represented in Event-B. On the other hand, we propose a set of classification-based argument patterns to facilitate the construction of safety cases from the associated Event-B models. The classification includes separate classes for safety requirements about global and local system properties, the absence of system deadlock, temporal and timing properties, etc. We are going to present this classification in detail in Section 3.2.

In this paper, we leave out of the scope the process of elicitation of system safety requirements. We assume that the given list of these requirements is completed beforehand by applying well-known hazard analysis techniques such as HAZard and OPerability (HAZOP) analysis, Preliminary Hazard Analysis (PHA), Failure Modes and Effects Analysis (FMEA), etc.

Incorporating safety requirements into formal models. Each class of safety requirements can be treated differently in an Event-B specification (model). In other words, various model expressions based on model elements, e.g., axioms, variables, invariants, events, etc., can

be used to formalise a considered safety requirement. Consequently, the argument strategies and resulting evidence in a safety case built based on such a formal model may also vary. Using the defined classification, we provide the reader with the precise guidelines on how to map safety requirements of some class into a particular subset of model elements. Moreover, we define how to construct from these model elements a specific theorem to be verified. Later on, we will show how the verification results (e.g., discharged proof obligations and model checking results) can be used as the evidence in the associated safety cases.

Our methodology allows us to cope with two cases: (1) when a formal Event-B specification of the system under consideration has been already developed, and (2) when it is performed simultaneously with the safety case construction. In the first case, we assume that adequate models are constructed and linked with the classification we propose. In the second case, the formal development is guided by our proposed classification and methodology. Consequently, both ways allow us to contribute towards obtaining adequate safety cases.

Constructing safety cases from formal models. Model-based development in general and development using formal methods in particular typically require additional argumentation about model correctness and well-definedness [6]. In this paper, we address this challenge and provide the corresponding argument pattern as shown in Section 4.1.

Having a well-defined classification of safety requirements benefits both stages of the proposed methodology, i.e., while incorporating safety requirements into formal models and while deriving safety cases from such formal models. To simplify the task of linking the formalised safety requirements with the safety case to be constructed, we propose a set of classification-based argument patterns (Sections 4.2-4.9). The patterns have been developed using the corresponding GSN extensions (Figure 1). Some parts of an argument pattern may remain the same for any instance, while others need to be further instantiated (they are labelled with a specific GSN symbol – a hollow triangle). The text highlighted by braces { } should be replaced by a concrete value.

The generic representation of a classification-based argument pattern is given in Figure 3. Here, a safety requirement *Requirement* of some class *Class* $\{X\}$ is reflected in the goal **GX**, where X is a class number (see the next section for the reference). According to the proposed approach, the requirement is verified within a formal model M in Event-B (the model element **MX.1**).

In order to obtain the evidence that a specific safety requirement is met, different construction techniques might be undertaken. The choice of a particular technique influences the argumentation strategies to be used in each pattern. For example, if a safety requirement can be associated with a model invariant property, the corresponding theorem for each event in the model M is required to be proved. Correspondingly, the proofs of these theorems are attached as the evidence for the constructed safety case.

The formulated properties and theorems associated with a particular requirement can be automatically derived from the given formal model. Nonetheless, to increase clarity of a safety case, any theorem or property whose verification result is provided as a solution of the top goal needs to be referred to in the GSN context element (**CX.2** in Figure 3).

To bridge a semantic gap in the mapping associating an informally specified safety re-

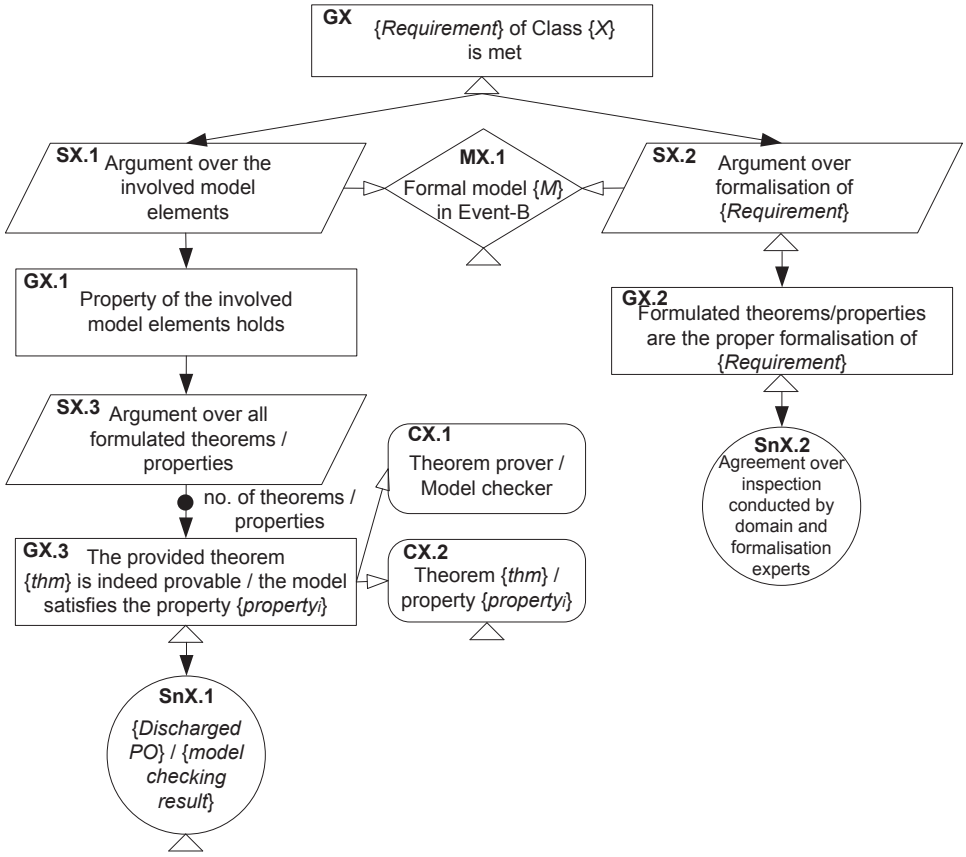

Figure 3: Generic argument pattern

quirement with the corresponding formal expression that is verified and connected to evidence, we need to argue over a correct formalisation of the requirement (**SX.2** in Figure 3). We rely on a joint inspection conducted by domain and formalisation experts (**SnX.2**) as the evidence that the formulated theorems/properties are proper formalisations of the requirement.

Generating code. Additionally, the most detailed (concrete) specification obtained during the refinement-based development can be used for code generation. The Rodin platform, Event-B tool support, allows for program code generation utilising a number of plug-ins. One of these plug-ins, EB2ALL [24], automatically generates a target programming language code from an Event-B formal specification. In particular, EB2C allows for generation of C code, EB2C++ supports C++ code generation, using EB2J one can obtain Java code, and using EBC# – C# code. The alternative solution is to use the constructed formal specification for verification of an already existing implementation code. Then, if the code has successfully passed such a verification, the existing safety case derived from the formal specification implies the code safety for the verified safety properties. Nonetheless, in both cases a safety case based on formal analysis cannot be used solely. It requires additional argumentation, for example, over the correctness of the code generation process itself [30, 43].

3.2 Requirements classification and its mapping into Event-B elements

To classify safety requirements, we have firstly adopted the taxonomy proposed by Bitsch [15] as presented in our previous work [54]. However, the Bitsch's approach uses *Computational Tree Logic (CTL)* to specify the requirements and relies on model checking as a formal verification technique. The differences between the semantics of CTL and Event-B significantly restrict the use of the Bitsch's classification in the Event-B framework. As a result, we extensively modified the original classification. In this paper, we propose the following classification of safety requirements, as shown in Figure 4.

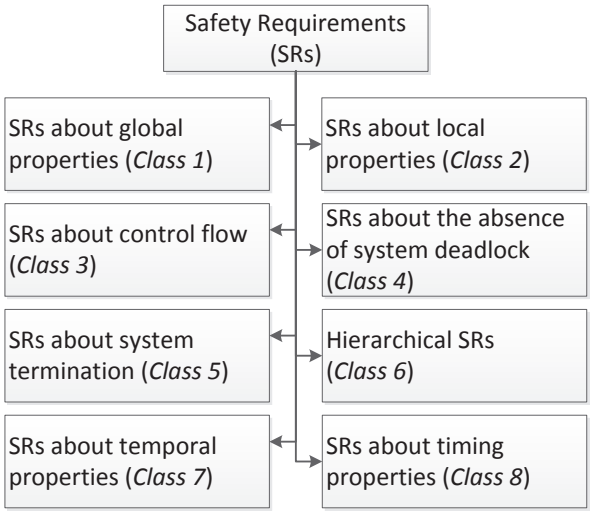

Figure 4: Classification of safety requirements

We divide safety requirements (*SRs*) into eight classes:

- *Class 1*: *SRs about global properties* are the requirements stipulating the system safety properties that must be always maintained by the modelled system;
- *Class 2*: *SRs about local properties* are the requirements that reflect the necessity of some property to be true at a specific system state;
- *Class 3*: *SRs about control flow* are the requirements that define the necessary flow (order) in occurrences of some system events;
- *Class 4*: *SRs about the absence of system deadlock* are the requirements related to a certain class of control systems where an unexpected stop of the system may lead to a safety-related hazard;
- *Class 5*: *SRs about system termination* are the requirements related to a certain class of control systems where non-termination of the system in a specific situation may lead to a safety-related hazard;
- *Class 6*: *Hierarchical SRs* are the requirements that are hierarchically structured to deal with the complexity of the system, i.e., a more general requirement may be decomposed into several more detailed ones;

- *Class 7*: *SRs about temporal properties* are the requirements that describe the properties related to reachability of specific system states;

- *Class 8*: *SRs about timing properties* are the requirements that establish timing constraints of a system, for example, of a safety-critical real-time system where the response time is crucial.

The given classes of SRs are represented differently in a formal model. For instance, SRs of *Class 1* are modelled as invariants in the MACHINE component, while SRs of *Class 2* are modelled by defining a theorem about the required post-state of a specific Event-B model event. However, in some cases requirements of *Class 2* can be also formalised as requirements of *Class 1* by defining implicative invariants, i.e., invariants that hold in specific system states. The SRs about control flow *(Class 3)* can be expressed as event-flow properties (e.g., by using Event-B extension – the graphical Usecase/Flow language [37]). The SRs about the absence of system deadlock *(Class 4)* are represented as deadlock freedom conditions, while the SRs of *Class 5* are modelled as shutdown conditions. In both cases, these conditions are turned into specific model theorems to be proved. The class of hierarchical SRs *(Class 6)* is expressed within Even-B based on refinement between the corresponding Event-B models. Finally, the associated ProB tool for the Rodin platform [52] allows us to support the SRs of *Class 7* by model checking.

Let us note however that the representation of timing properties *(Class 8)* in the Event-B framework is a challenging task. There are several works dedicated to address this issue [12, 18, 39, 58]. In this paper, we adopt the approach that establishes a link between timing constraints defined in Event-B and verification of real-time properties in Uppaal [39].

Formally, the described above relationships can be defined as a function F_M mapping safety requirements (*SRs*) into a set of the related model expressions:

$$\textit{SRs} \rightarrow \mathcal{P}\textit{(MExpr)},$$

where $\mathcal{P}(T)$ corresponds to a power set on elements of T and *MExpr* stands for a generalised type for all possible expressions that can be built from the model elements, i.e., *model expressions*. Here *model elements* are elements of Event-B models such as *axioms*, *variables*, *invariants*, *events*, and *attributes*. *MExpr* includes such model elements as trivial (basic) expressions. Among other possible expressions of this type are *state predicates* defining post-conditions and shutdown conditions, *event control flow* expressions as well as *Linear Temporal Logic (LTL)* and *Timed Computation Tree Logic (TCTL)* formulas based on elements of the associated Event-B model.

The defined strict mapping allows us to trace the safety requirements given in an informal manner into formal specifications in Event-B as well as into the accompanying means for verification, i.e., the Flow and ProB plug-ins and Uppaal. In Figure 5, we illustrate the steps of evidence construction in our proposed approach. Firstly, we map a safety requirement into a set of model expressions. Secondly, we construct a specific theorem or a set of theorems out of these model expressions, thus essentially defining the semantics of the formalised requirement. Finally, we prove each theorem using the theorem provers of Event-B or perform model checking using, e.g., Event-B extension ProB. As a result, we obtain

either a discharged proof obligation or a result of model checking. We include such results into the fragment of a safety case corresponding to the considered safety requirement as the evidence that this requirement holds. Table 1 illustrates the correspondence between safety requirements of different classes, model expressions and constructed theorems.

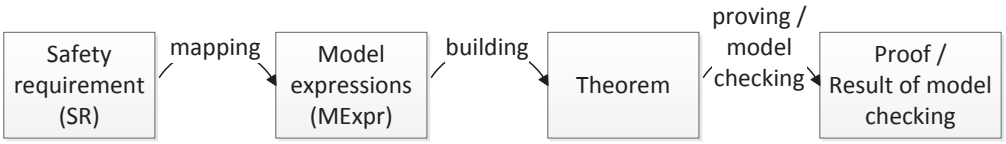

Figure 5: Steps of evidence construction

Table 1: Formalisation of safety requirements

Safety requirement	Model element expressions	Theorem
SR of Cl. 1	invariants	group of theorems for each event $Event_k$/*safety*$_i$/*INV*
SR of Cl. 2	event, state predicate	theorem about a specific post-state of an event thm_ap/*THM*
SR of Cl. 3	pairs of events, event control flow	group of theorems about enabling relationships between events, e.g., $Event_i/Event_j$/*FENA*
SR of Cl. 4	all events	theorem about the deadlock freedom thm_dlf/*THM*
SR of Cl. 5	state predicate, all events	theorem about a shutdown condition thm_shd/*THM*
SR of Cl. 6	abstract event, concrete event(s)	theorem about guard strengthening $Event'_k$/*grd*/*GRD*, theorem about action simulation $Event'_k/act$/*SIM*
SR of Cl. 7	LTL formula	*LTL* $property_i$
SR of Cl. 8	TCTL formula	*TCTL* $property_j$

As soon as all safety requirements are assigned to their respective classes and their mapping into Event-B elements is performed, we can construct the part of a safety case corresponding to assurance of these requirements. We utilise GSN to graphically represent such a safety case.

4 Argument patterns

In this section, we present the argument patterns corresponding to each of the introduced classes. In addition, to obtain an adequate safety case, we need to demonstrate well-definedness of the formal models we rely on. Therefore, we start this section by presenting a specific argument pattern to address this issue.

4.1 Argumentation that formal development of a system is well-defined

We propose the argument pattern shown in Figure 6 in order to provide evidence that the proposed formal development of a system is well-defined. To verify this (e.g., that a partial function is applied within its domain), Event-B defines a number of proof obligations (*well-definedness (WD)* for theorems, invariants, guards, actions, etc. and *feasibility (FIS)* for events [4]), which are automatically generated by the Rodin platform. We assume here that all such proof obligations are discharged for models in question. However, if model axioms are inconsistent (i.e., contradictory), the whole model becomes fallacious and thus logically meaningless. Demonstrating that this is not the case is a responsibility of the developer. To handle this problem, we introduce a specific argument pattern shown in Figure 6. In Event-B, well-definedness of a CONTEXT can be ensured by proving axiom consistency (the goal **G1.2** in Figure 6).

We propose to construct a theorem showing axiom consistency and prove it. However,

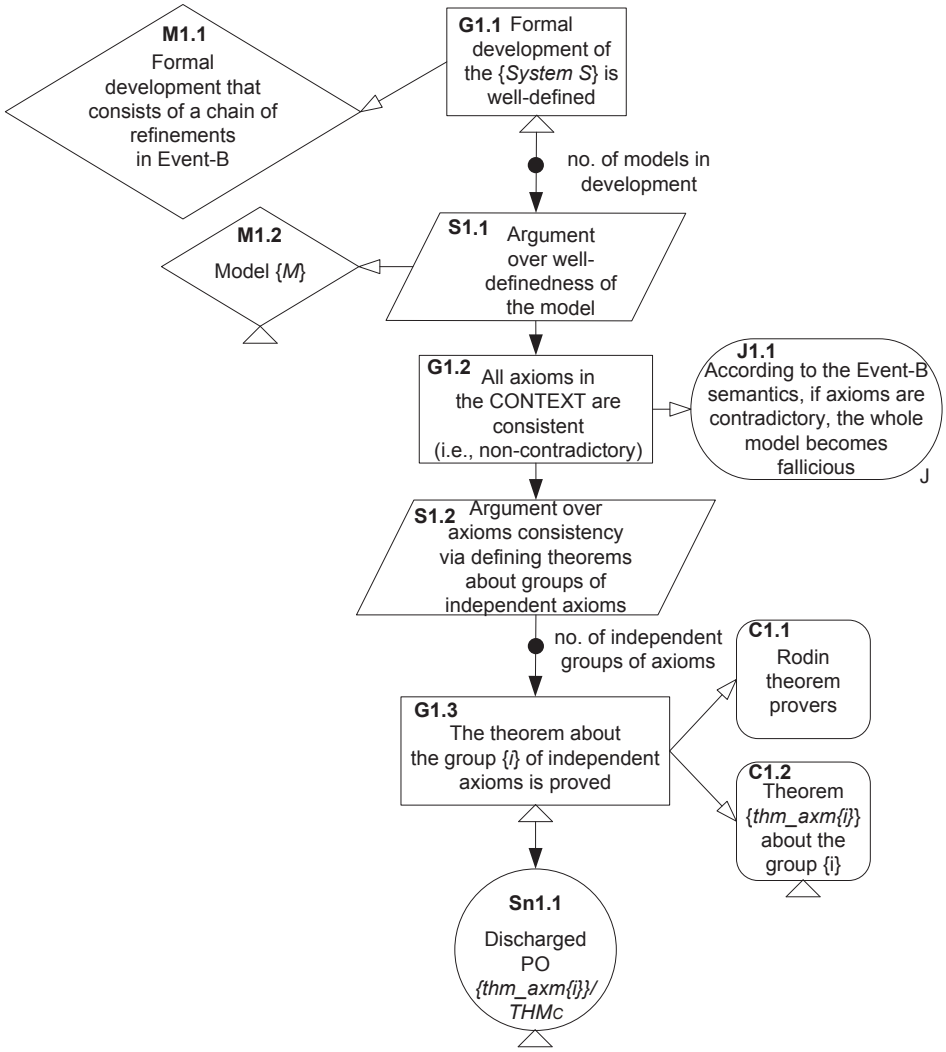

Figure 6: Argument pattern for well-definedness of formal development

such a theorem could be very large in size. Thus, for simplicity, we suggest to divide axioms into groups, where each group consists of axioms that use shared constants and sets. In other words, each group of axioms is independent from each other. Consequently, we define theorems for all groups of independent axioms (the strategy **S1.2**) as shown below:

$$\textit{thm\_axm}\{i\}: \quad A(d,c) \vdash \exists\, d,c \cdot A_1(d,c) \wedge \ldots \wedge A_N(d,c),$$

where i stands for *i-th* group of axioms such that $i \in 1 \,..\, K$ and K is the number of independent groups of axioms. The number of axioms in a group is represented by N. The generated proof obligation (**Sn1.1**) shown in Figure 6 is an instance of the (THM_C) proof obligation given in Section 2.1.

In order to instantiate this pattern for each model in the development,

- a formal development that consists of a chain of refinements in Event-B should be defined in a GSN model element;
- a formal model *M*, for which a particular fragment of the safety case is constructed, should be referred to in a GSN model element;
- theorems about the defined groups of independent axioms should be formulated using the Event-B formal language and referred to in GSN context elements;
- the proof obligations of the type THM_C discharged by the Rodin platform should be included as solutions of the goal ensuring consistency of model axioms.

The instantiation example for this fragment of the safety case can be found in Section 5.3.1.

In the remaining part of Section 4, we introduce the argument patterns that correspond to each class of safety requirements proposed in Section 3.2. It is not necessarily the case that the final safety case of the modelled system will include SRs of all the classes. Moreover, it is very common for the Event-B practitioners to limit the requirements model representation to invariants, theorems and operation refinement [41, 49, 59]. However, to achieve a strong safety case, the developers need to provide the evidence that all the safety requirements listed in the requirements document hold. The proposed argument patterns cover a broader range of safety requirements, including also those that specify temporal and timing properties which cannot be formalised in Event-B directly.

4.2 Argument pattern for SRs about global properties (Class 1)

In this section, we propose an argument pattern for the safety requirements stipulating global safety properties, i.e., properties that must be maintained by the modelled system during its operation (Figure 7).

We assume that there is a model *M*, which is a part of the formal development of a system in Event-B, where a safety requirement of *Class 1* is verified to hold (**M2.1.1**). In addition, we assume that the model invariant $I(d,c,v)$ contains the conjuncts $\textit{safety}_1$, ..., $\textit{safety}_N$, where *N* is the number of safety invariants, which together represent a proper formalisation of the considered safety requirement (**A2.1.1**) [2]. Then, for each safety invariant $\textit{safety}_i$, $i \in$

[2]Such an assumption can be substantiated by arguing over formalisation of the requirements as demonstrated in Figure 3 (the strategy **SX.2**). It is applicable to all the classification-based argument patterns and their instances.

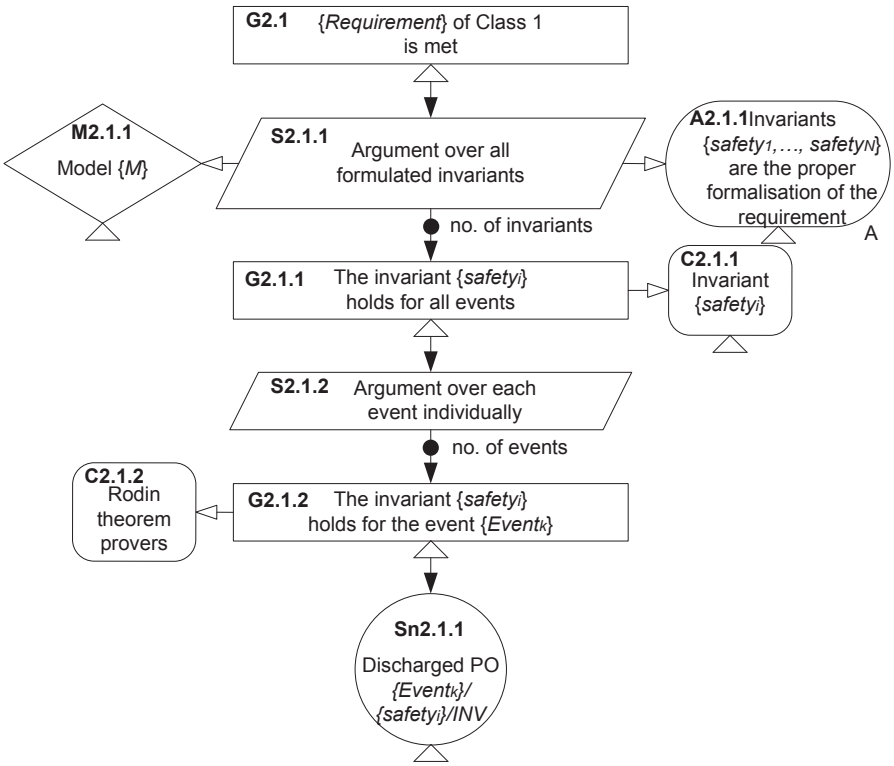

Figure 7: Argument pattern for safety requirements of Class 1

1..N, the event $Event_k$, $k \in 1..K$, where *K* is the number of all model events, represents some event for which this invariant must hold.

We build the evidence for the safety case in the way illustrated in Figure 5. Thus, for each model expression, in this case, an invariant, formalising the safety requirement, we construct a separate fragment of the safety case. Then, a separate theorem is defined for each event where a particular invariant should hold. In other words, we define a group of theorems, one per each event. The number of model events influences the number of branches into which the goal **G2.1.1** is split. In a special case when the set of variables referred in an invariant is mutually exclusive with the set of variables modified by an event, such an event can be excluded from the list of events because the theorem generated for such an event is trivially true.

According to our approach, the generic mapping function F_M is of the form $SRs \rightarrow \mathcal{P}(MExpr)$. In general case, for each requirement of this class the function returns a set of invariants $\{safety_1, ..., safety_N\}$ that can be represented as a conjunction. Due to this fact, each such an invariant can be verified independently. The theorem for verification that the safety invariant $safety_i$ (denoted by $I(d, c, v')$) holds for the event $Event_k$ is as follows:

$$A(d, c),\ I(d, c, v),\ g_{Event_k}(d, c, v),\ BA_{Event_k}(d, c, v, v') \vdash I(d, c, v'). \qquad \text{(INV)}$$

The Rodin platform allows us to prove this theorem using the integrated theorem provers and explicitly support the safety case to be built with the discharged proof obligations of the type INV for each event where the safety invariant has been verified to hold.

The key elements of the pattern to be instantiated are as follows:

- a requirement *Requirement* should be replaced with a particular safety requirement;
- a formal model *M* should be referred to in a GSN model element;
- the concrete mapping between the requirement and the corresponding model invariants should be provided, while the invariants *safety*$_1$, ..., *safety*$_N$ formalising the requirement from this mapping should be referred to in GSN context elements;
- the proof obligations of the type INV discharged by the Rodin platform should be included in the safety case as the respective solutions.

Let us consider instantiation of the proposed pattern by an example – a sluice gate control system [46]. The system is a sluice connecting areas with dramatically different pressures. The purpose of the system is to adjust the pressure in the sluice area and operate two doors (*door1* and *door2*) connecting outside and inside areas with the sluice area. To guarantee safety, a door may be opened only if the pressure in the locations it connects is equalized, namely

SR-cl1-ex1: *When the door1 is open, the pressure in the sluice area is equal to the pressure outside*;
SR-cl1-ex2: *When the door2 is open, the pressure in the sluice area is equal to the pressure inside*.

These safety requirements are formalised in the Event-B model (available from Appendix C, Refinement 2 of [46]) as the invariants **inv_cl1_ex1** and **inv_cl1_ex2** such that

$$\textbf{SR-cl1-ex1} \mapsto \{\textbf{inv\_cl1\_ex1}\},$$
$$\textbf{SR-cl1-ex2} \mapsto \{\textbf{inv\_cl1\_ex2}\},$$

where:

$$\textbf{inv\_cl1\_ex1:}\ \mathit{failure} = \mathit{FALSE} \wedge (\mathit{door1\_position} > 0 \vee \mathit{door1\_motor} = \mathit{MOTOR\_OPEN}) \Rightarrow \mathit{pressure\_value} = \mathit{PRESSURE\_OUTSIDE},$$
$$\textbf{inv\_cl1\_ex2:}\ \mathit{failure} = \mathit{FALSE} \wedge (\mathit{door2\_position} > 0 \vee \mathit{door2\_motor} = \mathit{MOTOR\_OPEN}) \Rightarrow \mathit{pressure\_value} = \mathit{PRESSURE\_INSIDE}.$$

The expressions *doorX_position* $>$ *0* and *doorX_motor* = *MOTOR_OPEN* indicate that the corresponding door *X* (where X = 1 or X = 2) is open. The variable *door1* models a door that connects the sluice area with the outside area and the variable *door2* models a door that connects the sluice area with inside area. The variable *pressure_value* stands for the pressure in the sluice area.

Then, according to the proposed approach, we show that these invariants hold for all events in the model. Due to the space limit, we give only an excerpt of the safety case that corresponds to the safety requirement **SR-cl1-ex1** (Figure 8). The associated invariant affects a number of model events, including such as *pressure_high* (changing pressure to high) and *closed2* (closing the door 2). From now on, we will hide a part of the safety case by three dots to avoid unnecessary big figures in the paper. We assume that the given part of the safety case is clear and can be easily repeated for the hidden items.

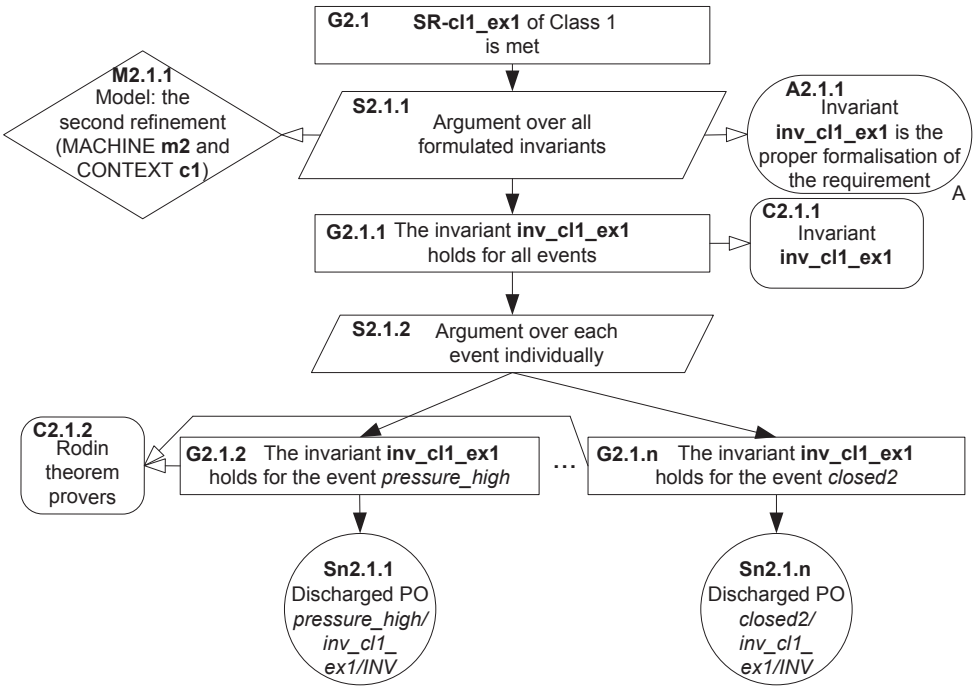

Figure 8: The pattern instantiation example

4.3 Argument pattern for SRs about local properties (Class 2)

Safety requirements of *Class 2* describe local properties, i.e., the properties that need to be true at specific system states. For example, in case of a control system relying on the notion of operational modes, a safety requirement of *Class 2* may define a (safety) mode which the system enters after the execution of some transition. In terms of Event-B, the particular system states we are interested in are usually associated with some desired post-states of specific model events.

Figure 9 shows the argument pattern for justification of a safety requirement of *Class 2*. As for *Class 1*, the key argumentation strategy here (**S2.2.1**) is defined by the steps of evidence construction illustrated in Figure 5. However, in contrast to the invariant theorems established and proved for each event in the model, the theorem formalising the safety requirement of *Class 2* is formulated and proved only once for the whole model *M*.

As mentioned above, local properties are usually expressed in Event-B in terms of post-states of specific model events. This suggests the mapping function F_M for *Class 2* to be of the form:

$$\textit{Requirement} \mapsto \{(e_1, q_1),\ ...,\ (e_K, q_S)\},$$

where the events e_1,..., e_K and the state predicates q_1,..., q_S are model expressions based on which the corresponding theorems are constructed. The number of such theorems reflects the number of branches of a safety case for the goal **G2.2** (Figure 9). Specifically, we can verify a safety requirement of *Class 2* by proving the following theorem for each pair (e_i, q_j), where $i \in 1..K$ and $j \in 1..S$:

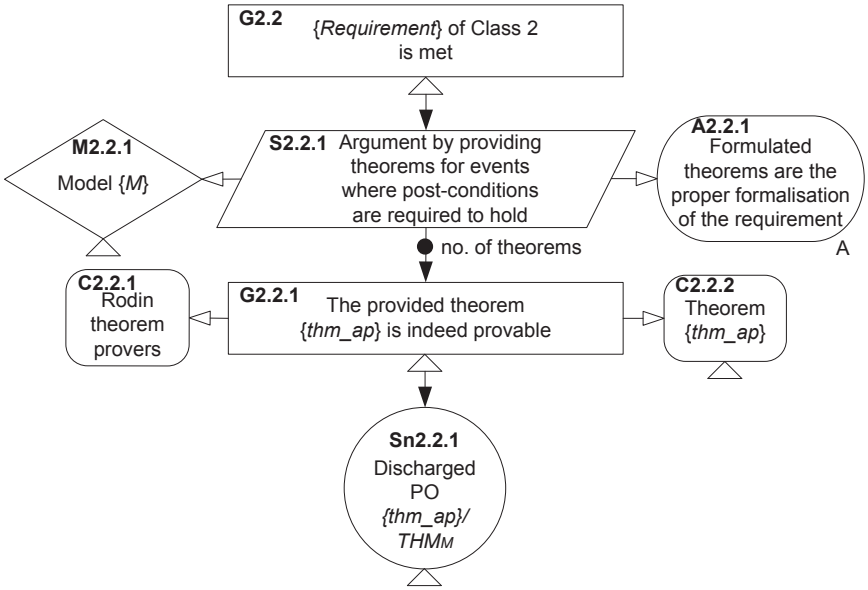

Figure 9: Argument pattern for safety requirements of Class 2

$$\textit{thm\_ap}: \quad A(d,c),\ I(d,c,v) \vdash \forall v' \cdot v' \in \textit{after}(e_i) \Rightarrow q_j(v').$$

Here $\textit{after}(e_i)$ is the set of all possible post-states of the event e_i as defined in Section 2.1.

This argument pattern (Figure 9) can be instantiated as follows:

- a requirement *Requirement* should be replaced with a particular safety requirement;
- a formal model *M* should be referred to in a GSN model element;
- the concrete mapping between the requirement and event-post-condition pairs should be supplied, while the theorems *thm_ap* obtained from this mapping should be referred to in GSN context elements;
- the proof obligations of the type THM_M discharged by the Rodin platform should be included in the safety case as the evidence supporting that the top-level claim (i.e., **G2.2**) holds.

In order to demonstrate application of this pattern, let us introduce another case study – Attitude and Orbit Control System (AOCS) [53]. The AOCS is a typical layered control system. The main function of this system is to control the attitude and the orbit of a satellite. Since the orientation of a satellite may change due to disturbances of the environment, the attitude needs to be continuously monitored and adjusted. At the top layer of the system there is a *mode manager (MM)*. The transitions between modes can be performed either to fulfil the predefined mission of the satellite (forward transitions) or to perform error recovery (backward transitions). Correspondingly, the MM component might be in either *stable*, *increasing* (i.e., in forward transition) or *decreasing* (i.e., in backward transition) state. As an example, let us consider the safety requirement

SR-cl2: *When a mode transition is completed, the state of the MM shall be stable.*

To verify this property on model events and variables, we need to prove that the corresponding condition q, namely

$$last\_mode = prev\_target \wedge next\_target = prev\_target,$$

holds after the execution of the event *Mode_Reached*. Here $prev\_target$ is the previous mode that a component was in transition to, $last\_mode$ is the last successfully reached mode, and $next\_target$ is the target mode that a component is currently in transition to. The event is enabled only when there is no critical error in the system, i.e., when the condition $error = No\_Error$ holds.

We represent the mapping of the shown safety requirement on Event-B as F_M such that **SR-cl2** $\mapsto \{(Mode\_Reached, q)\}$. According to *thm_ap* and the definition of *after(e)* given in Section 2.1, we can construct the theorem to be verified as follows:

thm_cl2_ex: $\forall last\_mode', prev\_targ', next\_targ' \cdot$
$(\exists next\_targ, error, prev\_targ \cdot$
$(next\_targ \neq prev\_targ \wedge error = No\_Error) \wedge$
$(last\_mode' = next\_targ \wedge prev\_targ' = next\_targ \wedge$
$next\_targ' = next\_targ))$
$\Rightarrow$
$last\_mode' = next\_targ \wedge prev\_targ' = next\_targ.$

Here, for simplicity, we omit showing types of the involved variables. The corresponding instance of the argument pattern is illustrated in Figure 10.

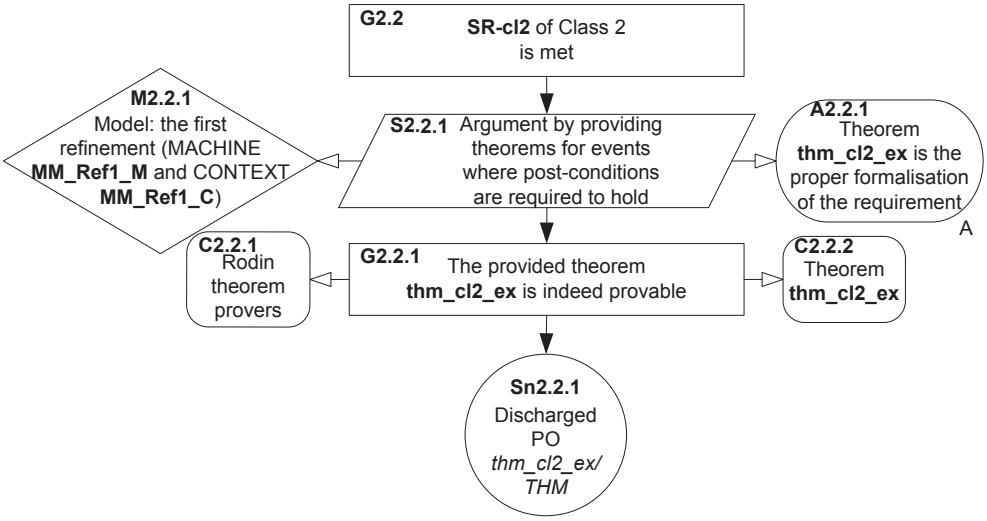

Figure 10: The pattern instantiation example

4.4 Argument pattern for SRs about control flow (Class 3)

In this section, we propose an argument pattern for the requirements that define the flow in occurrences of some system events, i.e., safety requirements about control flow. For instance,

this class may include certain requirements that define fault-tolerance procedures. Since fault detection, isolation and recovery actions are strictly ordered, we also need to preserve this order in a formal model of the system.

Formally, the ordering between system events can be expressed as a particular relationship amongst possible pre- and post-states of the corresponding model events. We consider three types of relationships proposed by Iliasov [37]: enabling (**ena**), disabling (**dis**) and possibly enabling (**fis**). In detail, enabling relationship between two events means that, when one event occurs, it is always true that the other one may occur next (i.e., the set of pre-states of the second event is included in the set of post-states of the first event). An event disables another event if the guard of the second event is always false after the first event occurs (i.e., the set of pre-states of the second event is excluded from the set of post-states of the first event). Finally, an event possibly enables another event if, after its occurrence, the guard of the second event is potentially enabled (i.e., there is a non-empty intersection of the set of pre-states of the second event with the set of post-states of the first event).

Let e_m and e_n be some events. Then, according to the usecase/flow approach [37], the proof obligations that support the relationships between these events can be defined as follows:

$$\begin{aligned} e_m \textbf{ ena } e_n \quad & \Leftrightarrow \mathit{after}(e_m) \subseteq \mathit{before}(e_n) \\ & \Leftrightarrow \forall v, v' \cdot I(v) \ \wedge \ g_{e_m}(v) \ \wedge \ BA_{e_m}(v, v') \ \Rightarrow \ g_{e_n}(v'), \end{aligned} \tag{FENA}$$

$$\begin{aligned} e_m \textbf{ dis } e_n \quad & \Leftrightarrow \mathit{after}(e_m) \cap \mathit{before}(e_n) = \varnothing \\ & \Leftrightarrow \forall v, v' \cdot I(v) \ \wedge \ g_{e_m}(v) \ \wedge \ BA_{e_m}(v, v') \ \Rightarrow \ \neg g_{e_n}(v'), \end{aligned} \tag{FDIS}$$

$$\begin{aligned} e_m \textbf{ fis } e_n \quad & \Leftrightarrow \mathit{after}(e_m) \cap \mathit{before}(e_n) \neq \varnothing \\ & \Leftrightarrow \exists v, v' \cdot I(v) \ \wedge \ g_{e_m}(v) \ \wedge \ BA_{e_m}(v, v') \ \wedge \ g_{e_n}(v'). \end{aligned} \tag{FFIS}$$

The flow approach and its supporting plug-in for the Rodin platform, called Usecase/Flow plug-in [27], allows us to derive these proof obligations automatically.

The argument pattern shown in Figure 11 pertains to the required events order (**C2.3.2**) which is proved to be preserved by the respective events of a model *M*. As explained above, each event $Event_{i'}$ can be either enabled (**ena**), or disabled (**dis**), or possibly enabled (**fis**) by some other event $Event_i$. This suggests that the mapping function F_M is of the form:

$$\begin{aligned} \mathit{Requirement} \mapsto \{ & (Event_i, \textbf{ ena}, Event_{i'}), \\ & (Event_j, \textbf{ dis}, Event_{j'}), \\ & (Event_k, \textbf{ fis}, Event_{k'}), \ ... \}. \end{aligned}$$

The corresponding theorem is constructed according to the definition of either (FENA), or (FDIS), or (FFIS). Then, the discharged proof obligations for each such a pair of events are provided as the evidence in a safety case, e.g., **Sn2.3.1** in Figure 11.

The instantiation of the proposed argument pattern can be achieved by preserving a number of the following steps:

- a requirement *Requirement* should be replaced with a particular safety requirement;
- a formal model *M* should be referred to in a GSN model element;

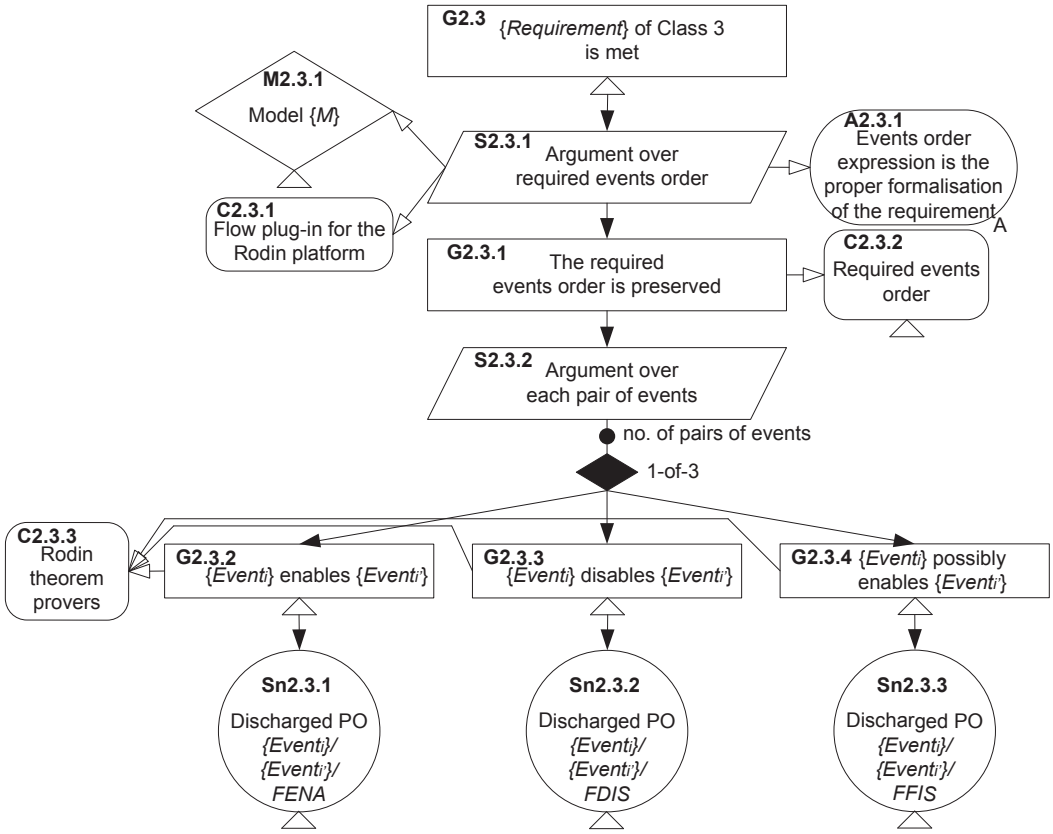

Figure 11: Argument pattern for safety requirements of Class 3

- the concrete mapping between the requirement and the corresponding pairs of events and relationships between them should be provided, while the required events order based on this mapping should be referred to in a GSN context element;
- a separate goal for each pair should be introduced in the safety case;
- each goal that claims the enabling relationship between events should be supported by the proof obligation of the type FENA in a GSN solution element;
- each goal that claims the disabling relationship between events should be supported by the proof obligation of the type FDIS in a GSN solution element;
- each goal that claims the possibly enabling relationship between events should be supported by the proof obligation of the type FFIS in a GSN solution element.

In the already introduced case study AOCS (Section 4.3), there is a set of requirements regulating the order of actions to take place in the system control flow. These requirements define the desired rules of transitions between modes, e.g.,

> **SR-cl3:** *The system shall perform its (normal or failure handling) operation only when there are no currently running transitions between modes at any level.*

This means that once a transition is initiated either by the high-level mode manager or lower level managers, it has to be completed before system operation continues.

As an example, we consider a formalisation of the requirement **SR-cl3** at the most abstract level, i.e., the MACHINE **MM_Abs_M** and the CONTEXT **MM_Abs_C**, where the essential behaviour of the high-level mode manager is introduced.

The required events order (**C2.3.2**) is depicted by the usecase/flow diagram in Figure 12. This flow diagram can be seen as a use case scenario specification attached to the MACHINE **MM_Abs_M**. The presented flow diagram is drawn in the graphical editor for the Usecase/Flow plug-in for the Rodin platform. While defining the desired relationships between events using this editor, the corresponding proof obligations are generated automatically by the Rodin platform.

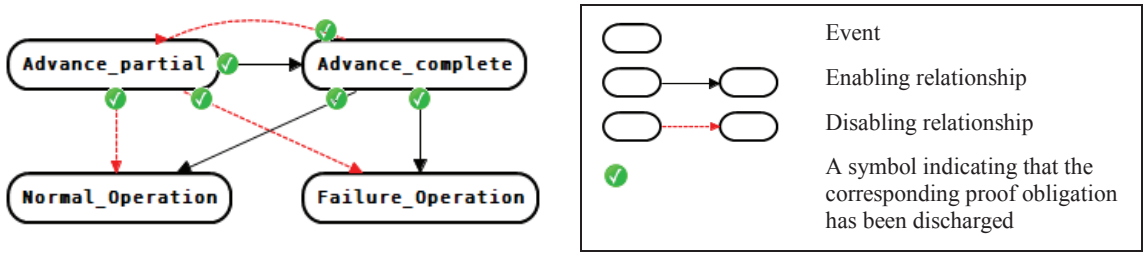

Figure 12: The partial flow diagram of the abstract machine of AOCS

In terms of the usecase/flow approach, the requirement **SR-cl3** states that the event *Advance_partial* enables the event *Advance_complete* and disables operation events *Normal_Operation* and *Failure_Operation*. In its turn, the event *Advance_complete* disables the event *Advance_partial* and enables system (normal or failure handling) operation events. Then, the mapping function F_M is instantiated as follows:

$$\begin{aligned}\textbf{SR-cl3} \mapsto \{&(\textit{Advance\_partial}, \textbf{ena}, \textit{Advance\_complete}),\\ &(\textit{Advance\_partial}, \textbf{dis}, \textit{Normal\_Operation}),\\ &(\textit{Advance\_partial}, \textbf{dis}, \textit{Failure\_Operation}),\\ &(\textit{Advance\_complete}, \textbf{dis}, \textit{Advance\_partial}),\\ &(\textit{Advance\_complete}, \textbf{ena}, \textit{Normal\_Operation}),\\ &(\textit{Advance\_complete}, \textbf{ena}, \textit{Failure\_Operation})\}.\end{aligned}$$

The instance of the argument pattern for the safety requirement **SR-cl3** is shown in Figure 13.

4.5 Argument pattern for SRs about the absence of system deadlock (Class 4)

In this section, we propose an argument pattern for the safety requirements stipulating the absence of the unexpected stop of the system (Figure 14). We formalise requirements of *Class 4* within an Event-B model *M* as the deadlock freedom theorem. Similarly to the SRs of *Class 2*, this theorem has to be proved only once for the whole model *M*. The theorem is reflected in the argument strategy that is used to develop the main goal of the pattern (**S2.4.1** in Figure 14).

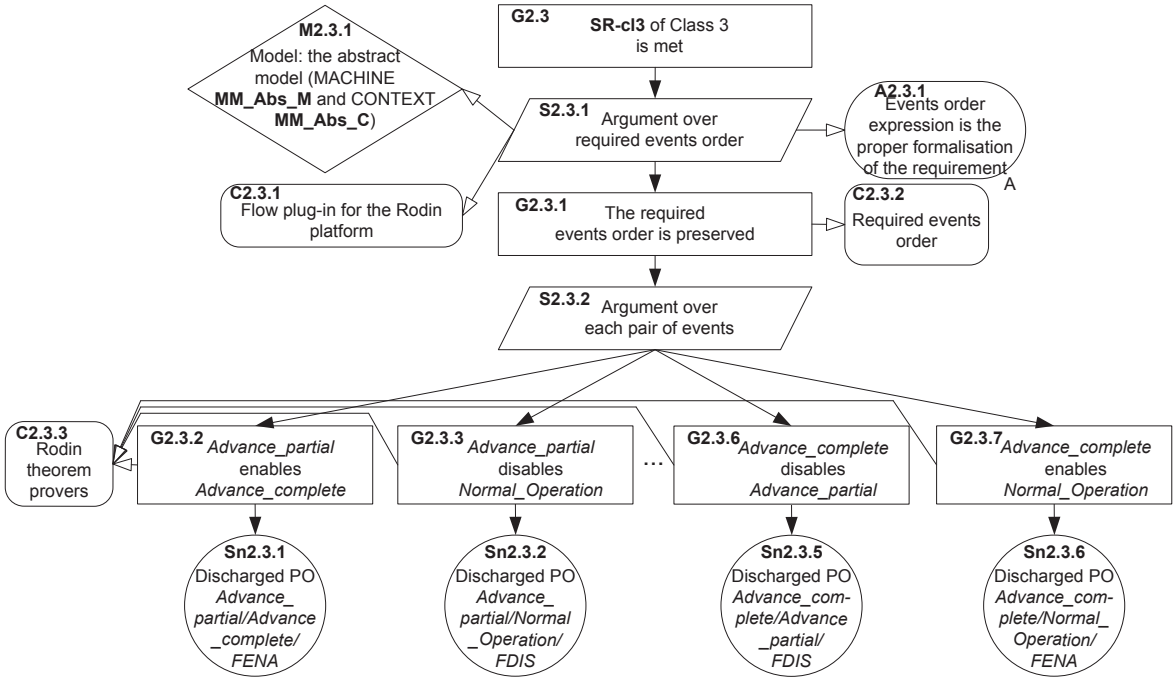

Figure 13: The pattern instantiation example

Formally, the deadlock freedom theorem is formulated as the disjunction of guards of all model events $g_1(d, c, v) \vee ... \vee g_K(d, c, v)$, where K is the total number of model events:

$$\textit{thm\_dlf}: \quad A(d, c),\ I(d, c, v) \ \vdash\ g_1(d, c, v) \vee ... \vee g_K(d, c, v).$$

The corresponding mapping function F_M for this argument pattern is defined as $\textit{Requirement} \mapsto \{\textit{event}_1, \ ... \ , \textit{event}_K\}$. Then, the instance of the (THM_M) proof obligation given in Section 2.1 provides the evidence for the safety case (**Sn2.4.1** in Figure 14).

The argument pattern presented in Figure 14 can be instantiated as follows:

- a requirement *Requirement* should be replaced with a particular safety requirement;
- a formal model *M* should be referred to in a GSN model element;
- the concrete mapping between the requirement and the corresponding model events should be supplied, while the theorem *thm_dlf* formalising the requirement from this mapping should be referred to in a GSN context element;
- the proof obligation of the type THM_M discharged by the Rodin platform should be included in the safety case as the evidence supporting that the top-level claim (i.e., **G2.4** in Figure 14) holds.

We illustrate the instantiation of this argument pattern by a simple example presented by Abrial in Chapter 2 of [4]. The considered system performs controlling cars on a bridge. The bridge connects the mainland with an island. Cars can always either enter the compound or leave it. Therefore, the absence of the system deadlock should be guaranteed, i.e.,

SR-cl4: *Once started, the system should work for ever.*

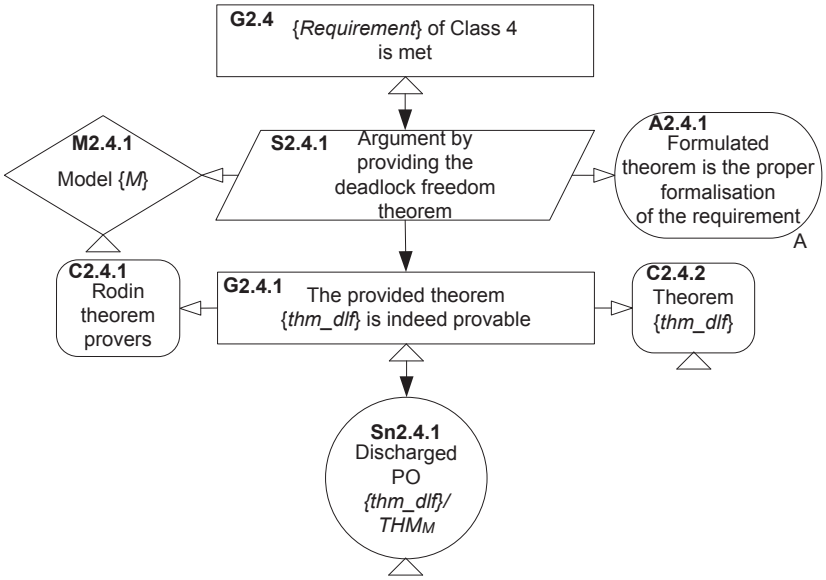

Figure 14: Argument pattern for safety requirements of Class 4

The semantics of Event-B allows us to chose the most abstract specification to argue over the deadlock freedom of a system. According to the notion of the *relative deadlock freedom*, which is a part of the Event-B semantics, new deadlocks cannot be introduced in a refinement step [3]. As a consequence, once the model is proved to be deadlock free, no new refinement step can introduce a deadlock.

The abstract model of the system has three events: *Initialisation*, *ML_out* and *ML_in*. Thus, the concrete mapping function F_M is as follows:

$$\textbf{SR-cl4} \mapsto \{\mathit{Initialisation}, \mathit{ML\_out}, \mathit{ML\_in}\}.$$

Here *ML_out* models leaving the mainland, while *ML_in* models entering the mainland. The former event has the guard $n < d$, where n is a number of cars on the bridge and d is a maximum number of cars that can enter the bridge. The latter event is guarded by the condition $n > 0$, which allows this event to be enabled only when some car is on the island or the bridge. Therefore, the corresponding deadlock freedom theorem **thm_cl4_ex** can be defined as follows:

$$\textbf{thm\_cl4\_ex:} \quad n > 0 \vee n < d.$$

The event *Initialisation* does not have a guard and therefore is not reflected in the theorem. The instantiated fragment of the safety case for this example is shown in Figure 15.

The details on the considered formal development in Event-B (Controlling cars on a bridge) as well as the derived proof obligation of the deadlock freedom can be found in [3,4].

[3] This may be enforced by the corresponding generated theorem to be proved for the respective model.

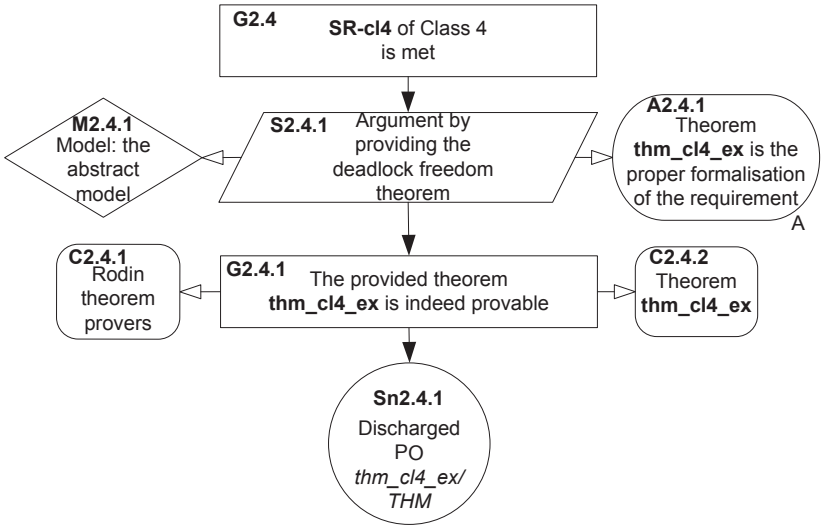

Figure 15: The pattern instantiation example

4.6 Argument pattern for SRs about system termination (Class 5)

In contrast to *Class 4*, *Class 5* contains the safety requirements stipulating the system termination in particular cases. For instance, it corresponds to failsafe systems (i.e., systems which need to be put into a safe but non-operational state to prevent an occurrence of a hazard). Despite the fact that the argument pattern is quite similar to the one about the absence of system deadlock, this class of safety requirements can be considered as essentially opposite to the previous one. Here the requirements define the conditions when the system must terminate. More specifically, the system is required to have a deadlock either (1) in a specific state of the model *M*, i.e., after the execution of some event e_i (where $i \in 1 \,..\, K$ and *K* is the total number of model events), or (2) once a shutdown condition (*shutdown_cond*) is satisfied:

$$(1) \quad \mathit{after}(e_i) \cap \mathit{before}(E) = \varnothing,$$
$$(2) \quad \mathit{shutdown\_cond} \cap \mathit{before}(E) = \varnothing,$$

where *shutdown_cond* is a predicate formalising a condition when the system terminates and *before(E)* is defined as a union of pre-states of all the model events:

$$\mathit{before}(E) = \bigcup_{e \in E} \mathit{before}(e).$$

Correspondingly, the mapping function F_M for *Class 5* can be either of the form

$$(1) \quad \mathit{Requirement} \mapsto \{e_i, e_1, ..., e_K\}, \text{ or}$$
$$(2) \quad \mathit{Requirement} \mapsto \{\mathit{state\ predicate}, e_1, ..., e_K\},$$

where *state predicate* is a formally defined shutdown condition.

Then, for the first case, the theorem about a shutdown condition has the following form:

$$\mathit{thm\_shd}: \quad A(d,c),\ I(d,c,v) \ \vdash\ \mathit{after}(e_i) \Rightarrow \neg \mathit{before}(E),$$

while, for the second case, it is defined as:

$$\textit{thm\_shd}: \quad A(d,c),\ I(d,c,v) \vdash \textit{shutdown\_cond} \Rightarrow \neg \textit{before}(E).$$

The argument pattern presented in Figure 16 can be instantiated as follows:

- a requirement *Requirement* should be replaced with a particular safety requirement;
- a formal model *M* should be referred to in a GSN model element;
- the concrete mapping between the requirement and the corresponding model events (and state predicates) should be provided, while the theorem *thm_shd* formalising the requirement from this mapping should be referred to in a GSN context element;
- the proof obligation of the type THM_M discharged by the Rodin platform should be included in the safety case as the evidence supporting that the top-level claim (i.e., **G2.5**) holds.

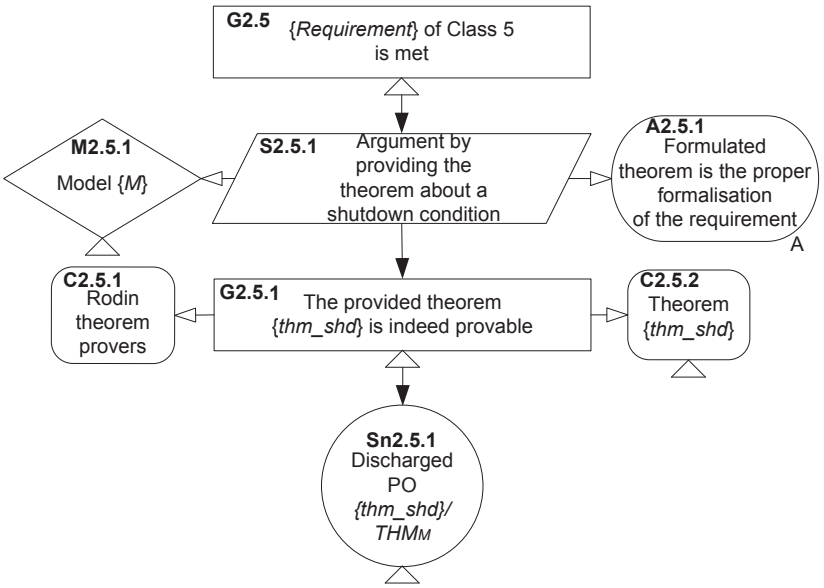

Figure 16: Argument pattern for safety requirements of Class 5

To show an example of the pattern instantiation, let us consider the sluice gate control system [46] described in detail in Section 4.2. This system is a failsafe system. To handle critical failures, it is required to raise an alarm and terminate:

> **SR-cl5:** *When a critical failure is detected, an alarm shall be raised and the system shall be stopped.*

Thus, we need to assure that our model also terminates after the execution of the event which sets the alarm on (i.e., the event *SafeStop* in the model). This suggests the concrete instance of the mapping function F_M to be of the form:

$$\textbf{SR-cl5} \mapsto \{\textit{SafeStop}, \textit{Environment}, \textit{Detection\_door1}, ..., \textit{close2}, \textit{closed2}\}.$$

Then, the corresponding theorem **thm_cl5_ex**, which formalises the safety requirement **SR-cl5**, can be formulated as follows:

$$
\begin{aligned}
\textbf{thm\_cl5\_ex:}\ & \forall flag', \mathit{Stop}' \cdot \\
& (\exists\ flag,\ \mathit{door1\_fail},\ \mathit{door2\_fail},\ pressure\_fail,\ \mathit{Stop} \cdot \\
& flag = \mathit{CONT} \wedge (\mathit{door1\_fail} = \mathit{TRUE} \vee \\
& \mathit{door2\_fail} = \mathit{TRUE} \ \vee\ \mathit{pressure\_fail} = \mathit{TRUE}) \wedge \\
& \mathit{Stop} = \mathit{FALSE} \ \wedge\ flag' = \mathit{PRED} \wedge \mathit{Stop}' = \mathit{TRUE}) \\
& \Rightarrow \\
& \neg(\mathit{before}(Environment) \vee \mathit{before}(Detection\_door1) \vee ... \vee \\
& \mathit{before}(close2) \vee \mathit{before}(closed2)),
\end{aligned}
$$

where the variable *flag* indicates the current phase of the sluice gate controller, while the variables *door1_fail*, *door2_fail* and *pressure_fail* stand for failures of the system components (the doors and the pressure pump respectively). The variable *Stop* models an alarm and a signal to stop the physical operation of the system components. Finally, *Environment*, *Detection_door1*, ..., *closed2* are model events. The corresponding instance of the argument pattern is given in Figure 17.

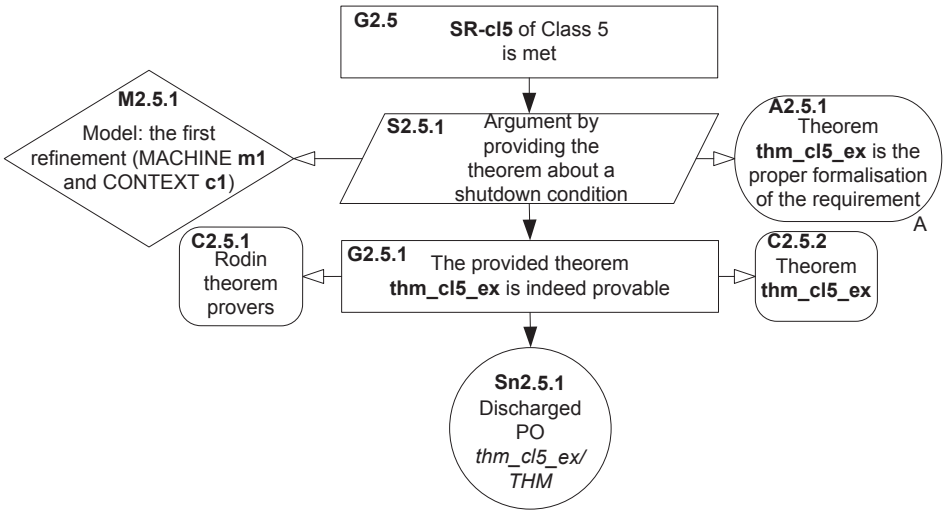

Figure 17: The pattern instantiation example

4.7 Argument pattern for Hierarchical SRs (Class 6)

Sometimes a whole requirements document or some particular requirements (either functional or safety) of a system may be structured in a hierarchical way. For example, a general safety requirement may stipulate actions to be taken in the case of a system failure, while more specific safety requirements elaborate on the general requirement by defining how the failures of different system components may contribute to such a failure of the system as well as regulate the actions to mitigate these failures. Often, the numbering of requirements may

indicate such intended hierarchical relationships. A more general requirement can be numbered *REQ X*, while its more specific versions – *REQ X.1*, *REQ X.2*, etc. In our classification, we call such requirements *Hierarchical SRs*.

The class of *Hierarchical SRs* (*Class 6*) differs from the previously described classes since it involves several, possibly quite different yet hierarchically linked requirements. To create the corresponding argument patterns for such cases, we apply a *composite* approach. This means that the involved individual requirements (a more general requirement and its more detailed counterparts) can be shown to hold separately in different models of the system development in Event-B, by instantiating suitable argument patterns from the described classes 1-5. Moreover, to ensure the consistency of their hierarchical link, an additional fragment in a safety case is needed. This fragment illustrates that the formalisation of the involved requirements is consistent, even if it is done in separate models of the Event-B formal development. To address the class of hierarchical requirements, in this section we propose an argument pattern that facilitates the task of construction of such an additional fragment of a safety case.

Since the main property of the employed refinement approach is the preservation of consistency between the models, it is sufficient for us to show that the involved models are valid refinements of one another. In Event-B, to guarantee consistency of model transformations, we need to show that the concrete events refine their abstract versions by discharging the corresponding proof obligations to verify guard strengthening (GRD) and action simulation (SIM), as given in Section 2.1. This procedure may involve the whole set of the refined events. However, to simplify the construction of the corresponding fragment of a safety case, we limit the number of events by choosing only those events that are affected by the requirements under consideration. To achieve this, we rely on the given mappings for higher-level and lower-level requirements, returning the sets of the involved model expressions $Req_h \Rightarrow \{Expr_1, ..., Expr_N\}$ and $Req_l \Rightarrow \{Expr_1, ..., Expr_P\}$. Making a step further, we can always obtain the set of affected model events:

$$Req_h \Rightarrow \{Expr_1, ..., Expr_N\} \Rightarrow \{Event_{h_1}, ..., Event_{h_K}\},$$
$$Req_l \Rightarrow \{Expr_1, ..., Expr_P\} \Rightarrow \{Event'_{l_1}, ..., Event'_{l_L}\}.$$

As a result, we attach proofs only for those events from $\{\text{Event}'_{l_1}, ..., \text{Event}'_{l_L}\}$ that refine some events from $\{\text{Event}_{h_1}, ..., \text{Event}_{h_K}\}$.

Each higher-level requirement may be linked with a set of more detailed requirements in the requirements document. Nevertheless, to simplify the task, let us consider the case where there is only one such a lower-level requirement. If there are more than one such a requirement, one could reiterate the proposed approach by building a separate fragment of a safety case for each pair of linked requirements.

In Figure 18, *Higher-level req.* stands for some higher-level requirement, while *Lower-level req.* is a requirement that is a more detailed version of the higher-level one. The higher-level requirement is mapped onto a formal model M_{abs} and the lower-level requirement is mapped onto a formal model M_{concr} (where M_{concr} is a refinement of M_{abs}) using one of the mapping functions defined for the classes 1-5.

Following the procedure described above, we can associate *Higher-level req.* with the set

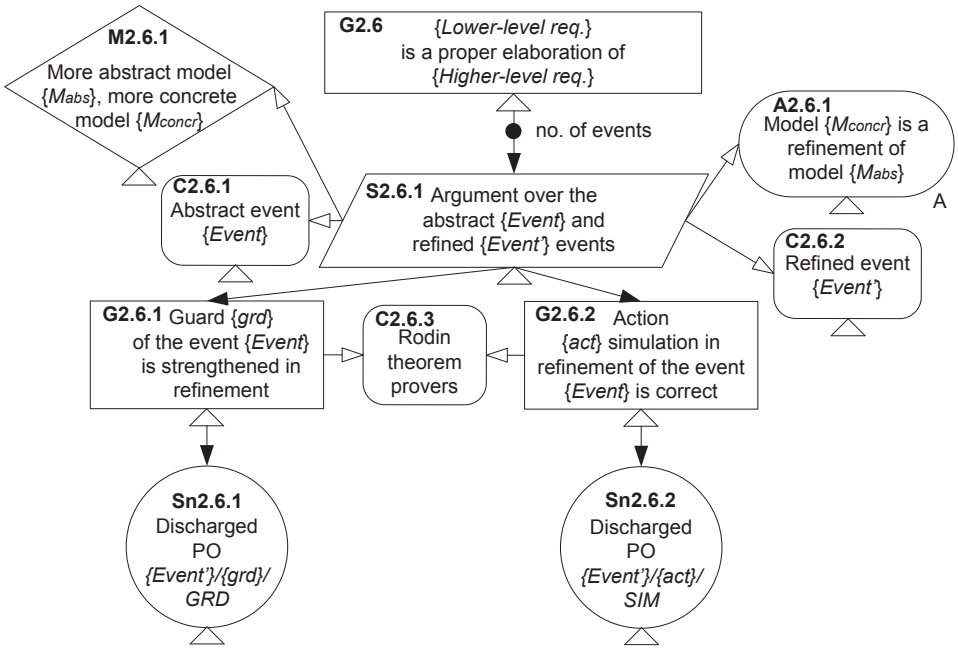

Figure 18: Argument pattern for safety requirements of Class 6

of affected events $\{Event_{h_1}, ..., Event_{h_K}\}$. Similarly, *Lower-level req.* is associated with its own set of affected events $\{Event'_{l_1}, ..., Event'_{l_L}\}$.

For each pair of events $Event$ and $Event'$ from the obtained sets, the following two generated proof obligations (GRD) and (SIM) are needed to be proved to establish correctness of model refinement (Section 2.1):

$$H(d,c,v,w),\ g'_{Event'}(d,c,w) \vdash g_{Event}(d,c,v),$$
$$H(d,c,v,w),\ g'_{Event'}(d,c,w),\ BA'_{Event'}(d,c,w,w') \vdash \exists v'.BA_{Event}(d,c,v,v') \wedge I'(d,c,v',w').$$

The established proofs of the types GRD and SIM serve as solutions in our pattern, **Sn2.6.1** and **Sn2.6.2** in Figure 18 respectively.

The instantiation of the pattern proceeds as shown below:

- requirements *Higher-level req.* and *Lower-level req.* should be replaced with specific requirements;
- a more abstract formal model M_{abs} and a more concrete formal model M_{concr} should be referred to in a GSN model element;
- the pairs of the associated events of the respective abstract and concrete system models should be referred to in GSN context elements;
- the proof obligations of the types GRD and SIM discharged by the Rodin platform should be included in the safety case as solutions.

Moreover, there can be several hierarchical levels of requirements specification. To cope with this case, we propose to instantiate patterns for each such a level separately.

To illustrate the construction of a safety case fragment for this class of requirements, we refer to the sluice gate control system [46] described in Sections 4.2 and 4.6. Some safety

requirements of this system are hierarchically structured. Thus, there is a more generic safety requirement **SR-cl6-higher-level**:

SR-cl6-higher-level: *The system shall be able to handle a critical failure by either initiating a shutdown or a recovery procedure*

stipulating that some actions should take place in order to tolerate any critical failure. However, it does not define the precise procedures associated with this failure handling. In contrast, there is a more detailed counterpart **SR-cl6-lower-level** of the requirement **SR-cl6-higher-level** (it was presented in the previous section as the requirement **SR-cl5**). It regulates precisely that an alarm should be raised and the system should stop its operation (the system should terminate):

SR-cl6-lower-level: *When a critical failure is detected, an alarm shall be raised and the system shall be stopped.*

These safety requirements are shown to hold in different models of the system development. The requirement **SR-cl6-higher-level** is formalised as two invariants at the most abstract level of the formal specification in Event-B, the MACHINE **m0**, while the requirement **SR-cl6-lower-level** is formalised as a theorem in the MACHINE **m1**. Note that the MACHINE **m1** is the refinement of the MACHINE **m0**.

The instance of the mapping function F_M for the requirement **SR-cl6-higher-level** is as follows:

$$\textbf{SR-cl6-higher-level} \mapsto \{\textbf{inv\_1\_cl6}, \textbf{inv\_2\_cl6}\},$$

where:

inv_1_cl6: $Failure = FALSE \Rightarrow Stop = FALSE$,
inv_2_cl6: $Failure = TRUE \wedge flag \neq CONT \Rightarrow Stop = TRUE$.

The handling of critical failures is non-deterministically modelled in the event *ErrorHandling* of the abstract model (Figure 19). The local variable *res* is of the type *BOOL* and can be either *TRUE* or *FALSE*. It means that, if a successful error handling procedure that does not lead to the system termination has been performed, both variables standing for a critical failure (*Failure*) and for the system shutdown (*Stop*) are assigned the values *FALSE* and the system continues its operation. Otherwise, they are assigned the values *TRUE* leading to the system termination.

The fragment of a safety case for the safety requirement **SR-cl6-higher-level** can be constructed preserving the instructions determined in Section 4.2, while the fragment of a safety case for the requirement **SR-cl6-lower-level** can be found in Section 4.6.

Now let us focus on ensuring the hierarchical link between these requirements by instantiating the argument pattern for *Class 6*. Following the proposed approach, we define a set of the affected model events for the higher-level safety requirement: {*Environment*, *Detection*, *ErrorHandling*, *Prediction*, *NormalOperation*}, and for the lower-level safety requirement: {*Environment*, *Detection_NoFault*, *Detection_Fault*, *SafeStop*, *Prediction*, *NormalSkip*}. For

*// Event in the MACHINE **m0***	*// Event in the refined MACHINE **m1***
event ErrorHandling	**event** SafeStop
any *res*	**refines** ErrorHandling
where	**where**
@grd1 flag = CONT	@grd1 flag = CONT
@grd2 Failure = TRUE	@grd2 door1_fail = TRUE ∨
@grd3 Stop = FALSE	door2_fail = TRUE ∨
@grd4 *res* ∈ BOOL	pressure_fail = TRUE
then	@grd3 Stop = FALSE
@act1 flag ≔ PRED	**with**
@act2 Stop ≔ *res*	@res res = TRUE
@act3 Failure ≔ *res*	**then**
end	@act1 flag ≔ PRED
	@act2 Stop ≔ TRUE
	end

Figure 19: Events *ErrorHandling* and *SafeStop*

simplicity, here we consider only one pair of events *ErrorHandling* and *SafeStop* shown in Figure 19.

In the Event-B development of the sluice gate system, the non-determinism modelled by the local variable *res* is eliminated via introduction of a specific situation leading to the system shutdown. All other fault tolerance procedures are left out of the scope of the presented development.

Additionally to the introduction of the deterministic procedures for error handling, the variable *Failure* is data refined in the first refinement **m1**. Now, the system failure may occur either if the component *door1* fails ($door1\_fail = TRUE$), or *door2* fails ($door2\_fail = TRUE$), or the pressure pump fails ($pressure\_fail = TRUE$). This relationship between the old abstract variable and new concrete ones is defined by the corresponding gluing invariant.

The corresponding instance of the argument pattern is presented in Figure 20. To ensure that the requirement **SR-cl6-lower-level** is a proper elaboration of the requirement **SR-cl6-higher-level** (the goal **G2.6** in Figure 20), we argue over the abstract event *ErrorHandling* and the refined event *SafeStop*. We show that the guard **grd2** is strengthened in the refinement (the discharged proof obligation (GRD)) and the action **act2** is not contradictory to the abstract version (SIM). The corresponding proof obligations are shown in Figure 21.

4.8 Argument pattern for SRs about temporal properties (Class 7)

So far, we have considered the argument patterns of safety requirements classes where the evidence that the top goal of the pattern holds is constructed based on the proof obligations generated by the Rodin platform. Not all types of safety requirements can be formally demonstrated in this way, however. In particular, the Event-B framework lacks direct support of temporal system properties such as reachability, liveness, existence, etc. Nevertheless, the Rodin platform has an accompanying plug-in, called ProB [47], which allows for model checking of temporal properties.

Therefore, in this section, we propose an argument pattern for the class of safety require-

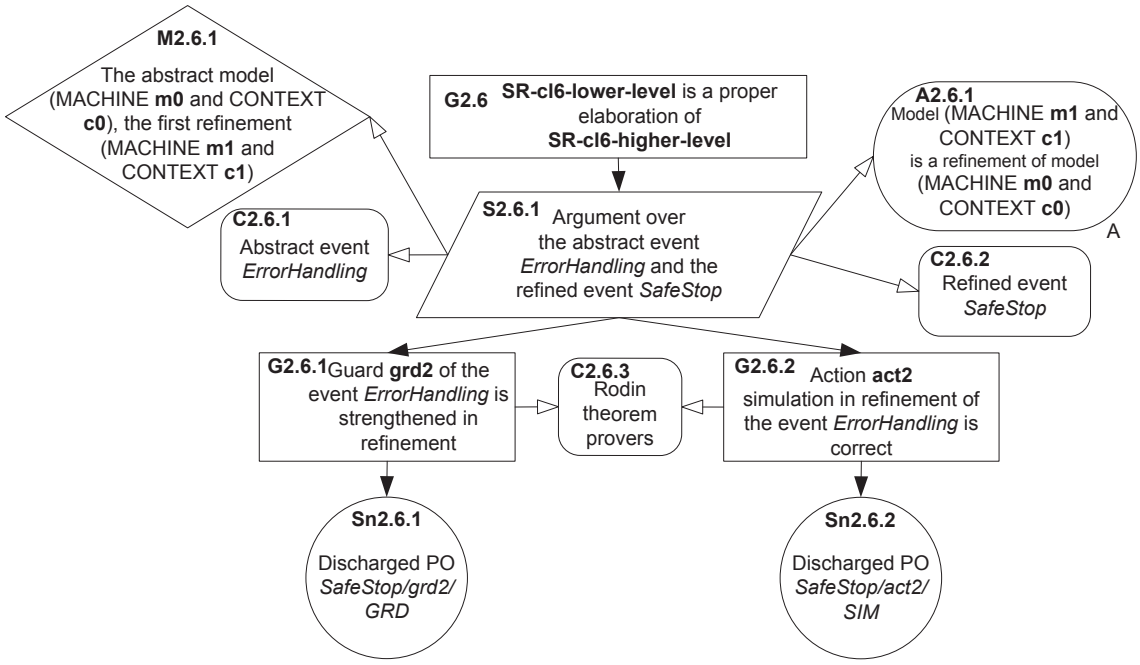

Figure 20: The pattern instantiation example

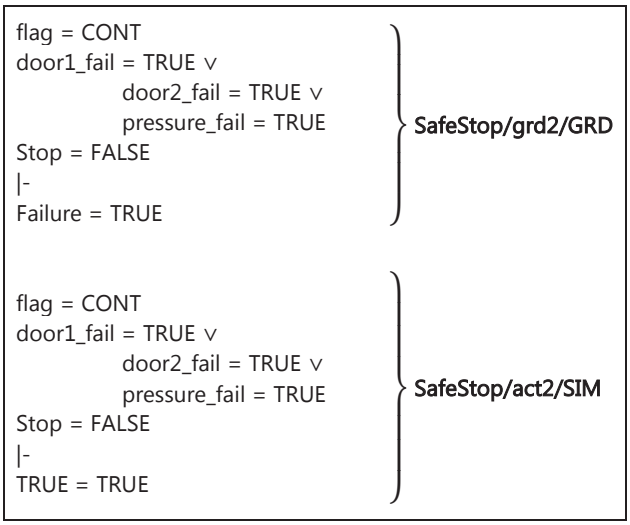

Figure 21: The proof obligations of the types GRD and SIM

ments that can be expressed as temporal properties. The pattern is graphically shown in Figure 22. Here $property_i$ stands for some temporal property to be verified, for $i \in 1 .. N$, where N is the number of temporal properties of the system.

The property to be verified should be formulated as an LTL formula in the *LTL Model Checking* wizard of the ProB plug-in for some particular model M. This suggests the mapping function F_M for *Class 7* to be of the form

$$Requirement \mapsto \{LTL\,formula\}.$$

Each such a temporal property should be well-defined according to restrictions imposed on LTL in ProB. The tool can generate three possible results: (1) the given LTL formula is true for all valid paths (no counter-example has been found, all nodes have been visited); (2) there is a path that does not satisfy the formula (a counter-example has been found and it is shown in a separate view); (3) no counter-example has been found, but the temporal property in question cannot be guaranteed because the state space was not fully explored.

To instantiate this pattern, one needs to proceed as follows:

- a requirement *Requirement* should be replaced with a particular safety requirement;
- a formal model M should be referred to in a GSN model element;
- the concrete mapping between the requirement and the corresponding LTL formula should be supplied, while each temporal property $property_i$ from this mapping should be referred to in a GSN context element;
- model checking results on an instantiated property that have been generated by ProB should be included as the evidence that this property is satisfied.

There are several alternative ways to reason over temporal properties in Event-B [5, 29, 34, 48]. The most recent of them is that of Hoang and Abrial [34]. The authors propose a set of proof rules for reasoning about such temporal properties as liveness properties (existence, progress and persistence). The main drawback of this approach is that, even though it does not require extensions of the proving support of the Rodin platform, it necessitates extension of the Event-B language by special clauses (annotations) corresponding to different types of temporal properties. Alternatively, in the cases when a temporal property can be expressed as a condition on the system control flow, the usecase/flow approach [37] described in Section 4.4 can be used.

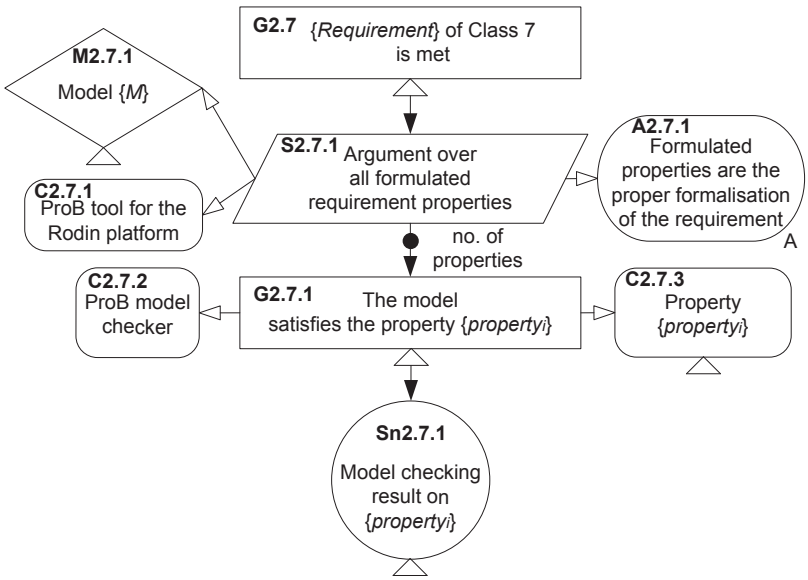

Figure 22: Argument pattern for safety requirements of Class 7

To exemplify the instantiation of the argument pattern for safety requirements of *Class 7*, we consider a distributed monitoring system – Temperature Monitoring System (TMS). The full system formal specification in Event-B is presented in [56]. In brief, the TMS consists of three data processing units (DPUs) connected to operator displays in the control room. At each cycle, the system performs readings of the temperature sensors, distributes preprocessed data among DPUs where they are analysed (processed), and finally displays the output to the operator. The system model is developed in such a way that it allows for ensuring integrity of the temperature data as well as its freshness.

A safety requirement about a temporal property of this system, which we consider here, is as follows:

SR-cl7: *Each cycle the system shall display fresh and correct data.*

We leave out of the scope of this paper the mechanism of ensuring data freshness, correctness and integrity, while focusing on the fact of displaying data at each cycle. In the given Event-B specification, a new cycle starts when the event *Environment* is executed. To verify that the system will eventually display the data to the operator (i.e., the corresponding event *Displaying* will be enabled), we formulate an LTL formula for the abstract model of the system (**temp_pr_ex**). Then, the instance of the mapping function F_M is defined as

$$\textbf{SR-cl7} \mapsto \{\textbf{temp\_pr\_ex}\},$$

where

$$\textbf{temp\_pr\_ex:}\ \Box\,(\mathit{after}(\mathit{Environment})\ \rightarrow\ \Diamond\ \mathit{before}(\mathit{Displaying})).$$

Here $\Box$ is an operator *"always"* and $\Diamond$ stands for *"eventually"*.

The formula **temp_pr_ex** has the following representation in ProB:

$$\begin{array}{l} G\,(\{\forall\, main\_phase',\ temp\_sensor',\ curr\_time' \cdot \\ \quad main\_phase' \in \mathit{MAIN\_PHASES}\ \wedge\ temp\_sensor' \in \mathbb{N}\ \wedge\ curr\_time' \in \mathbb{N}\ \wedge \\ \quad (\exists\, main\_phase,\ sync\_t\ \cdot\ main\_phase \in \mathit{MAIN\_PHASES}\ \wedge\ sync\_t \in \mathbb{N}\ \wedge \\ \quad main\_phase' = PROC\ \wedge\ temp\_sensor' \in \mathbb{N}\ \wedge\ curr\_time' = sync\_t)\} \\ \quad \Rightarrow \\ \quad F\ \{\exists\, ss, \mathit{TEMP\_SET} \cdot main\_phase = \mathit{DISP} \wedge packet\_sent\_flag = \mathit{TRUE}\ \wedge \\ \quad \mathit{TEMP\_SET}\ \subseteq\ \mathbb{N}\ \wedge\ time\_progressed = \mathit{TRUE}\ \wedge \\ \quad ss = \{x\ \mapsto y \mid \exists\, i\ \cdot\ i \in dom(timestamp)\ \wedge\ x = timestamp(i)\ \wedge \\ \quad y = temperature(i)\}[curr\_time - Fresh\_Delta\ \cdot\cdot\ curr\_time]\ \wedge \\ \quad (ss \neq \varnothing \Rightarrow \mathit{TEMP\_SET} = ss) \wedge (ss = \varnothing \Rightarrow \mathit{TEMP\_SET} = \{\mathit{ERR\_VAL}\})\}), \end{array}$$

where G stands for the temporal operator *"globally"* and F is the temporal operator *"finally"*. These operators correspond to the standard LTL constructs *"always"* and *"eventually"* respectively. For the detailed explanation of the used variables, constants and language constructs, see [56].

In this case, the result of the model checking of this property in ProB is *"no counter-example has been found, all nodes have been visited"*. Figure 23 illustrates the corresponding instance of the argument pattern.

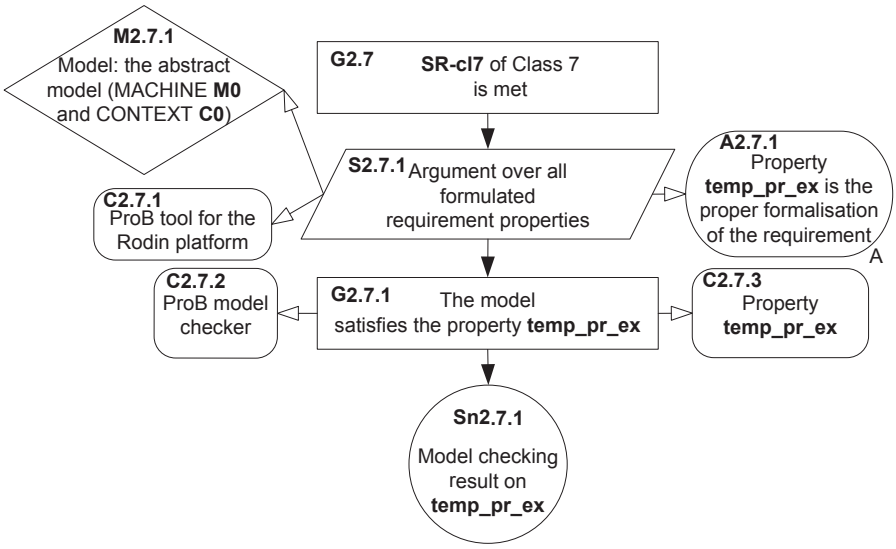

Figure 23: The pattern instantiation example

4.9 Argument pattern for SRs about timing properties (Class 8)

Another class of safety requirements that requires to be treated in a different way is *Class 8* containing timing properties of the considered system. As we have already mentioned, the representation of timing properties in Event-B has not been explicitly defined yet. Nonetheless, the majority of safety-critical systems rely on timing constraints for critical functions. Obviously, the preservation of such requirements must be verified. To address this, we propose to bridge Event-B modelling with model checking of timing properties in Uppaal.

Figure 24 shows our argument pattern for the safety requirements about timing properties. In our pattern, $property_j$ stands for some timing property to be verified, for $j \in 1 \,..\, N$, where N is the number of timing properties.

Following the approach proposed by Iliasov et al. [39], we rely on the Uppaal toolset for obtaining model checking results that further can be used as the evidence in a safety case. The timing property in question can be formulated using the TCTL language. A timed automata model (an input model of Uppaal) is obtained from a process-based view extracted from an Event-B model as well as additionally introduced clocks and timing constraints. The generic mapping function F_M for this class is then of the form $Requirement \mapsto \{TCTL\,formula\}$.

Uppaal uses a subset of TCTL to specify properties to be checked [11]. The results of the property verification can be of three types: (1) a trace is found, i.e., a property is satisfied (user can then import the trace into the simulator); (2) a property is violated; (3) the verification is inconclusive with the approximation used.

We propose the following steps in order to instantiate this pattern:

- a requirement *Requirement* should be replaced with a particular safety requirement;
- a formal development that consists of a chain of refinements in Event-B and the corresponding Uppaal model should be referred to in GSN model elements;
- the concrete mapping between the requirement and the corresponding TCTL formula

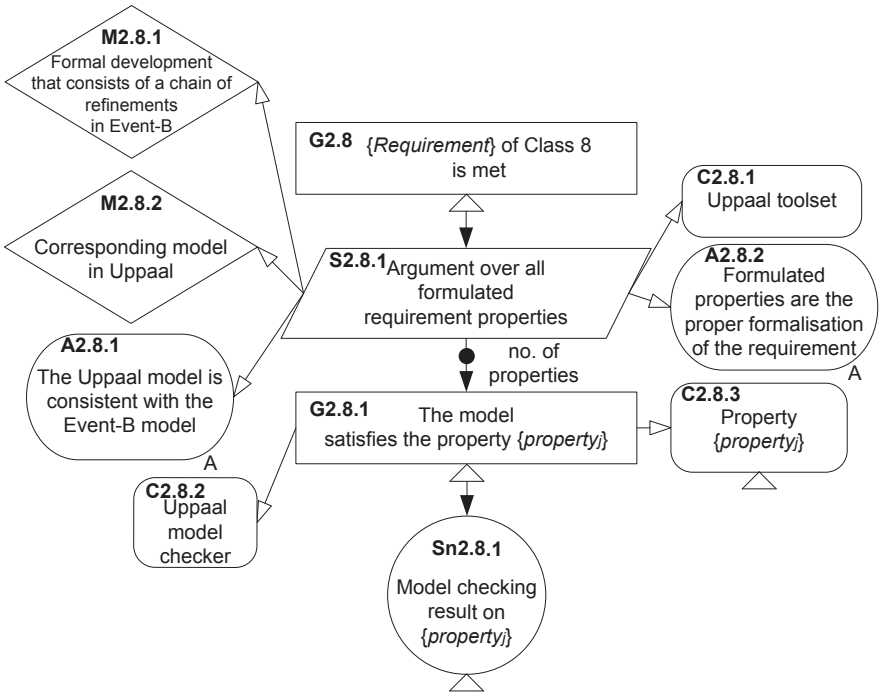

Figure 24: Argument pattern for safety requirements of Class 8

should be provided, while each timing property $property_j$ from this mapping should be referred to in a GSN context element;

- model checking results on an instantiated property that have been generated by Uppaal should be included as the evidence that this property is satisfied.

We adopt a case study considered in [38, 39] in order to show the instantiation of the proposed argument pattern for a safety requirement of *Class 8*. The case study is the Data Processing Unit (DPU) – a module of Mercury Planetary Orbiter of the BepiColombo Mission. The DPU consists of the core software and software of two scientific instruments. The core software communicates with the BepiColombo spacecraft via interfaces, which are used to receive telecommands (TCs) from the spacecraft and transmit science and housekeeping telemetry data (TMs) back to it. In this paper, we omit showing the detailed specification of the DPU in Event-B as well as we do not explain how the corresponding Uppaal model was obtained. We rather illustrate how the verified liveness and time-bounded reachability properties of the system can be reflected in the resulting safety case (Figure 25).

The DPU is required to eventually return a TM for any received TC and must respond within a predefined time bound:

> **SR-cl8:** *The DPU shall eventually return a TM for any received TC and shall respond no later than the maximal response time.*

We consider two timing properties associated with this requirement, i.e.,

time_pr_ex1: $(new\_tc == id) \ \rightarrow \ (last\_tm == id)$,
time_pr_ex2: $A[](last\_tm == id \ \&\& \ \mathit{Obs1}.stop) \ imply \ (\mathit{Obs1}\_c < upper\_bound)$,

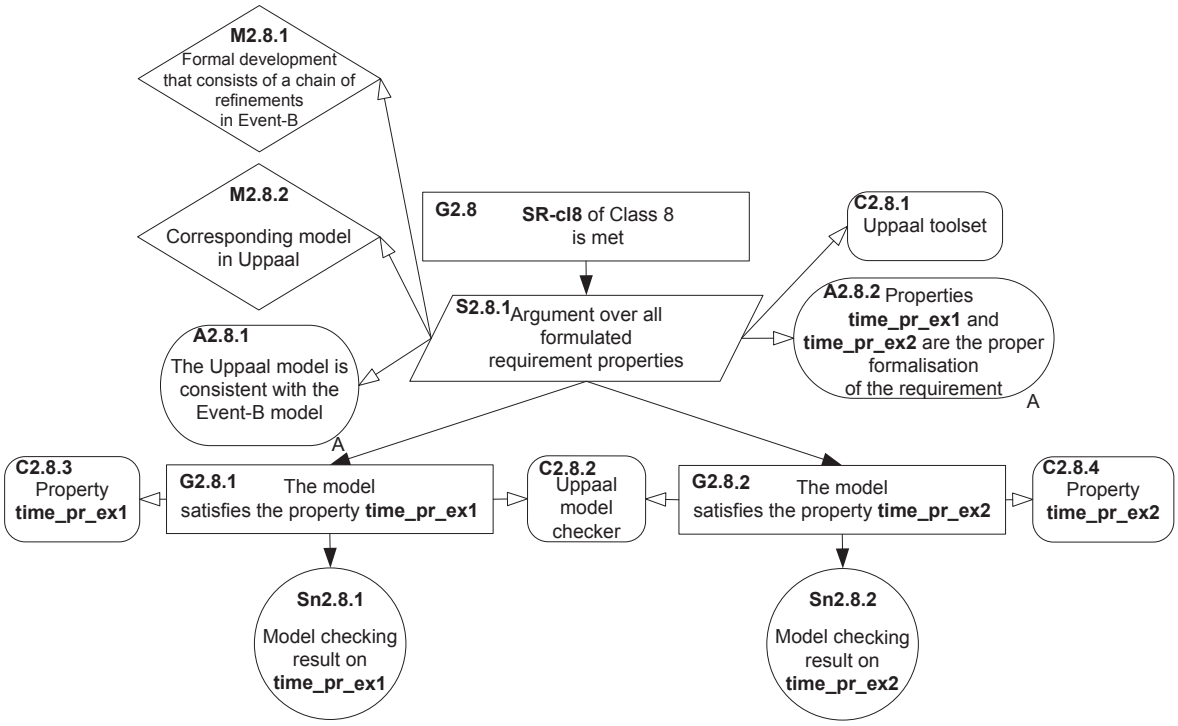

Figure 25: The pattern instantiation example

such that the concrete instance of the generic mapping function F_M is as follows:

$$\textbf{SR-cl8} \mapsto \{\textbf{time\_pr\_ex1}, \textbf{time\_pr\_ex2}\}.$$

The symbol $\rightarrow$ stands for the TCTL *"leads-to"* operator, and id is some TC identification number. $A[]$ stands for *"Always, for any execution path"* and *Obs1* is a special observer process that starts the clock *Obs1_c*, whenever a TC command with id is received, and stops it, once the corresponding TM is returned. The variable *upper_bound* corresponds to the maximal response time. The corresponding instance of the argument pattern is given in Figure 25.

4.10 Summary of the proposed argument patterns

To facilitate the construction of safety cases, we have defined a set of argument patterns graphically represented using GSN. The argumentation and goal decomposition in these patterns were influenced by the formal reasoning in Event-B.

However, since the development utilising formal methods typically require additional reasoning about model correctness and well-definedness, we firstly proposed an argument pattern for assuring well-definedness of the system development in Event-B. Secondly, we proposed a number of argument patterns for assuring safety requirements of a system. We associated these argument patterns with the classification of safety requirements presented

in Section 3.2. Therefore, we distinguished eight classification-based argument patterns. Despite the fact that the proposed classification of safety requirements covers a wide range of different safety requirements, the classification and subsequently the set of argument patterns can be further extended if needed.

Unfortunately, at the meantime not all the introduced classes of safety requirements can be formally demonstrated utilising Event-B solely. Therefore, among the proposed argument patterns there are several patterns where the evidence was constructed using accompanying toolsets – the Usecase/Flow and ProB plug-ins for the Rodin platform, as well as the external model checker for verification of real-time systems Uppaal.

In this section, we exemplified the instantiation of the proposed argument patterns for assuring safety requirements on several case studies. Among them are the sluice gate control system [46], the attitude and orbit control system [53], the system for controlling cars on a bridge [4], the temperature monitoring system [56], and the data processing unit of Mercury planetary orbiter of the BepiColombo mission [38, 39].

The instantiation of the proposed argument patterns is a trivial task. Nonetheless, the application of the overall approach requires basic knowledge of principles of safety case construction as well as a certain level of expertise in formal modelling. Therefore, experience in formal modelling and verification using state-based formalisms would be beneficial for safety and software engineers.

Currently, the proposed approach is restricted by the lack of the tool support. Indeed, manual construction of safety cases especially of large-scale safety-critical systems may be error-prone. We believe that the well-defined steps of evidence construction and the detailed guidelines on the pattern instantiation given in this paper will contribute to the development of the corresponding plug-in for the Rodin platform.

5 Case study – steam boiler

In this section, we demonstrate our proposed methodology (based on argument patterns) for building safety cases on a bigger case study. The considered case study is a steam boiler control system. It is a well-known safety-critical system widely used in industrial applications. Due to the large number of safety requirements of different types imposed on it, this system is highly suitable for demonstration of our methodology.

5.1 System description

The steam boiler (Figure 26) is a safety-critical control system that produces steam and adjusts the quantity of water in the steam boiler chamber to maintain it within the predefined safety boundaries [1]. The situations when the water level is too low or too high might result in loss of personnel life, significant equipment damage (the steam boiler itself or the turbine placed in front of it), or damage to the environment.

The system consists of the following units: a chamber, a pump, a valve, a sensor to measure the quantity of water in the chamber, a sensor to measure the quantity of steam

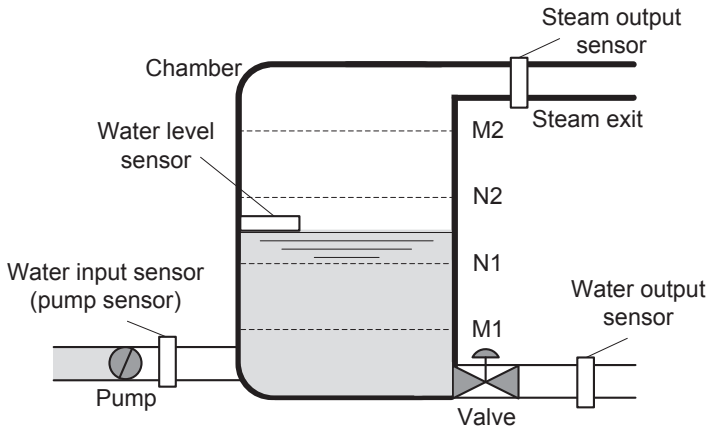

Figure 26: Steam boiler

Table 2: Parameters of the steam boiler

Label	Description	Unit
C	the total capacity of the steam boiler chamber	litre
P	the maximal capacity of the pump	litre/sec
W	the maximal quantity of steam produced	litre/sec
M1	the minimal quantity of water, i.e., the lower safety boundary	litre
M2	the maximal quantity of water, i.e., the upper safety boundary	litre
N1	the minimal normal quantity of water to be maintained during regular operation	litre
N2	the maximal normal quantity of water to be maintained during regular operation	litre

which comes out of the steam boiler chamber, a sensor to measure water input through the pump, and a sensor to measure water output through the valve. The essential system parameters are given in Table 2.

The considered system has several modes. After being powered on, the system enters the **Initialisation** mode. At each control cycle, the system reads sensors and performs failure detection. Then, depending on the detection result, the system may enter either one of its operational modes or the non-operational mode. There are three operational modes in the system: **Normal**, **Degraded**, **Rescue**. In the **Normal mode**, the system attempts to maintain the water level in the chamber between the normal boundaries N1 and N2 (such that N1 $<$ N2) providing that no failures of the system units have occurred. In the **Degraded mode**, the system tries to maintain the water level within the normal boundaries despite failures of some physical non-critical units. In the **Rescue mode**, the system attempts to maintain the normal water level in the presence of a failure of the critical unit – the water level sensor. If failures of the system units and the water level sensor occur simultaneously or the water level is outside of the predefined safety boundaries M1 and M2 (such that M1 $<$ M2), the system enters the non-operational mode **Emergency_Stop**.

In our development, we consider the following failures of the system and its units. The failure of the steam boiler control system is detected if either the water level in the chamber is outside of the safety boundaries (i.e., if it is lower than M1 or higher than M2) or the

combination of a water level sensor failure and a failure of any other system unit (the pump or the steam output sensor) is detected. The water level sensor is considered as failed if it returns a value which is outside of the nominal sensor range or the estimated range predicted in the last cycle. Analogously, a steam output sensor failure is detected. The pump fails if it does not change its state when required.

A water level sensor failure by itself does not lead to a system failure. The steam boiler contains the information redundancy, i.e., the controller is able to estimate the water level in the chamber based on the amount of water produced by the pump and the amount of the released steam. Similarly, the controller is able to maintain the acceptable level of efficiency based on the water level sensor readings if either the pump or the steam output sensor fail. The detailed description of the system, its functional and safety requirements as well as the models of our formal development in Event-B can be found in [55].

5.2 Brief overview of the development

Our Event-B development of the steam boiler case study consists of an abstract specification and its four refinements [55]. The abstract model (MACHINE **M0**) implements a basic control loop. The first refinement (MACHINE **M1**) introduces an abstract representation of the activities performed after the system is powered on and during system operation (under both nominal and failure conditions). The second refinement (MACHINE **M2**) introduces a detailed representation of the conditions leading to a system failure. The third refinement (MACHINE **M3**) models the physical environment of the system as well as elaborates on more advanced failure detection procedures. Finally, the fourth refinement (MACHINE **M4**) introduces a representation of the required execution modes. Each MACHINE has the associated CONTEXT where the necessary data structures are introduced and their properties are postulated as axioms.

Let us now give a short overview of the basic model elements (i.e., constants, variables and events). The parameters of the steam boiler system presented in Table 2 are defined as constants in one of the CONTEXT components. Moreover, several abstract functions are defined there to formalise, for example, the critical water level (*WL_critical*).

The dynamic behaviour of the system is modelled in the corresponding MACHINE components. Some essential variables and events are listed below:

- The variables modelling the steam boiler actuators – the pump and the valve:
  - *pump_ctrl*: the value of this variable equals to *ON* if the pump is switched on, and *OFF* otherwise;
  - *valve_ctrl*: the value of this variable equals to *OPEN* if the valve is open, and *CLOSED* otherwise.
- The variables representing the amount of water passing through the pump and the valve:
  - *pump* stands for the amount of water incoming into the chamber through the pump;

  - *water_output* models the amount of water coming out of the chamber through the valve.

- The variables representing the water level in the chamber:
  - *water_level* models the latest water level sensor readings;
  - *min_water_level* and *max_water_level* represent the estimated interval for the *sensed* water level.
- The variables representing the amount of the steam coming out of the chamber:
  - *steam_output* models the latest steam output sensor readings;
  - *min_steam_output* and *max_steam_output* represent the estimated interval for the *sensed* amount of steam.
- The variables representing failures of the system and its components:
  - *failure* is an abstract boolean variable modelling occurrence of a system failure;
  - *wl_sensor_failure* represents a failure of the water level sensor;
  - *pump_failure* models a failure of the pump actuator;
  - *so_sensor_failure* represents a failure of the steam output sensor.
- The variables modelling phases of the control cycle and the system modes:
  - *phase*: the value of this variable can be equal either to *ENV*, *DET*, *CONT*, *PRED* corresponding to the current controller stage (i.e., reading environment, detecting system failures, performing routing control, or predicting the system state in the next cycle);
  - *preop_flag* is a flag which indicates whether the system is in the pre-operational stage or not;
  - *mode* models the current mode of the system, i.e., *Initialisation*, *Normal*, *Degraded*, *Rescue*, or *Emergency_Stop*.
- The variable *stop* abstractly models system shutdown and raising an alarm.
- Essential events of the modelled system:
  - *Environment*, modelling the behaviour of the environment;
  - *Detection*, representing detection of errors;
  - *PreOperational1* and *PreOperational2*, modelling the initial system procedures to establish the amount of water in the chamber within the safety boundaries;
  - *Operational*, performing controller actions under the nominal conditions;
  - *EmergencyStop*, modelling error handling;
  - *Prediction*, computing the next estimated states of the system.

In the refinement process, such events as *Detection* and *Operational* are split into a number of more concrete events modelling detection of failures of different system components as well as different system operational modes.

5.3 Application of the proposed approach

In this section, we follow our proposed approach to constructing a safety case of a system from its formal model in Event-B. More specifically, firstly we show that our formal development of the steam boiler control system is well-defined by instantiating the corresponding argument pattern (introduced in Section 4.1). Secondly, we apply the classification-based argument patterns (presented in Sections 4.2 – 4.9) to construct the corresponding fragments of the safety case related to specific safety requirements of the considered system.

The steam boiler control system is a complex system, which has a rich functionality and adheres to a large number of safety requirements. The accomplished formal development of this system as well as its safety case are also complex and large in size. Therefore, we will not show the system in its entirety but rather demonstrate application of our methodology on selected system fragments.

5.3.1 Instantiation of the argument pattern for well-definedness of the development

Due to a significant size of the system safety case, here we show only a part of the instantiated pattern for demonstrating well-definedness of a formal development (Section 4.1). Figure 27 presents the resulting fragment of the safety case concerning the first refinement model (MACHINE **M1** and the associated CONTEXT **C1**).

Let us remind that, to apply the well-definedness argument pattern, we have to formally demonstrate axiom consistency in the CONTEXT **C1**. To argue over axiom consistency, we define two groups of axioms. The first group includes axioms defining generic parameters of the system, e.g., the constants associated with the criticality of the water level, which is based on the pre-defined safety boundaries. The second group consists of the axioms defining the abstract function *Stable* needed to model the failure stability property. Here stability means that, once a failure occurred, the value of the variable representing this failure remains unchanged until the whole system is restarted. These groups are independent because they refer to distinct Event-B constants and sets. The corresponding theorems **thm_axm1** and **thm_axm2** are shown below. The first theorem verifies that the parameters of the steam boiler are introduced in the model correctly:

$$
\begin{aligned}
\textbf{thm\_axm1:}\ & \exists\, \mathit{N1}, \mathit{N2}, \mathit{M1}, \mathit{M2}, \mathit{C}, \mathit{WL\_critical} \cdot \mathit{N1} \in \mathbb{N}1 \ \wedge\ \mathit{N2} \in \mathbb{N}1 \ \wedge \\
& \mathit{M1} \in \mathbb{N}1 \wedge \mathit{M2} \in \mathbb{N}1 \wedge \mathit{C} \in \mathbb{N}1 \wedge \mathit{WL\_critical} \in \mathbb{N} \times \mathbb{N} \rightarrow \mathit{BOOL} \ \wedge \\
& 0 < \mathit{M1} \ \wedge\ \mathit{M1} < \mathit{N1} \ \wedge\ \mathit{N1} < \mathit{N2} \ \wedge\ \mathit{N2} < \mathit{M2} \ \wedge\ \mathit{M2} < \mathit{C} \wedge \\
& (\forall\, x, y \cdot x \in \mathbb{N} \ \wedge\ y \in \mathbb{N} \wedge \\
& ((x < \mathit{M1} \vee y > \mathit{M2}) \Leftrightarrow \mathit{WL\_critical}(x \mapsto y) = \mathit{TRUE})) \wedge \\
& (\forall\, x, y \cdot x \in \mathbb{N} \ \wedge\ y \in \mathbb{N} \wedge \\
& ((x \geq \mathit{M1} \ \wedge\ y \leq \mathit{M2}) \Leftrightarrow \mathit{WL\_critical}(x \mapsto y) = \mathit{FALSE})).
\end{aligned}
$$

The second theorem verifies that the group of axioms introduced to define a function about the failure stability is consistent:

$$
\begin{aligned}
\textbf{thm\_axm2:}\ & \exists\, \mathit{Stable} \cdot \mathit{Stable} \in \mathit{BOOL} \times \mathit{BOOL} \rightarrow \mathit{BOOL} \wedge \\
& (\forall\, x, y \cdot x \in \mathit{BOOL} \ \wedge\ y \in \mathit{BOOL} \ \Rightarrow \\
& (\mathit{Stable}(x \mapsto y) = \mathit{TRUE} \ \Leftrightarrow\ (x = \mathit{TRUE} \ \Rightarrow\ y = \mathit{TRUE}))).
\end{aligned}
$$

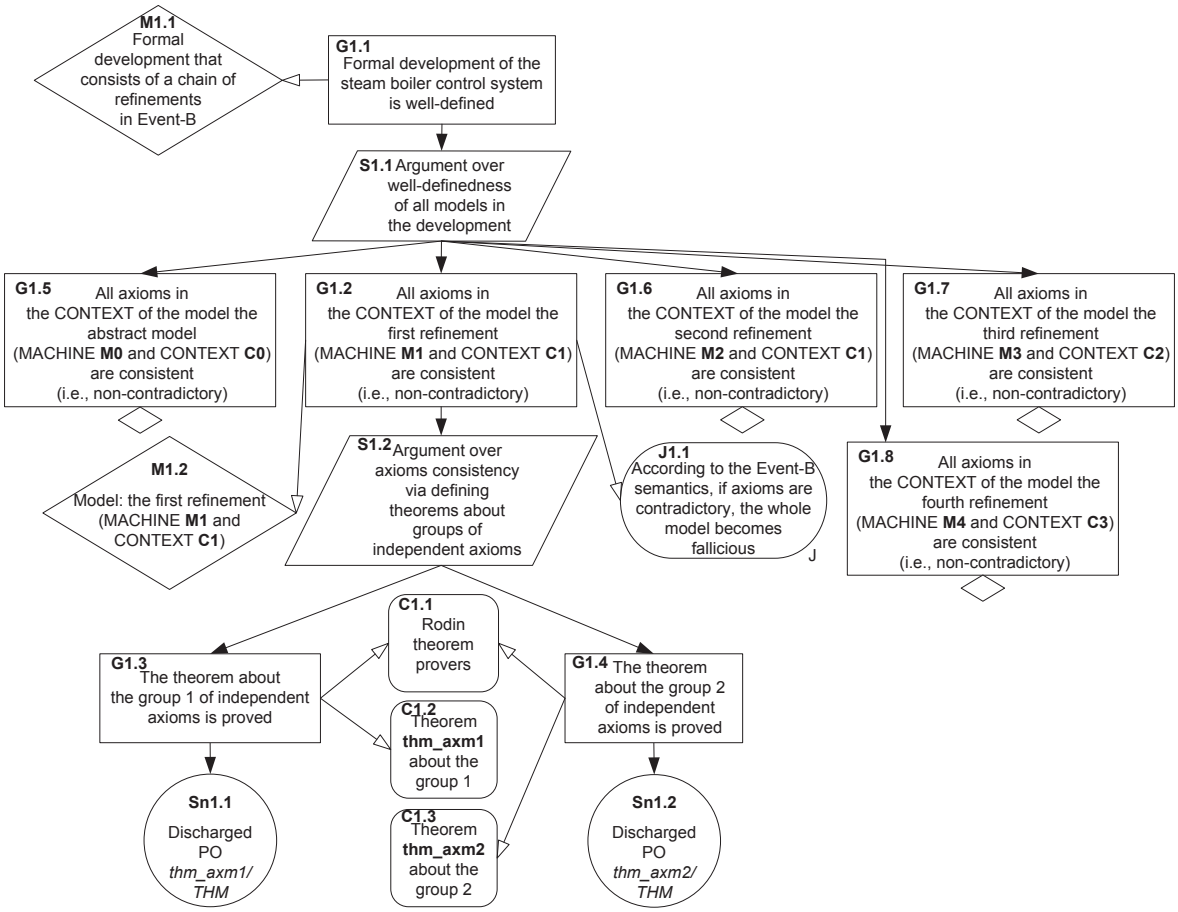

Figure 27: A fragment of the safety case corresponding to well-definedness of the development

The obtained proofs of these theorems are included in the safety case as the solutions **Sn1.1** and **Sn1.2** correspondingly.

5.3.2 Instantiation of the argument pattern for Class 1

The steam boiler control system has a large number of safety requirements [55]. Among them there are several requirements that can be classified as SRs belonging to *Class 1*. Let us demonstrate the instantiation of the corresponding argument pattern by the example of one such a safety requirement:

> **SR-02** : *During the system operation the water level shall not exceed the predefined safety boundaries.*

We formalise it as the invariant **inv1.2** at the first refinement step of the Event-B development (MACHINE **M1**):

inv1.2: $failure = FALSE \ \wedge \ phase \neq ENV \ \wedge \ phase \neq DET \ \Rightarrow$
$min\_water\_level \geq M1 \ \wedge \ max\_water\_level \leq M2,$

where the variable *failure* represents a system failure, the variable *phase* models the stages of the steam boiler controller behaviour (i.e., the stages of its control loop), and finally the variables *min_water_level* and *max_water_level* represent the estimated interval for the sensed water level.

The mapping function F_M for this case is

$$\textbf{SR-02} \mapsto \{\textbf{inv1.2}\},$$

which is a concrete instance of its general form *Requirement* $\mapsto \{safety_1, ..., safety_N\}$ for *Class 1* given in Section 4.2.

To provide evidence that this safety requirement is met by the system, we instantiate the argument pattern for *Class 1* as shown in Figure 28.

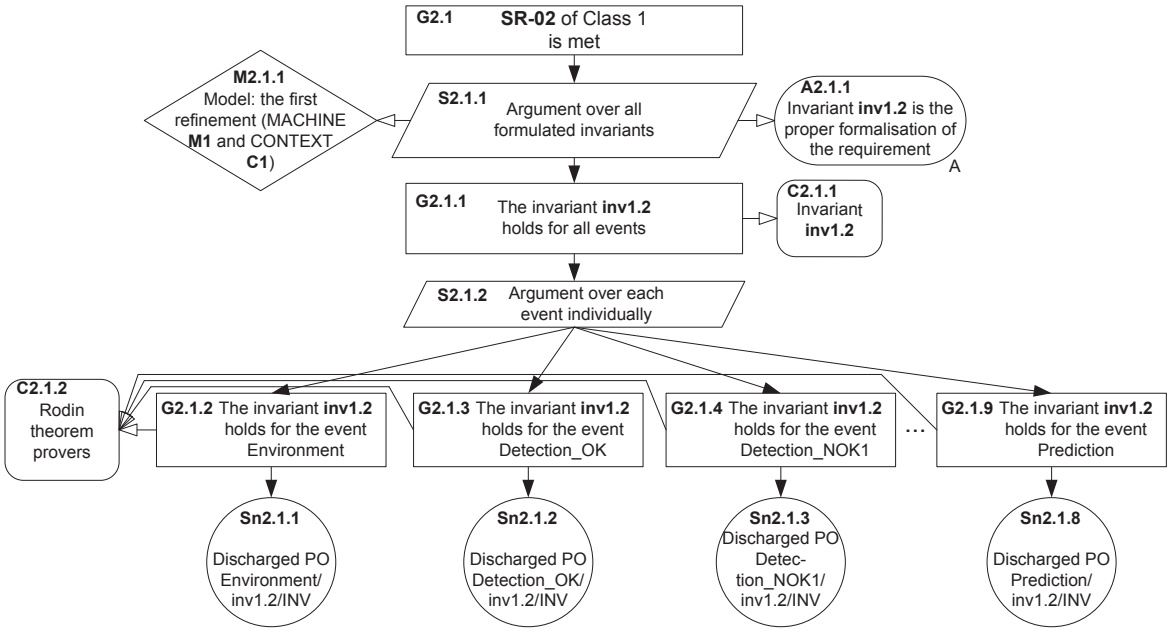

Figure 28: A fragment of the safety case corresponding to assurance of **SR-02**

The list of affected model events where this invariant must hold is the following: *Environment*, *Detection_OK*, *Detection_NOK1*, *Detection_NOK2*, *PreOperational1*, *PreOperational2*, *Operational*, *Prediction*. To support the claim that **inv1.2** holds for all these events, we attach the discharged proof obligations as the evidence. For brevity, we present only the supporting evidence **Sn2.1.2** of the goal **G2.1.3** as shown in Figure 29. This discharged proof obligation ensures that **inv1.2** holds for the event *Detection_OK* modelling detection of no failures.

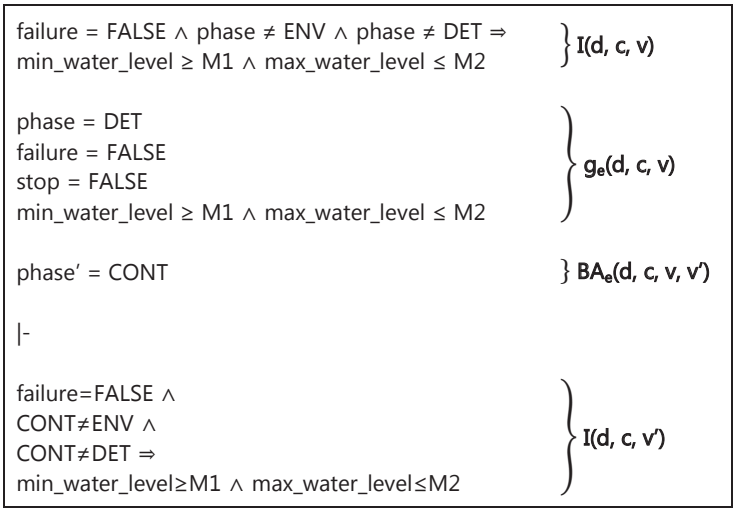

Figure 29: The proof obligation of the type INV for the event *Detection_OK* in **M1**

5.3.3 Instantiation of the argument pattern for Class 2

Since the steam boiler system is a failsafe system (i.e., it has to be put into a safe but non-operational state to prevent an occurrence of a hazard), whenever a system failure occurs, the system should be stopped. However, we abstractly model such failsafe procedures by assuming that, when the corresponding flag *stop* is raised thus indicating a system failure, the system is shut down and an alarm is activated. This condition is defined by the safety requirement **SR-01**:

> **SR-01** : *When a system failure is detected, the steam boiler control system shall be shut down and an alarm shall be activated.*

The stipulated property does not rely on a detailed representation of the steam boiler system and therefore can be incorporated at early stages of the development in Event-B, e.g., at the first refinement step (MACHINE **M1**). Since the property needs to be true at a specific state of the model, we classify this safety requirement as a SR belonging to *Class 2* and formalise it as the following theorem:

$$\begin{aligned}\textbf{thm1.1:}\ & \forall stop' \cdot stop' \in BOOL \wedge \\ & (\exists phase, stop \cdot phase \in \mathit{PHASE} \ \wedge \ stop \in BOOL \wedge \\ & phase = CONT \ \wedge \ stop = \mathit{FALSE} \ \wedge \ stop' = \mathit{TRUE}) \\ & \Rightarrow \\ & stop' = \mathit{TRUE},\end{aligned}$$

where $stop' = \mathit{TRUE}$ is a predicate defining the required post-condition of the event *EmergencyStop*.

The corresponding instance of the mapping function F_M for this class of safety requirements in this case is

$$\textbf{SR-01} \mapsto \{(\mathit{EmergencyStop}, stop' = \mathit{TRUE})\}.$$

The instantiated fragment of the safety case is presented in Figure 30. The proof obligation (*thm1.1/THM*) serves as the evidence that this requirement holds.

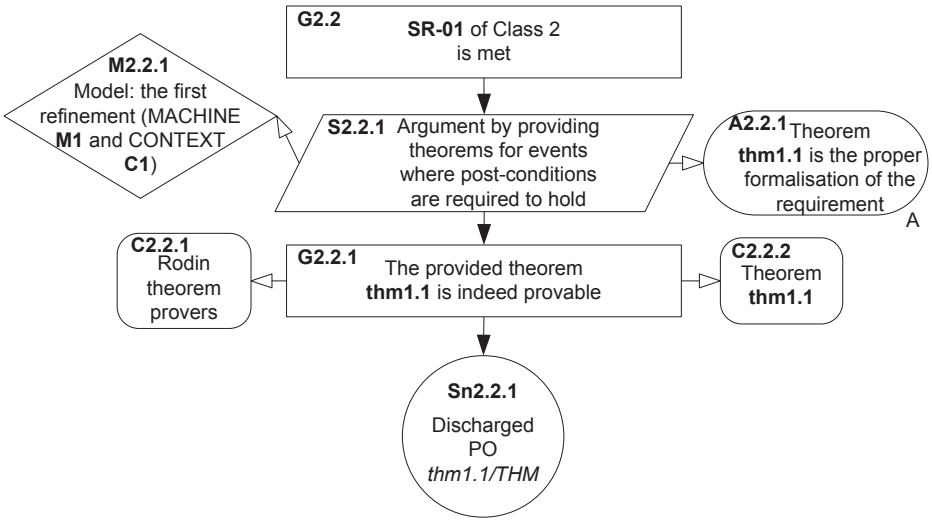

Figure 30: A fragment of the safety case corresponding to assurance of **SR-01**

5.3.4 Instantiation of the argument pattern for Class 3

Another safety-related property of the system under consideration is its cyclic behaviour. At each cycle the controller reads the sensors, performs computations and sets the actuators accordingly. Thus, the described control flow needs to be preserved by the Event-B model of the system as well. The safety requirement **SR-12** reflects the desired order in the control flow, associated with the corresponding order of the events in the Event-B model.

> **SR-12** : *The system shall operate cyclically. Each cycle it shall read the sensors, detect failures, perform either normal or degraded or rescue operation, or, in case of a critical system failure, stop the system, as well as compute the next values of variables to be used for failure detection at the next cycle if no critical system failure is detected.*

For the sake of simplicity, here we consider only the abstract model of the system (MACHINE **M0**). The refinement-based semantics of Event-B allows us to abstract away from detailed representation of the operational modes of the system (i.e., normal, degraded and rescue), ensuring nevertheless that the control flow properties proved at this step will be preserved by more detailed models.

We represent the required events order (**C2.3.2**) using the flow diagram (Figure 31). The generic mapping function *Requirement* $\mapsto \{(event_i, relationship, event_j)\}$ for *Class 3* can be instantiated in this case as

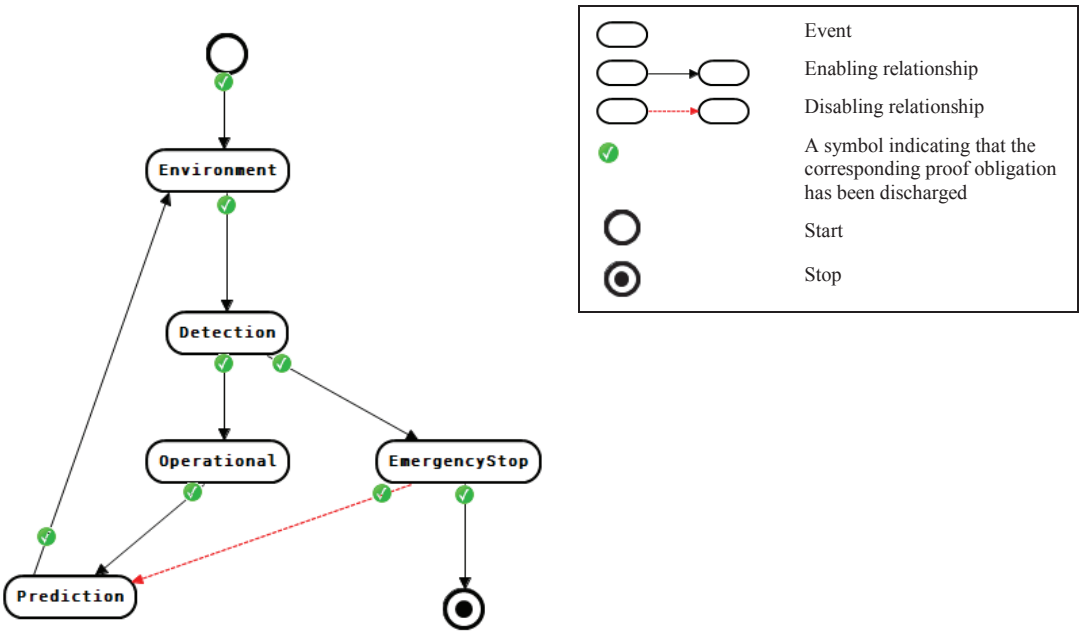

Figure 31: The flow diagram of the abstract MACHINE **M0**

$$\textbf{SR-12} \mapsto \{(\textit{Environment}, \textbf{ena}, \textit{Detection}), (\textit{Detection}, \textbf{ena}, \textit{Operational}), (\textit{Detection}, \textbf{ena}, \textit{EmergencyStop}), (\textit{Operational}, \textbf{ena}, \textit{Prediction}), (\textit{Prediction}, \textbf{ena}, \textit{Environment}), (\textit{EmergencyStop}, \textbf{dis}, \textit{Prediction})\}.$$

The instance of the pattern that ensures the order of the events in the MACHINE **M0** is presented in Figure 32. Due to the lack of space, we show only two proof obligations (Figure 33) discharged to support this branch of the safety case – *Environment/Detection/FENA* (**Sn2.3.1**) and *EmergencyStop/Prediction/ FDIS* (**Sn2.3.6**).

5.3.5 Instantiation of the argument pattern for Class 4

As we have already mentioned in Section 5.3.3, the steam boiler control system is a failsafe system. This means that it does contain a deadlock and therefore we do not need to construct the system safety case based on the argumentation defined by the pattern for *SRs about the absence of system deadlock* (*Class 4*). Quite opposite, we need to ensure that when the required sutdown condition is satisfied, the system terminates. Thus, we instantiate the pattern for *Class 5* instead.

5.3.6 Instantiation of the argument pattern for Class 5

Let us consider again the safety requirement **SR-01** given in Section 5.3.3:

SR-01 : *When a system failure is detected, the steam boiler control system shall be shut down and an alarm shall be activated.*

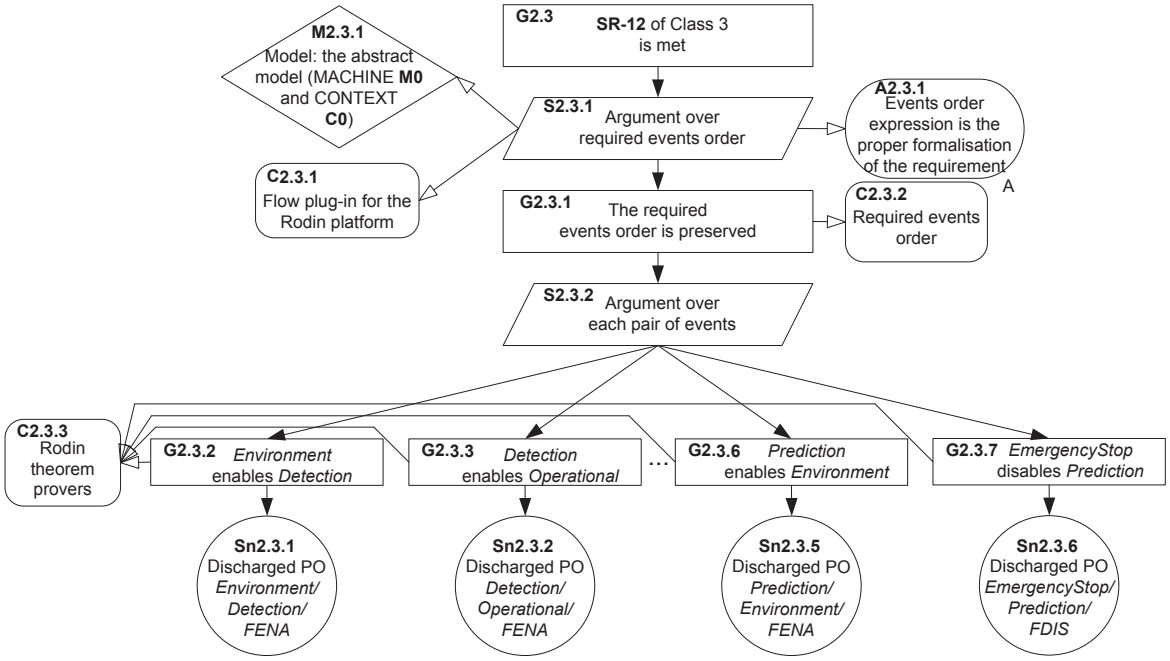

Figure 32: A fragment of the safety case corresponding to assurance of **SR-12**

phase ∈ PHASE
failure ∈ BOOL
stop ∈ BOOL
failure = FALSE ⇒ stop = FALSE
failure = TRUE ∧ phase ≠ CONT ⇒ stop=TRUE
phase=ENV
stop=FALSE
|-
DET=DET ∧
failure=FALSE ∧
stop=FALSE

} Environment/Detection/FENA

failure ∈ BOOL
stop ∈ BOOL
failure = FALSE ⇒ stop = FALSE
failure = TRUE ∧ phase ≠ CONT ⇒ stop = TRUE
phase = CONT
failure = TRUE
stop = FALSE
|-
¬ (phase = PRED ∧ TRUE = FALSE)

} EmergencyStop/Prediction/FDIS

Figure 33: The proof obligations of the types FENA and FDIS

The corresponding model theorem **thm1.1** (see Section 5.3.3) guarantees that the system variables are updated accordingly to prepare for a system shutdown, e.g., the stop flag is

raised. However, it does not ensure that the system indeed terminates, i.e., there are no enabled system events anymore. This should be done separately. Therefore, this safety requirement can be classified as a requirement belonging to both *Class 2* and *Class 5*. To show that our system definitely meets this requirement, we instantiate the argument pattern for *Class 5* as well (Figure 34).

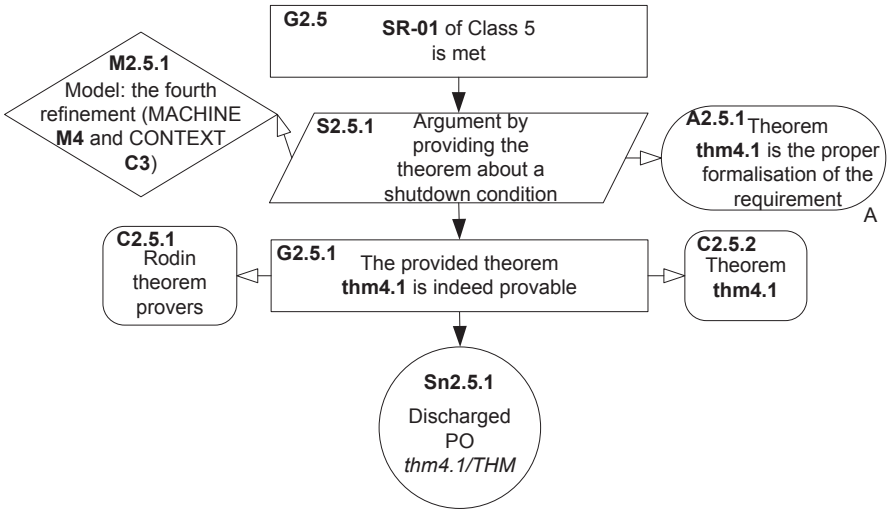

Figure 34: A fragment of the safety case corresponding to assurance of **SR-01**

In this case, the corresponding instance of the generic mapping function F_M for *Class 5* is

$$\textbf{SR-01} \mapsto \{stop = TRUE, Environment, Detection\_OK\_no\_F, ..., EmergencyStop, Prediction\},$$

where *stop = TRUE* stands for the required shutdown condition.

Then, the corresponding theorem **thm4.1** is formulated as follows:

$$\textbf{thm4.1:}\ stop = TRUE \ \Rightarrow\ \neg(before(Environment) \vee before(Detection\_OK\_no\_F) \vee .. \vee before(EmergencyStop) \vee before(Prediction)),$$

which in turn can be rewritten (by expanding the definition of *before(e)* described in detail in Section 2.1) as:

$$\begin{aligned}\textbf{thm4.1:}\ & stop = TRUE \Rightarrow \neg((stop = FALSE \wedge phase = ENV \wedge ..) \vee \\ & (stop = FALSE \wedge phase = DET \wedge ..) \vee \ .. \vee \\ & (stop = FALSE \wedge phase = CONT \wedge ..) \vee \\ & (stop = FALSE \wedge phase = PRED \wedge ..));\end{aligned}$$

$$\begin{aligned}& stop = TRUE \Rightarrow \neg(stop = FALSE \wedge ((phase = ENV \wedge ..) \vee \\ & (phase = DET \wedge ..) \vee \ .. \vee \\ & (phase = CONT \wedge ..) \ \vee \ (phase = PRED \wedge ..)));\end{aligned}$$

$$stop = \mathit{TRUE} \Rightarrow \neg stop = \mathit{FALSE} \vee \neg((phase = ENV \wedge ..) \vee$$
$$(phase = DET \wedge ..) \vee \ .. \vee$$
$$(phase = CONT \wedge ..) \ \vee \ (phase = PRED \wedge ..));$$

$$stop = \mathit{TRUE} \Rightarrow stop = \mathit{TRUE} \vee \neg((phase = ENV \wedge ..) \vee$$
$$(phase = DET \wedge ..) \vee \ .. \vee$$
$$(phase = CONT \wedge ..) \ \vee \ (phase = PRED \wedge ..)).$$

The discharged proof obligation (*thm4.1/THM*) provides the evidence for validity of the claim **G2.5** (see Figure 34).

5.3.7 Instantiation of the argument pattern for Class 6

We demonstrate an application of the argument pattern for *Class 6* on a pair of hierarchically linked requirements for the steam boiler system.

The requirement **R-09-higher-level** describes general behaviour of the pump actuator in the operational system phase, which concerns safety of the system only implicitly:

R-09-higher-level : *In the operational phase of the system execution, the pump actuator can be switched on or off (based on the water level estimations), or stay in the same mode*,

while its more detailed counterpart (**SR-09-lower-level**) does this explicitly. It stipulates the behaviour of the system and the pump actuator in the presence of a pump actuator failure:

SR-09-lower-level : *When the pump actuator fails, it shall stay in its current mode.*

In our formal development, these requirements are also introduced gradually at different refinement steps. More specifically, the first one is formalised at the first refinement step (MACHINE **M1**), while the second one is incorporated at the second refinement step (MACHINE **M2**).

We consider both requirements as requirements belonging to *Class 2*. Therefore, their verification is done by proving the corresponding theorems about post-states of specific events. Here we assume that the corresponding separate fragments of the safety case have been constructed using the argument pattern for *Class 2* to guarantee that the requirements **R-09-higher-level** and **SR-09-lower-level** hold. However, in this section we leave out these fragments of the safety case while focusing on ensuring the hierarchical consistency between these requirements. In other words, we focus on application of the argument pattern for *Class 6*.

The correctness of the hierarchical link between the requirements **R-09-higher-level** and **SR-09-lower-level** is guaranteed via operation refinement of the affected events belonging to MACHINE **M1** and MACHINE **M2** correspondingly. In this particular case, these are the abstract event *Operational* in **M1** and its refinement – the event *Degraded_Operational* in **M2**. The events are presented in Figure 35.

```
// Event in the MACHINE M1
event Operational refines Operational
  where
    @grd1 phase = CONT
    @grd2 failure = FALSE
    @grd3 stop = FALSE
    @grd4 preop_flag = FALSE
    @grd5 min_water_level ≥ M1 ∧
          max_water_level ≤ M2
  then
    @act1 phase ≔ PRED
    @act2 pump_ctrl :| pump_ctrl' ∈ PUMP_MODE ∧
          (pump_ctrl' = pump_ctrl ∨
          ((min_water_level ≥ M1 ∧ max_water_level < N1 ⇒
          pump_ctrl' = ON) ∧
          (min_water_level > N2 ∧ max_water_level ≤ M2 ⇒
          pump_ctrl' = OFF) ∧
          (min_water_level ≥ N1 ∧ max_water_level ≤ N2 ⇒
          pump_ctrl' = pump_ctrl)))
end
```

```
// Event in the refined MACHINE M2
event Degraded_Operational refines Operational
  where
    @grd1 phase = CONT
    @grd3 stop = FALSE
    @grd4 preop_flag = FALSE
    @grd6 wl_sensor_failure = FALSE ∧
          (pump_failure = TRUE ∨ so_sensor_failure = TRUE)
    @grd7 valve_ctrl = CLOSED
    @grd8 WL_critical(min_water_level ↦
                      max_water_level) = FALSE
  then
    @act1 phase ≔ PRED
    @act2 pump_ctrl :| pump_ctrl' ∈ PUMP_MODE ∧
          (pump_failure = TRUE ⇒ pump_ctrl' = pump_ctrl) ∧
          (pump_failure = FALSE ∧ min_water_level ≥ M1 ∧
          max_water_level < N1 ⇒ pump_ctrl' = ON) ∧
          (pump_failure = FALSE ∧ min_water_level > N2 ∧
          max_water_level ≤ M2 ⇒ pump_ctrl' = OFF) ∧
          (pump_failure = FALSE ∧ min_water_level ≥ N1 ∧
          max_water_level ≤ N2 ⇒ pump_ctrl' = pump_ctrl)
end
```

Figure 35: Events *Operational* and *Degraded_Operatinal*

In the MACHINE **M1**, we abstractly model a system failure by the variable *failure*. Then, in the MACHINE **M2**, we substitute this abstract variable and introduce the variables standing for failures of the system components, namely, the water level sensor failure – the variable *wl_sensor_failure*, the pump failure – the variable *pump_failure*, and the steam output sensor failure – the variable *so_sensor_failure*. The precise formal relationships between these new variables and the old one is depicted by the respective gluing invariant. In other words, the gluing invariant added to the MACHINE **M2** relates these concrete variables with the variable *failure* modelling an abstract failure.

The described data refinement directly affects the considered events *Operational* and *Degraded_Operational*. To guarantee that the refinement of the variable *failure* in the event *Degraded_Operational* does not weaken the corresponding guard of the event *Operational*, i.e., **grd2**, the proof obligation of the type GRD is discharged (see Section 4.7). Moreover, to satisfy the requirement **SR-09-lower-level**, we modify the action **act2** as shown in Figure 35. The correctness of this kind of simulation is guaranteed by the proof obligation of the type SIM. This pair of discharged proof obligations serves as the evidence that the consistency relationship between the corresponding hierarchically linked requirements is preserved by refinement.

The resulting instance of the argument pattern is shown in Figure 36. Here the mentioned proof obligations are attached as the safety case evidence – **Sn2.6.1** and **Sn2.6.2** respectively. Due to the large size, we do not show the details of these proof obligations in this paper.

5.3.8 Instantiation of the argument pattern for Class 7

To demonstrate an instantiation of the argument pattern for *Class 7* (i.e., a class representing safety requirements about temporal properties), we consider the following safety requirement

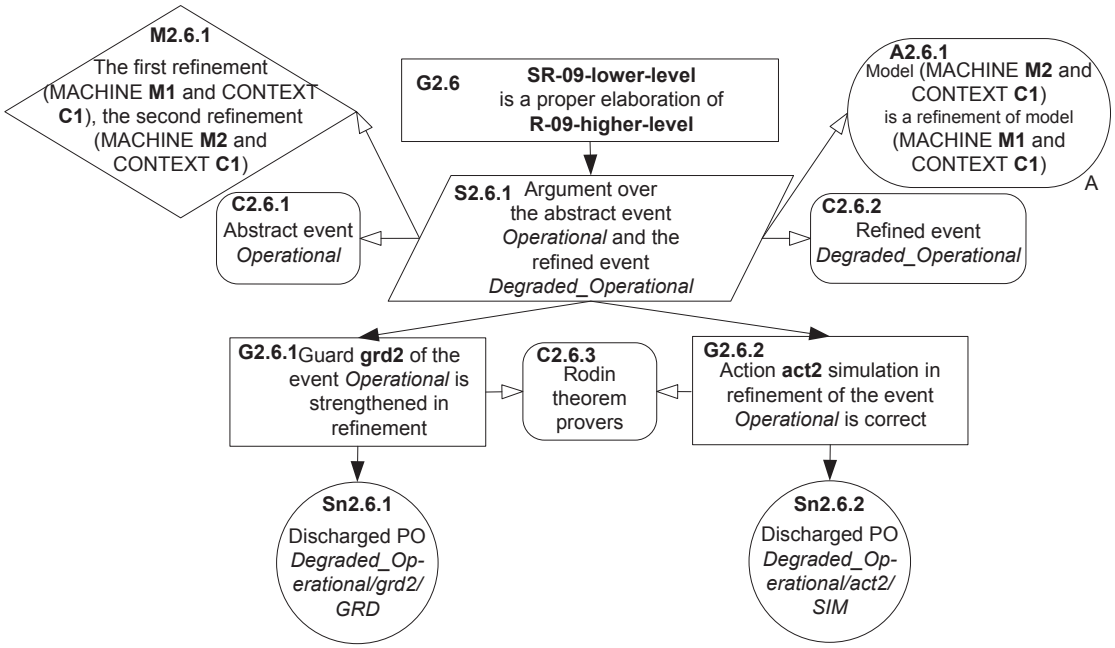

Figure 36: A fragment of the safety case corresponding to assurance of hierarchical requirements **R-09-higher-level** and **SR-09-lower-level**

of the steam boiler system:

SR-13 : *If there is no system failure, the system shall continue its operation in a new cycle.*

In our Event-B specification of the steam boiler system, the new cycle starts when the system enables the event *Environment* (Figure 31). Therefore, we have to show that, whenever no failure is detected in the detection phase, the system will start a new cycle by eventually reaching the event *Environment*. According to our pattern, we associate the requirement **SR-13** with a temporal reachability property. The corresponding instance of the generic mapping function F_M for *Class 7* in this case is

$$\textbf{SR-13} \mapsto \{\textbf{temp\_property}\},$$

where **temp_property** is an LTL formula defined as

$$\textbf{temp\_property:}\ \Box\,(\mathit{after}(\mathit{Detection})\ \wedge\ \mathit{failure} = \mathit{FALSE}\ \rightarrow\ \Diamond\,\mathit{before}(\mathit{Environment})).$$

This formula has the following representation in the ProB plug-in:

$$\begin{array}{l} G\,(\{(\forall phase', failure'\ \cdot\ phase' \in \mathit{PHASE}\ \wedge\ failure' \in BOOL\ \wedge \\ \quad (\exists\, phase, stop, failure\ \cdot\ phase \in \mathit{PHASE}\ \wedge\ stop \in BOOL\ \wedge \\ \quad failure \in BOOL\ \wedge\ phase = DET\ \wedge\ failure = \mathit{FALSE}\ \wedge \\ \quad stop = \mathit{FALSE})\ \wedge\ phase' = CONT\ \wedge\ failure' \in BOOL)\ \wedge \\ \quad failure = \mathit{FALSE}\} \\ \quad \Rightarrow \\ \quad F\,\{phase = ENV\ \wedge\ stop = \mathit{FALSE}\}). \end{array}$$

As a result of the model checking on this property, ProB yields the following outcome: *"no counter-example has been found, all nodes have been visited"*. Therefore, we can attach this result as the evidence for the corresponding fragment of our safety case (**Sn2.7.1**). The resulting instance of the argument pattern is shown in Figure 37.

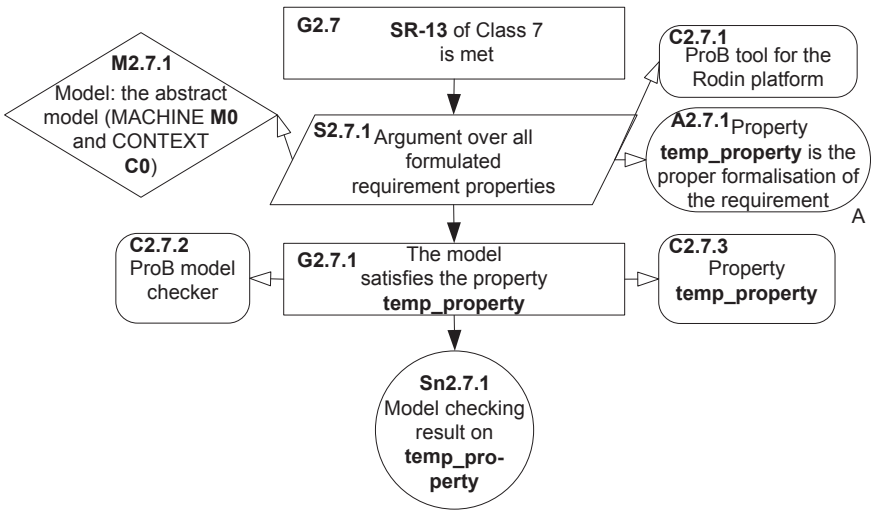

Figure 37: A fragment of the safety case corresponding to assurance of **SR-13**

5.3.9 Instantiation of the argument pattern for Class 8

We did not take into account timing constraints imposed on the steam boiler control system while developing the formal system specification in Event-B. Therefore, we could not support the system safety case with a fragment associated with the safety requirements about timing properties (*Class 8*).

5.4 Discussion on the application of the approach

Despite the fact that the accomplished Event-B development of the steam boiler control system is quite complicated and, as a result, a significant number of proof obligations has been discharged, we have not been able to instantiate two argument patterns, namely the patterns for *Class 4* and *Class 8*. First of all, the steam boiler control system is a failsafe system, which means that there is a deadlock in its execution. Consequently, there are no requirements about the absence of system deadlock (*Class 4*). Second of all, timing properties (*Class 8*) were not a part of the given system requirements either. Nevertheless, the presented guidelines on the instantiation of the argument patterns have allowed us to easily construct the corresponding fragments of the system safety case for the remaining safety requirements as well as to demonstrate well-definedness of the overall development of the system.

The use of the Rodin platform and accompanying plug-ins has facilitated derivation of the proof- and model checking-based evidence that the given safety requirements hold for the modelled system. The proof-based semantics of Event-B (a strong relationship between

model elements and the associated proof obligations) has given us a direct access to the corresponding proof obligations. It has allowed us to not just claim that the verified theorems were proved but also explicitly include the obtained proof obligations into the resulting safety case.

6 Related work

In this section, we overview related contributions according to the following three directions: firstly, we consider the publications on the use of formal methods for safety cases; secondly, we overview the works that aim at formalising safety requirements; and thirdly, we take a closer look at the approaches focusing on argument patterns.

Formal methods in safety cases. There are two main research directions in utilising formal methods for safety cases. On the one hand, a safety case argument itself can be formally defined and verified. On the other hand, safety requirements can be formalised and formally verified allowing us to determine the safety evidence such as the obtained results of static analysis, model checking or theorem proving. Note that such evidence corresponds to the class of safety evidence called *formal verification results* defined in the safety evidence taxonomy proposed by Nair et al. in [51].

In the former case, soundness of a safety argument can be proved by means of theorem proving in the classical or higher order logic, e.g., using the interactive theorem prover PVS [33, 57]. In particular, Rushby [57] formalises a top-level safety argument to support automated checking of soundness of a safety argument. He proposes to represent a safety case argument in the classical logic as a theorem where antecedents are the assumptions under which a system (or design) satisfies the consequent, whereas the consequent is a specific claim in the safety case that has to be assured. Then, such a theorem can be verified by an automated interactive theorem prover or a model checker.

In the latter case, soundness of an overall safety case is not formally examined. The focus is rather put on the evidence derived from formal analysis to show that the specific goals reflecting safety requirements are met. For example, to support the claim that the source code of a program module does not contain potentially hazardous errors, the authors of [32] use as the evidence the results of static analysis of program code. In [8,9], the authors assure safety of automatically generated code by providing formal proofs as the evidence. They ensure that safety requirements hold in specific locations of software system implementations. In [22, 23], the authors automate generation of heterogeneous safety cases including a manually developed top-level system safety case, and lower-level fragments automatically generated from the formal verification of safety requirements. According to this approach, the implementation is formally verified against a mathematical specification within a logical domain theory. This approach is developed for the aviation domain and illustrated by an unmanned aircraft system. To ensure that a model derived during model-driven development of a safety critical system, namely pacemaker, satisfies all the required properties, the authors of [43] use the obtained model checking results. Our approach proposed in this paper also belongs to this category. Formalisation and verification of safety requirements of

a critical system allows us to obtain the proof- and model checking-based evidence that these requirements hold.

Formalisation of safety requirements. Incorporation of requirements in general, and safety requirements in particular, into formal specifications is considered to be a challenging task. We overview some recent approaches that address this problem dividing them into two categories: those that aim at utilising model checking for verification of critical properties, and those that employ theorem proving for this purpose.

For example, a formalisation of safety requirements using the Computation Tree Logic (CTL) and then verification of them using a model checker is presented in [14]. The author classifies the given requirements associating them with the corresponding CTL formulas. A similar approach is presented in [35]. Here safety properties defined as LTL formulas are verified by using the SPIN model checker.

In contrast, the authors of [16] perform a systematic transformation of a Petri net model into an abstract B model for verification of safety properties by theorem proving. Another work that aims at verifying safety requirements by means of theorem proving is presented in [46]. The authors incorporate the given requirements into an Event-B model via applying a set of automated patterns, which are based on Failure Modes and Effects Analysis (FMEA).

Similarly to these works, we take an advantage of using theorem proving and a refinement-based approach to formal development of a system. We gradually introduce the required safety properties into an Event-B model and verify them in the refinement process. This allows us to avoid the state explosion problem commonly associated with model checking, thus making our approach more scalable for systems with higher levels of complexity. Nonetheless, in this paper, we also rely on model checking for those properties that cannot be verified by our framework directly.

Furthermore, there are other works that aim at formalising safety requirements, specifically in Event-B [41, 42, 49, 59]. Some of them propose to incorporate safety requirements as invariants and before-after predicates of events [41, 59], while others, e.g., [49], represent them as invariants or theorems only. Moreover, all these works show the correspondence between some particular requirements and the associated elements of the Event-B structure. However, they neither classify the safety requirements nor give precise guidelines for formal verification of those requirements that cannot be directly verified by the Event-B framework. In contrast, to be able to argue over each given safety requirement by relying on its formal representation, we propose a classification of safety requirements and define a precise mapping of each class onto a set of the corresponding model expressions. Moreover, for some of these classes, we propose bridging Event-B with other tools (model checkers).

Argument patterns. In general, argument patterns (or safety case patterns) facilitate construction of a safety case by capturing commonly used structures and allowing for simplified instantiation. Safety case patterns have been introduced by Kelly and McDermid [45] and received recognition among safety case developers. In [20], the authors give a formal definition for safety case patterns, define formal semantics of patterns, and propose a generic data model and algorithm for pattern instantiation. For example, a safety case pattern for arguing the correctness of implementations developed from a timed automata model using a model-based development approach has been presented in [6]. An instantiation of this pattern

has been illustrated on the implementation software of a patient controlled analgesic infusion pump. In [7], the author proposes a set of property-oriented and requirement-oriented safety case patterns for arguing safe execution of automatically generated program code with respect to the given safety properties as well as safety requirements. Additionally, the author defines architecture-oriented patterns for safety-critical systems developed using a model-based development approach.

An approach to automatically integrating the output generated by a formal method or tool into a software safety assurance case as an evidence argument is described in [21]. To capture the reasoning underlying a tool-based formal verification, the authors propose specific safety case patterns. The approach is software-oriented. A formalised requirement is verified to hold at a specific location of code. The proposed patterns allow formal reasoning and evidence to be integrated into the language of assurance arguments. Our approach is similar to the approach presented in [21]. However, we focus on formal system models rather that the code. Moreover, the way system safety requirements are formalised and verified in Event-B varies according to the proposed classification of safety requirements. Consequently, the resulting evidence arguments are also different. Nevertheless, we believe that the approach given in [21] can be used to complement our approach.

In this paper, we contribute to a set of existing safety case patterns and describe in detail their instantiation process for different classes of safety requirements. Moreover, our proposed patterns facilitate construction of safety cases where safety requirements are verified formally and the corresponding formal-based evidence is derived to represent justification of safety assurance. The evidence arguments obtained by applying our approach explicitly reflect the formal reasoning instead of just references to the corresponding proofs or the model checking results.

7 Conclusions

In this paper, we propose a methodology supporting rigorous construction of safety cases from formal Event-B models. It guides the developers starting from informal representation of safety requirements to building the corresponding parts of safety cases via formal modelling and verification in Event-B and the accompanying toolsets.

We believe that the proposed methodology has shown good scalability. In this paper, we have illustrated the application of our methodology both by small examples and a larger case study without major difficulties. Moreover, we have applied the methodology in two different situations: when formal models of systems were developed beforehand, and when the development was performed in parallel with the construction of the associated safety case. Specifically, all the formal models for illustrating the argument patterns in Section 4 were taken as given, while the formal development of the steam boiler system (presented in our previous work [55] and partially in Section 5.3) was done taking into account the proposed classification of safety requirements and the need to produce a safety case of the system. We have additionally observed the fact that, to construct an adequate safety case of a system based on its formal model, a *feedback loop* between two processes, namely,

the process of formal system development and construction of safety cases, is required. It means that, if construction of a safety case indicates that the associated formal model is "weak", i.e., it does not contain an adequate formalisation of some safety requirements that need to be demonstrated in the safety case, the developers should be able to react on that by improving the model.

Our main contribution, namely, the proposed methodology for rigorous construction of safety cases, has led us to achieving the following two sub-contributions. Firstly, we have classified safety requirements and shown how they can be formalised in Event-B. To attain this, we have proposed a strict mapping between the given safety requirements and the associated elements of formal models, thus establishing a clear traceability of those requirements. Secondly, we have proposed a set of argument patterns based on the proposed classification, the use of which facilitates the construction of safety cases. Due to the strong relationship between model elements and the associated proof obligations provided by the proof-based semantics of Event-B, we have been able to formally verify the mapped safety requirements and derive the corresponding proofs. Moreover, via developing the argument strategies based on formal reasoning and using the resulting proofs as the evidence for a safety case, we have achieved additional assurance that the desired safety requirements hold.

Furthermore, application of the well-defined Event-B theory for formal verification of safety requirements and formal-based construction of safety cases has clarified the use of particular safety case goals or strategies. It has allowed us to omit the additional explanations why the defined strategies are needed and why the proposed evidence is relevant. Otherwise, we would have needed to extend each proposed argument pattern with multiple instances of a specific GSN element called *justification* [28]. Consequently, this would have led to a significant growth of already large safety cases.

In this work, we have focused on safety aspects however the proposed approach can be extended to cover other dependability attributes, e.g., reliability and availability. We also believe that the generic principles described in this paper by the example of the Event-B formalism are applicable to any other formalism defined as a state transition system, e.g., B, Z, VDM, refinement calculus, etc.

So far, all the proposed patterns and their instantiation examples have been developed manually. However, the larger a system under consideration is, the more difficult this procedure becomes. Therefore, the necessity of automated tool support is obvious. We consider development of a dedicated plug-in for the Rodin platform as a part of our future work. Moreover, the proposed classification of the safety requirements is by no means complete. Consequently, it could be further extended with some new classes and the corresponding argument patterns.

Acknowledgements

Yuliya Prokhorova's work is partially supported by the Foundation of Nokia Corporation. Additionally, the authors would like to thank Prof. Michael Butler for the valuable feedback on the requirements classification.

References

[1] J.-R. Abrial. Steam-Boiler Control Specification Problem. In *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995)*, pages 500–509, London, UK, 1996. Springer-Verlag.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[3] J.-R. Abrial. Controlling Cars on a Bridge. http://deploy-eprints. ecs.soton.ac.uk/112/, April 2010.

[4] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[5] J.-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science*, pages 83–128. Springer Berlin Heidelberg, 1998.

[6] A. Ayoub, B.G. Kim, I. Lee, and O. Sokolsky. A Safety Case Pattern for Model-Based Development Approach. In *Proceedings of the 4th International Conference on NASA Formal Methods (NFM'12)*, pages 141–146, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] N. Basir. *Safety Cases for the Formal Verification of Automatically Generated Code*. Doctoral thesis, University of Southampton, 2010.

[8] N. Basir, E. Denney, and B. Fischer. Constructing a Safety Case for Automatically Generated Code from Formal Program Verification Information. In M.D. Harrison and M.-A. Sujan, editors, *Computer Safety, Reliability, and Security*, volume 5219 of *Lecture Notes in Computer Science*, pages 249–262. Springer Berlin Heidelberg, 2008.

[9] N. Basir, E. Denney, and B. Fischer. Deriving Safety Cases from Automatically Constructed Proofs. In *Systems Safety 2009. Incorporating the SaRS Annual Conference, 4th IET International Conference on*, pages 1–6, 2009.

[10] N. Basir, E. Denney, and B. Fischer. Deriving Safety Cases for Hierarchical Structure in Model-Based Development. In E. Schoitsch, editor, *Computer Safety, Reliability, and Security*, volume 6351 of *Lecture Notes in Computer Science*, pages 68–81. Springer Berlin Heidelberg, 2010.

[11] G. Behrmann, A. David, and K.G. Larsen. A Tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004.

[12] J. Berthing, P. Boström, K. Sere, L. Tsiopoulos, and J. Vain. Refinement-Based Development of Timed Systems. In J. Derrick, S. Gnesi, D. Latella, and H. Treharne, editors, *Integrated Formal Methods*, volume 7321 of *Lecture Notes in Computer Science*, pages 69–83. Springer Berlin Heidelberg, 2012.

[13] P. Bishop and R. Bloomfield. A Methodology for Safety Case Development. In *Safety-Critical Systems Symposium, Birmingham, UK*. Springer-Verlag, 1998.

[14] F. Bitsch. Classification of Safety Requirements for Formal Verification of Software Models of Industrial Automation Systems. In *Proceedings of 13th International Conference on Software and Systems Engineering and their Applications (ICSSEA'00)*, Paris, France, 2000. CNAM.

[15] F. Bitsch. Safety Patterns - The Key to Formal Specification of Safety Requirements. In *Proceedings of the 20th International Conference on Computer Safety, Reliability and Security (SAFECOMP'01)*, pages 176–189, London, UK, UK, 2001. Springer-Verlag.

[16] P. Bon and S. Collart-Dutilleul. From a Solution Model to a B Model for Verification of Safety Properties. *Journal of Universal Computer Science*, 19(1):2–24, 2013.

[17] Claims, Arguments and Evidence (CAE). http://www.adelard.com/ asce/choosing-asce/cae.html, 2014.

[18] D. Cansell, D. Méry, and J. Rehm. Time Constraint Patterns for Event-B Development. In J. Julliand and O. Kouchnarenko, editors, *B 2007: Formal Specification and Development in B*, volume 4355 of *Lecture Notes in Computer Science*, pages 140–154. Springer Berlin Heidelberg, 2007.

[19] UK Ministry of Defence. 00-56 Safety Management Requirements for Defence Systems, 2007.

[20] E. Denney and G. Pai. A Formal Basis for Safety Case Patterns. In F. Bitsch, J. Guiochet, and M. Kaâniche, editors, *Computer Safety, Reliability, and Security*, volume 8153 of *Lecture Notes in Computer Science*, pages 21–32. Springer Berlin Heidelberg, 2013.

[21] E. Denney and G. Pai. Evidence Arguments for Using Formal Methods in Software Certification. In *Proceedings of IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'13)*, pages 375–380, 2013.

[22] E. Denney, G. Pai, and J. Pohl. Heterogeneous Aviation Safety Cases: Integrating the Formal and the Non-formal. In *Proceedings of the 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems (ICECCS 2012)*, pages 199–208, Washington, DC, USA, 2012. IEEE Computer Society.

[23] E.W. Denney, G.J. Pai, and J.M. Pohl. Automating the Generation of Heterogeneous Aviation Safety Cases. NASA Contractor Report NASA/CR-2011-215983, August 2011.

[24] EB2ALL - The Event-B to C, C++, Java and C# Code Generator. http://eb2all.loria.fr/, October 2013.

[25] European Committee for Electrotechnical Standardization (CENELEC). EN 50128 Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems. June 2011.

[26] Event-B and the Rodin Platform. http://www.event-b.org/, 2014.

[27] The Flow plug-in. http://iliasov.org/usecase/, 2014.

[28] Goal Structuring Notation Working Group. Goal Structuring Notation Standard. http://www.goalstructuringnotation.info/, November 2011.

[29] J. Groslambert. Verification of LTL on B Event Systems. In J. Julliand and O. Kouchnarenko, editors, *B 2007: Formal Specification and Development in B*, volume 4355 of *Lecture Notes in Computer Science*, pages 109–124. Springer Berlin Heidelberg, 2006.

[30] I. Habli and T. Kelly. A Generic Goal-Based Certification Argument for the Justification of Formal Analysis. *Electronic Notes in Theoretical Computer Science*, 238(4):27–39, September 2009.

[31] J. Hatcliff, A. Wassyng, T. Kelly, C. Comar, and P. Jones. Certifiably Safe Software-dependent Systems: Challenges and Directions. In *Proceedings of the Track on Future of Software Engineering (FOSE'14)*, pages 182–200, New York, NY, USA, 2014. ACM.

[32] R. Hawkins, I. Habli, T. Kelly, and J. McDermid. Assurance cases and prescriptive software safety certification: A comparative study. *Safety Science*, 59:55–71, 2013.

[33] H. Herencia-Zapana, G. Hagen, and A. Narkawicz. Formalizing Probabilistic Safety Claims. In M. Bobaru, K. Havelund, G.J. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 162–176. Springer Berlin Heidelberg, 2011.

[34] T.S. Hoang and J.-R. Abrial. Reasoning about Liveness Properties in Event-B. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 456–471. Springer Berlin Heidelberg, 2011.

[35] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[36] IEC61508. International Electrotechnical Commission. IEC 61508, functional safety of electrical/electronic/programmable electronic safety-related systems. April 2010.

[37] A. Iliasov. Use Case Scenarios as Verification Conditions: Event-B/Flow Approach. In *Proceedings of the 3rd International Workshop on Software Engineering for Resilient Systems (SERENE'11)*, pages 9–23, Berlin, Heidelberg, 2011. Springer-Verlag.

[38] A. Iliasov, L. Laibinis, E. Troubitsyna, A. Romanovsky, and T. Latvala. Augmenting Event B Modelling with Real-Time Verification. TUCS Technical Report 1006, 2011.

[39] A. Iliasov, L. Laibinis, E. Troubitsyna, A. Romanovsky, and T. Latvala. Augmenting Event-B Modelling with Real-Time Verification. In *Proceedings of Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA'12)*, pages 51–57, 2012.

[40] International Organization for Standardization. ISO 26262 Road Vehicles Functional Safety. November 2011.

[41] M. Jastram, S. Hallerstede, and L. Ladenberger. Mixing Formal and Informal Model Elements for Tracing Requirements. In *Electronic Communications of the EASST*, volume 46, 2011.

[42] M. Jastram, S. Hallerstede, M. Leuschel, and A.G. Russo Jr. An Approach of Requirements Tracing in Formal Refinement. In *Proceedings of the Third International Conference on Verified Software: Theories, Tools, Experiments (VSTTE'10)*, pages 97–111, Berlin, Heidelberg, 2010. Springer-Verlag.

[43] E. Jee, I. Lee, and O. Sokolsky. Assurance Cases in Model-Driven Development of the Pacemaker Software. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6416 of *Lecture Notes in Computer Science*, pages 343–356. Springer Berlin Heidelberg, 2010.

[44] T.P. Kelly. *Arguing Safety – A Systematic Approach to Managing Safety Cases*. Doctoral thesis, University of York, September 1998.

[45] T.P. Kelly and J.A. McDermid. Safety Case Construction and Reuse Using Patterns. In P. Daniel, editor, *Proceedings of the 16th International Conference on Computer Safety, Reliability and Security (SAFECOMP'97)*, pages 55–69. Springer-Verlag London, 1997.

[46] I. Lopatkin, Y. Prokhorova, E. Troubitsyna, A. Iliasov, and A. Romanovsky. Patterns for Representing FMEA in Formal Specification of Control Systems. TUCS Technical Report 1003, 2011.

[47] The ProB Animator and Model Checker. http://www.stups.uni-duesseldorf.de/ProB/index.php5/LTL_Model_ Checking, 2014.

[48] D. Méry. Requirements for a Temporal B Assigning Temporal Meaning to Abstract Machines ... and to Abstract Systems. In K. Araki, A. Galloway, and K. Taguchi, editors, *Integrated Formal Methods*, pages 395–414. Springer London, 1999.

[49] D. Méry and N.K. Singh. Technical Report on Interpretation of the Electrocardiogram (ECG) Signal using Formal Methods. Technical Report INRIA-00584177, 2011.

[50] C. Metayer, J.-R. Abrial, and L. Voisin. Event-B Language. Rigorous Open Development Environment for Complex Systems (RODIN) Deliverable 3.2. http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf, May 2005.

[51] S. Nair, J.L. de la Vara, M. Sabetzadeh, and L. Briand. Classification, Structuring, and Assessment of Evidence for Safety – A Systematic Literature Review. In *Proceedings of IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST'13)*, pages 94–103, 2013.

[52] The ProB Animator and Model Checker. http://www.stups.uni-duesseldorf.de/ProB/index.php5/Main_Page, 2014.

[53] Y. Prokhorova, L. Laibinis, E. Troubitsyna, K. Varpaaniemi, and T. Latvala. Deriving a mode logic using failure modes and effects analysis. *International Journal of Critical Computer-Based Systems*, 3(4):305—328, 2012.

[54] Y. Prokhorova and E. Troubitsyna. Linking Modelling in Event-B with Safety Cases. In P. Avgeriou, editor, *Software Engineering for Resilient Systems*, volume 7527 of *Lecture Notes in Computer Science*, pages 47–62. Springer Berlin Heidelberg, 2012.

[55] Y. Prokhorova, E. Troubitsyna, and L. Laibinis. A Case Study in Refinement-Based Modelling of a Resilient Control System. TUCS Technical Report 1086, 2013.

[56] Y. Prokhorova, E. Troubitsyna, L. Laibinis, D. Ilić, and T. Latvala. Formalisation of an Industrial Approach to Monitoring Critical Data. TUCS Technical Report 1070, 2013.

[57] J. Rushby. Formalism in Safety Cases. In C. Dale and T. Anderson, editors, *Making Systems Safer: Proceedings of the Eighteenth Safety-Critical Systems Symposium*, pages 3–17, Bristol, UK, 2010. Springer.

[58] M.R. Sarshogh and M. Butler. Specification and Refinement of Discrete Timing Properties in Event-B. In *Electronic Communications of the EASST*, volume 46, 2011.

[59] S. Yeganefard and M. Butler. Structuring Functional Requirements of Control Systems to Facilitate Refinement-based Formalisation. In *Electronic Communications of the EASST*, volume 46, 2011.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 A, 20520 TURKU, Finland | www.tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics

Turku School of Economics
- Institute of Information Systems Sciences

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

ISBN 978-952-12-3064-6
ISSN 1239-1891