

From Requirements Engineering to Safety Assurance: Refinement Approach

Linus Laibinis¹✉, Elena Troubitsyna¹, Yuliya Prokhorova², Alexei Iliasov³,
and Alexander Romanovsky³

¹ Åbo Akademi University, Turku, Finland
{linas.laibinis,elena.troubitsyna}@abo.fi

² Space Systems Finland, Espoo, Finland
yuliya.prokhorova@ssf.fi

³ Newcastle University, Newcastle Upon Tyne, UK
{alexei.iliasov,alexander.romanovsky}@ncl.ac.uk

Abstract. Formal modelling and verification are widely used in the development of safety-critical systems. They aim at providing a mathematically-grounded argument about system safety. In particular, this argument can facilitate construction of a safety case – a structured safety assurance document required for certification of safety-critical systems. However, currently there is no adequate support for using the artefacts created during formal modelling in safety case development. In this paper, we present an approach and the corresponding tool support that tackles this problem in the Event-B modelling framework. Our approach establishes a link between safety requirements, Event-B models and corresponding fragments of a safety case. The supporting automated tool ensures traceability between requirements, models and safety cases.

1 Introduction

Formal techniques provide the designers with a rigorous mathematical basis for reasoning about the system behaviour and properties. Usually formal modelling helps in uncovering problems in requirements definition as well as deriving additional constraints for ensuring safety. Though formal modelling provide the designers with a valuable input for safety assurance, currently there is no adequate support for integrating the results of formal modelling into construction of safety argument – a safety case. In this paper, we propose a method and tool support for linking formal modelling in Event-B and safety case construction.

Event-B is a formal framework for correct-by-construction system development [1]. It has been extensively experimented with in the industrial setting [2,3]. The framework employs refinement as the main development technique and proofs to verify correctness of the system behaviour with respect to system-level properties, such as safety. The industrial-strength tool support – the Rodin platform [4] – provides the developers with highly automated environment for modelling and verification.

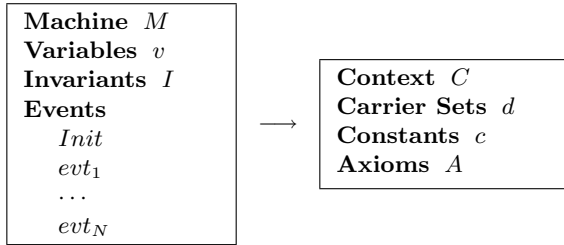


Fig. 1. Event-B machine and context

To efficiently exploit the benefits of formal modelling, in this paper we present an automated integrated approach that facilitates construction of safety cases from Event-B models. The approach spans over requirements engineering, formal modelling and safety argumentation via safety cases. While automating the proposed approach, we aim at creating a non-obtrusive tool support that nevertheless allows us to maintain the link between the dynamically changing requirements, models and safety cases. To achieve this goal, we relied on a novel industry-driven standard OSLC – *Open Services for Life Cycle Collaborations* [5]. The standard allows the engineers to achieve inter-operability between engineering tools by specifying the access to the external resources of these tools. We believe that the proposed approach has two main benefits: it supports co-engineering of requirements, models and safety cases, while the tool support ensures seamless interoperability and traceability across the domains.

The paper is structured as follows. In Section 2, we describe our chosen formal framework – Event-B as well as discuss classification and formalisation of safety requirements in Event-B. Section 3 presents our methodology for constructing safety cases from requirements and artefacts of formal modelling. In Section 4, we present the steam boiler case study demonstrating the proposed methodology. Section 5 presents our proposal on dynamic tool integration in a common information environment. Finally, in Section 6, we overview the related work and give some concluding remarks.

2 Modelling and Verification of Safety-Critical Systems in Event-B

Event-B: Background. Event-B is a state-based framework that promotes the top-down, correct-by-construction approach to system development and formal verification by theorem proving. In Event-B, a system model is specified as an *abstract state machine* [1]. An abstract state machine encapsulates the model state, represented as a collection of variables, and defines operations on the state, i.e., it describes the dynamic behaviour of a modelled system. The variables are strongly typed by the constraining predicates that together with other important properties of the systems are defined in the model *invariants*. Usually, a machine has an accompanying component, called *context*, which includes user-defined sets, constants and their properties given as a list of model axioms.

A general form for Event-B models is given in Fig. 1. The machine is uniquely identified by its name M . The state variables, v , are declared in the **Variables** clause and initialised in the *Init* event. The variables are strongly typed by the constraining predicates I given in the **Invariants** clause. The invariant clause might also contain other predicates defining properties (e.g., safety invariants) that should be preserved during system execution.

The dynamic behaviour of the system is defined by a set of atomic *events*. Generally, an event has the following form:

$$e \hat{=} \mathbf{any } a \mathbf{ where } G_e \mathbf{ then } R_e \mathbf{ end,}$$

where e is the event's name, a is the list of local variables, the *guard* G_e is a predicate over the local variables of the event and the state variables of the system. The event body is defined by a *multiple* (possibly nondeterministic) assignment over the system variables. In Event-B, such an assignment represents the corresponding next-state relation R_e . The guard defines the conditions under which the event is *enabled*, i.e., its body can be executed. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

Event-B development starts from an abstract specification that nondeterministically models most essential functional requirements. In a sequence of refinement steps, we gradually reduce nondeterminism and introduce detailed design decisions. In particular, we can add new events, split events as well as replace abstract variables by their concrete counterparts, i.e., perform *data refinement*.

The consistency of Event-B models, i.e., verification of well-formedness and invariant preservation as well as correctness of refinement steps, is demonstrated by discharging a number of verification conditions – proof obligations. Moreover, the Event-B formalism allows the developers themselves to formulate theorems to be proven. Full definitions of all the proof obligations are given in [1].

The Rodin platform [6] provides an automated support for formal modelling and verification in Event-B. In particular, it automatically generates the required proof obligations and attempts to discharge them. The remaining unproven conditions can be dealt with by using the provided interactive provers.

Formalisation of Safety Requirements in Event-B. Formal modelling is especially beneficial for requirements engineering. It helps to spot missing or contradictory requirements and rigorously define system properties and constraints. In a succession of EU projects [2, 4, 7], the most prominent of which is Deploy [2], we have gained significant experience in modelling safety-critical systems from different domains. It allowed us to identify a number of typical solutions for representing requirements in formal models. These solutions can be represented as classes of requirements (for more details, see [8]). Below we give a few examples of the classes of the requirements:

- *Class 1: Global properties* – contain invariant properties to be maintained;
- *Class 2: Local properties* – define effects of certain action in a particular system state;
- *Class 3: Causal order* – define the required order of system events;
- *Class 4: Absence of system deadlock* – require that execution of safety actions should not be prevented by a deadlock.

Table 1. Formalisation of safety requirements

Safety requirement	Model element expressions	Associated verification theorem(s)
<i>SR of Cl. 1</i>	invariants	group of invariance theorems for each event
<i>SR of Cl. 2</i>	event, state predicate	theorem about a specific post-state of an event
<i>SR of Cl. 3</i>	pairs of events, event control flow	group of theorems about enabling relationships between events
<i>SR of Cl. 4</i>	all events	theorem about the deadlock freedom

Table 1 summarises typical representation of the above classes in an Event-B model. Formally, the described relationships can be defined as a function F_M mapping safety requirements (*SRs*) into a set of the related model expressions:

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

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 basic elements of Event-B models such as *axioms*, *variables*, *invariants*, *events*, and *attributes*. Such defined mapping allows us to trace the system safety requirements given in an informal manner into formal specifications in Event-B.

Formal modelling and verification allow the designers to not only achieve a high confidence in system design, but also justify system safety during certification. The increasing reliance on safety cases in the certification process has motivated our work on linking formal modelling in Event-B with safety case construction – the work that we describe next.

3 From Event-B Models to Safety Cases

Safety Cases: Background. 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” [9]. The construction, review and acceptance of safety cases are valuable steps in the safety assurance process of critical systems. Several industrial standards, e.g., ISO 26262 [10] and EN 50126 [11], prescribe production and evaluation of safety cases for system certification.

In general, safety cases can be documented either textually or graphically. Currently, the graphical notation called *Goal Structuring Notation (GSN)* [12, 13] is gaining popularity for presenting safety arguments within safety cases. GSN aims at graphical representation of safety case elements as well as the relationships between them. The building blocks of GSN are shown in Fig. 2. Essentially, such a constructed safety case consists of *goals*, *strategies* and *solutions*. Here *goals* are the requirements, targets or constraints to be met by a system. *Solutions* contain the information extracted from analysis, verification 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 into sub-goals.

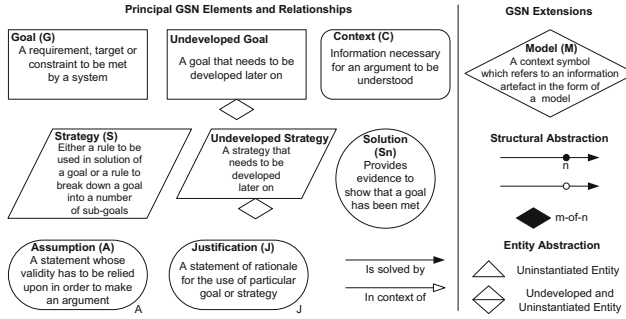


Fig. 2. Elements of GSN

Safety case elements can be in two types of relationships: “*Is solved by*” and “*In context of*”. The former is used between goals, strategies and solutions. The latter links a goal (a strategy) to a context, an assumption or a justification.

GSN has been extended with generic argument patterns [12], supporting structural and entity abstraction. The examples of structural abstraction are *multiplicity* and *optionality*. Multiplicity is a generalised n -ary relationship between the GSN elements, while optionality stands for optional and alternative relationship between them. There are also two extensions for entity abstraction: uninstantiated entity as well as undeveloped and uninstantiated entity. The former one specifies that the entity requires to be instantiated, i.e., to be replaced with a more concrete instance. In Fig. 2, this is depicted as a hollow triangle. The latter one indicates that the entity needs both further development and instantiation. This is displayed as a hollow diamond with a line in the middle.

From Requirements to Safety Cases via Event-B Models. The approach proposed at this paper should create an information continuum that spans requirements engineering, formal modelling and safety case construction as shown in Fig. 3. Problems with defining a safety argument during safety case construction might indicate that some safety requirements are overlooked or a formal specification is not sufficiently constrained. Such a feedback should invoke the corresponding corrective actions (a dashed line in Fig. 3).

The proposed approach encompasses two main activities : (1) representation of safety requirements in Event-B models, and (2) derivation of safety cases from the associated Event-B specifications. The activities are tightly connected with each other. They depend on several factors such as adequacy of representation of the system behaviour by a formal model and availability of modelling and verification artefact to substantiate safety argument.

To facilitate the first activity – representation of safety requirements in Event-B models – we rely on our classification and mapping rules defined above. To simplify the task of linking the formalised system safety requirements with the constructed safety case, we propose a set of classification-based argument patterns. In addition, a special pattern is created to provide the argumentation that the formal model we rely on is by itself correct and well-defined.

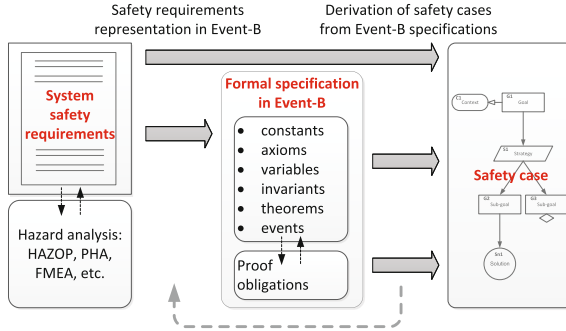


Fig. 3. High-level representation of the overall approach

The patterns have been developed using the described above GSN extensions. Some parts of an argument pattern may remain the same for any instance, while others need to be further instantiated (those are labelled with a hollow triangle). The text highlighted by braces { } should be replaced by a concrete value.

A generic representation of a classification-based argument pattern is given in Fig. 4. Here, a safety requirement *Requirement* of some class *Class* {*X*} is reflected in the goal **GX**. 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.

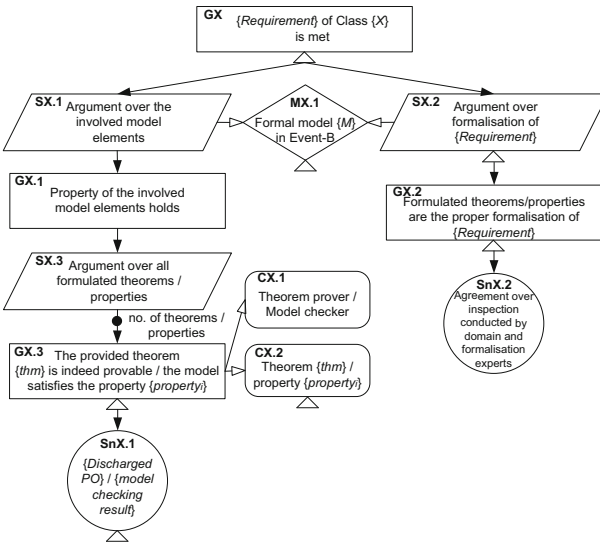


Fig. 4. Generic argument pattern

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.

To bridge a semantic gap in the mapping associating an informally specified safety requirement with the corresponding formal expression in Event-B, we need to argue over a correct formalisation of the requirement (**SX.2** in Fig. 4). We rely on a joint inspection conducted by domain and formalisation experts (**SnX.2**) as the evidence that the associated model elements (via the defined mappings) are proper formalisations of the requirement under consideration.

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. To make this construction generic, we associate each class with the corresponding safety case argument pattern that can be instantiated in different ways. Note that the process of safety requirements elicitation is left outside of consideration in this paper. We assume that the given list of these requirements is completed beforehand by applying safety analysis techniques.

In the next section, we will illustrate such safety case construction by a case study. More details of the proposed arguments patterns can be found in [14].

4 Case Study: A Steam Boiler System

In this section, we demonstrate our approach by a case study – a steam boiler control system [15]. Due to lack of space, we only give a brief overview of system requirements, constructed formal models and fragments of the safety case. The complete description can be found in [16].

Steam Boiler: Requirements and Development Strategy. The steam boiler is a safety-critical control system that produces steam and adjusts the quantity of water in the steam boiler chamber to maintain it between the lower safety boundary $M1$ and upper safety boundary $M2$. The situations when the water level is too low or high are hazardous and must be avoided.

The system consists of the following units: a chamber, a pump, a valve, a sensor to measure the water quantity in the chamber, a sensor to measure the steam quantity out of the chamber, a sensor to measure water input through the pump, and a sensor to measure water output through the valve.

After being powered on, the system enters the **Initialisation** mode. At each control cycle, the system reads sensors and performs failure detection. If no failure detected, the system may enter one of its operational modes **Normal**, **Degraded** or **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

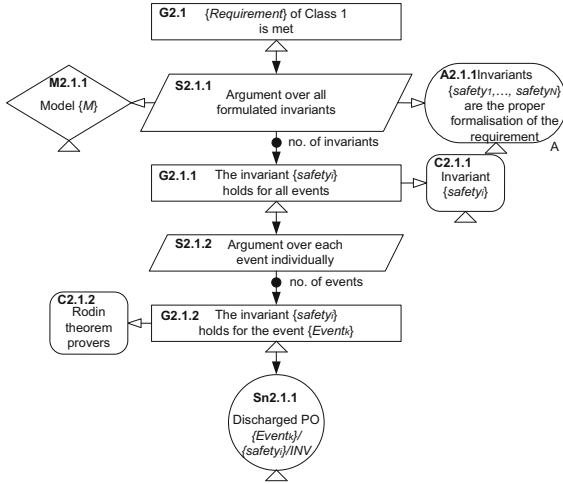


Fig. 5. Argument pattern for safety requirements of Class 1

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**.

The failure of the steam boiler control system is detected if either the water level in the chamber is outside of the safety boundaries or the combination of a water level sensor failure and a failure of any other system unit (the pump or the steam 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. In a similar way, a steam output sensor failure is detected. The pump fails if it does not change its state when required.

Our Event-B development of the steam boiler case study consists of an abstract specification and its four refinements [16]. 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. The second refinement (MACHINE M2) introduces a detailed representation of the system failure conditions. The third refinement (MACHINE M3) models the system physical environment 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.

From an Event-B Model to a Safety Case. The steam boiler control system should adhere to a number of safety requirements. Let us illustrate construction of fragments of a safety case for some given safety requirements.

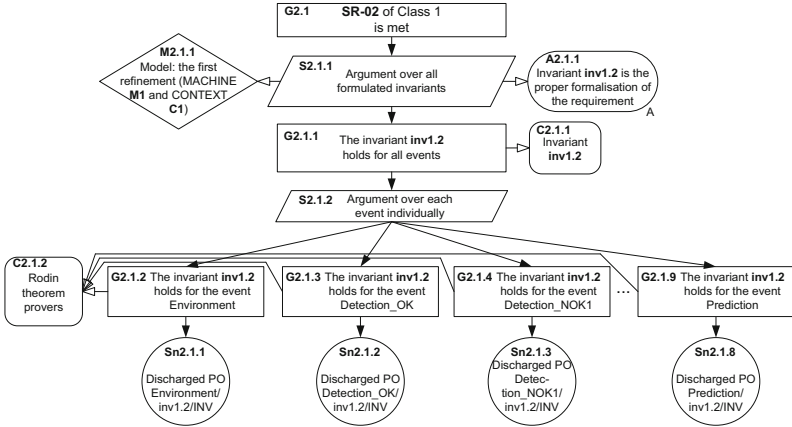


Fig. 6. A fragment of the safety case corresponding to assurance of SR1

The main safety requirement – **SR-02**: *During the system operation the water level shall not exceed the predefined safety boundaries* belongs to requirements Class 1. A natural way to formalise these requirements is by associating them with the corresponding invariant properties in the associated Event-B model. Therefore, the proposed form of the mapping function for *Class 1* is

$$Req_i \mapsto \{safety_inv_{i1}, \dots, safety_inv_{iN}\}$$

for each such requirement Req_i and its associated invariants.

To formally verify the requirement, we have to prove the invariant preservation for all the affected model events.¹ The discharged proof obligations can be used then as the safety case evidence that the requirement holds. This is reflected in the associated safety case argument pattern for *Class 1* (see Fig. 5).

We formalise the requirement **SR-02** as the invariant **inv1.2** at the first refinement step of the Event-B development (MACHINE **M1**):

$$\mathbf{inv1.2}: failure = FALSE \wedge phase \in \{CONT, PRED\} \Rightarrow \\ min_water_level \geq M1 \wedge max_water_level \leq M2,$$

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 (fragment of) mapping function F_M for this case is

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

which is a concrete instance of its general form given above.

Finally, we instantiate the argument pattern for *Class 1* as shown in Fig. 6. To support the claim that **inv1.2** holds for all the affected events, we attach the discharged proof obligations as the evidence.

¹ The affected model events are those that change any variables appearing in the considered invariant(s).

Since the steam boiler system is a failsafe system, whenever an unrecoverable system failure occurs, the system should be stopped. In our model, such a failure is associated with raising the corresponding flag *stop*. The overall condition is defined by the safety requirement **SR-01**: *When a system failure is detected, the steam boiler control system shall be shut down and an alarm shall be activated.*

The requirements belonging to *Class 2* that represents local properties, i.e., the properties that need to be true at particular points of system execution. 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. Hence, the proposed form of the mapping function for *Class 2* is

$$Req_i \mapsto \{(event_{i1}, q_1), \dots, (event_{iN}, q_N)\},$$

where Req_i is a requirement, $event_{ij}$ are the associated events, and q_j are the desired post-conditions for those events. For each pair of an event and a predicate, it is rather straightforward in Event-B to generate the corresponding theorem, which becomes an additional proof obligation. In its turn, the proved theorem becomes the evidence for the constructed safety case (see [14] for details).

The corresponding instance of the mapping function F_M for **SR-01** is

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

Thus, we formalise the requirement by associating it with the desired post-condition $stop = TRUE$ of the event *EmergencyStop*. To verify it, we construct and prove the following theorem:

Thm1.1: $\forall stop' \cdot stop' \in BOOL \wedge (\exists phase, stop \cdot phase \in PHASE \wedge stop \in BOOL \wedge phase = CONT \wedge stop = FALSE \wedge stop' = TRUE) \Rightarrow stop' = TRUE,$

The theorem is trivially true (i.e., automatically discharged by the Rodin provers).

The instantiated fragment of the safety case is presented in Fig. 7. The proof obligation *thm1.1/THM* serves as the evidence that this requirement holds.

The steam boiler is a typical control system that cyclically executes a predefined sequence of actions: reading sensors, detecting failures, executing control actions or error recovery, and predicting the next system state. We can formulate that sequence of events as a corresponding requirement belonging to Class 3.

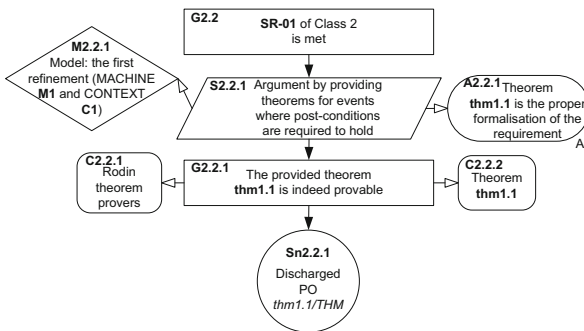


Fig. 7. A fragment of the safety case corresponding to assurance of **SR-01**

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 rely on flow Event-B extension proposed by Iliasov [17] to verify that the required order of events is enforced. The Flow plug-in for the Rodin platform allows us to express all these relationships in a diagrammatic way, generating the corresponding theorems automatically.

In this paper, we omit further illustration of safety case construction for different classes of requirements. However, let us note that to ensure that the constructed safety arguments are valid, we also have to define a special argument pattern that demonstrates well-definedness of the formal models themselves as described in the accompanying technical report[14].

The use of the Rodin platform and accompanying plug-ins has facilitated derivation of formal 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, which in turn allowed us to explicitly refer to their proofs in the resulting safety case.

In this section, we demonstrated how models and proofs in Event-B can be used in construction of a safety case. Though the resultant development appears as a linearly constructed refinement chain, in practice it is a result of several iterations of trials and errors. To maintain traceability between the requirements, models and safety case fragments, we need to create an automated integrated engineering environment – the problem that we discuss next.

5 Integrated Automated Tool Support

Development of safety-critical systems is a joint effort of engineers from diverse domains, including electro-mechanical, hardware, software, safety etc. Each of the engineering teams applies domain-specific analysis and design methods and correspondingly uses the dedicated engineering tools. Though the engineering environment is inherently heterogenous, productivity of the development process and safety per se depend on how seamlessly the information about design decisions and constraints propagates across domains.

Let us consider the interactions between requirements engineering and formal modelling. Formal modelling typically results in identifying problems in given requirements (e.g., missing or contradictory ones) as well as deriving the constraints for the requirements to be satisfied. Hence, requirements definition and model creation co-evolve, and changes in one domain should invoke changes in the other. In its turn, these changes should be reflected in a safety case. Since the safety case construction should proceed alongside the development, the inability to produce safety argument may trigger the whole chain of requirement re-definition, formal model change and safety case re-construction.

Therefore, to address establishing an information continuum from requirements to safety cases, we should create a platform for non-obtrusive integration of tools in an integrated tool chain. The work presented in the paper is a part

of a more general ongoing effort of tool integration as well as formalisation and mechanisation of rules that turn a collection of disparate tools into a tool chain.

We believe that dynamic tool integration enabling real-time sharing of data is the way to build tool chains of tomorrow. There is enough technological context to make such integration relatively cheap and painless, even for pre-existing tools not meant to operate in a dynamic setting. The enabling technologies we consider crucial are the structured data representation with stable identifiers and the actor paradigm. The former gives a common syntactic base for all the tools without enforcing unreasonable restrictions. An example of such a technology is OSLC [5] described below. The actor model provides a simple and flexible integration framework detached from the logic and code of integrated tools.

Before describing our solution in more detail, let us give a brief overview of a new industry-driven interoperability standard – OSLC – that we rely on.

OSLC: Background. *Open Services for Lifecycle Collaboration* (OSLC) [5] is an open community, the goal of which is to create specifications for integrating tools, their data and workflows to support lifecycle processes. OSLC address integration scenarios for individual topics such as change management, test management, requirements management and configuration management.

In simple terms, OSLC specifications focus on how the external resources of a particular tool can be accessed, browsed over, and specific change requests can be made. OSLC is not trying to standardise the behaviour of any tool. Instead, OSLC specifies a minimum amount of protocol and a small number of resource types to allow two different tools to work together relatively seamlessly.

To ensure coherence and integration across these domains, each workgroup builds on the concepts and rules defined in the OSLC Core specification [5]. OSLC Core consists mostly of standard rules and patterns for using HTTP and RDF (Resource Description Framework) that all the domains must adopt.

In OSLC, each artefact in the lifecycle – a requirement, test case, source file etc. – is an HTTP resource that is manipulated using the standard HTTP methods (GET, POST, etc.). Each resource has its RDF representation, which allows statements about resources in the form of subject/predicate/object expressions, i.e., as linked data. Other formats, like JSON or HTML, are also supported.

OSLC Requirements Management (RM) specification is built on the top of OSLC Core. It supports key REST APIs for software Requirements Management systems. The additionally specified properties describe the requirements-related resources and the relationships between them.

There are several different approaches to implementing an OSLC provider for software. For this work, we rely on so called the *Adapter* approach. It proposes to create a new web application that acts as an OSLC adapter, runs along-side of the target application, provides OSLC support and ”under the hood” makes calls to the application web APIs to create, retrieve, and update external resources.

OSLC Tool Bus. To enable tool interconnection, we require that each tool has an OSLC adapter. The adapter offers a web service style API for traversing as well as changing tool data in real time. Generally, all well designed tools following

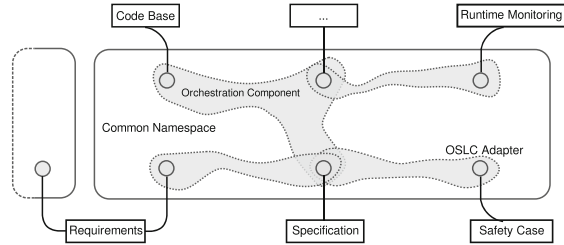


Fig. 8. OSLC-based tool bus.

the model-view-controller design pattern (e.g., based on Eclipse GMF) can be easily extended with an OSLC adapter.

An OSLC adapter is purely passive: it offers access to structured data that may be rendered in differing formats. It does not by itself link two tools together. The linking, or as we call it, tool orchestration, requires an additional piece of logic to define how and when the tools need to exchange information as illustrated in Fig. 8. An orchestration solution must (i) ensure that common names refer to same concepts, and (ii) manage the information flow between the tools.

To address this, we propose to use the agent paradigm [18], where each tool comes with one or more agents necessary for tool coordination. A collection of agents working together coordinating several tools is called an orchestration component. A tool may be a part of several interactions (see Fig. 8). Thus, e.g., a specification may be interlinked with code base, a safety case and requirements.

The role of an agent is to represent the interests of a respective tool by notifying other agents of any relevant new data and also acting on any such updates from other agents. The underlying communication framework implements a *federated tuple space* [19] - a distributed implementation of a shared blackboard with Linda coordination primitives [20]. To simplify agent implementation, we also offer the publisher/subscriber and mailbox communication styles realised on top of the tuple space API. Asynchronous message passing is a good fit for real-time coordination of a distributed tool chain, while federated tuple space is especially well-suited to loosely coupled parties that at times may be disconnected from some or all of peers. It is possible to construct generic agents able to handle simple tasks like synchronising certain data of some two tools or constantly broadcasting changes to a certain part of tool data. This enables, in principle, a compositional approach to agent design where complex orchestration logic is built, brick by brick, from the predefined agents.

A logical extension of this idea is fusion of an agent specification and the tool OSLC interface. Recall that OSLC is primarily a gateway to the tool data. It does not span across several tools. We are working on a way to extend an OSLC specification with the coordination meta-data defining the logic of orchestration components via static documents serialised in, e.g., XML or RDF form.

Prototype Tool Chain. In our prototype implementation, we aim at building an environment that integrates requirements engineering, formal modelling and

verification, and safety case development. We strive to retain flexibility and notation that is native for each domain. For instance, requirements are defined in natural language. To maintain the link between the dynamically changing requirements and the associated formal models, we have created a prototype Requirements-Rodin adapter [21]. Formal modelling is done using the Event-B language, while safety cases are generated in a goal-structuring notation.

Our requirements tool uses the generic principle of organising requirements into a tree with further optional cross-links between them, and their classifications (by taxonomy, component, developer, etc.). The tool provides a simple form-based user interface. It embeds a web-service that provides OSLC-compliant RDF descriptions of requirements. Every requirement may be referred to by the project name and requirement id.

The second part of the prototype achieves a similar goal for the Rodin Platform. We have developed a Rodin plug-in that exposes the Event-B model database and proofs as externally referable OSLC resources. Once again, each model element (variable, invariant, refinement) has a unique global identifiers that can be used to cross-link with other OSLC and RDF resources.

The third part of the environment facilitates generation of safety case. It maps relevant elements of requirements and models into the corresponding parts of safety case, i.e., allows to reuse the results of formal modelling and verification to construct a safety argument.

6 Related Work and Conclusions

Related Work. The relationships between formal methods and safety cases have been studied along two main directions: to prove soundness of safety argument and gather evidences from formal modelling to substantiate safety argument. The most prominent work on the former is by Rushby [22], in which he formalises the top-level safety argument to support automated soundness checking. The obtained theorem can be then verified by a theorem prover or a model checker.

Our work is closer to the second research direction. Hawkins et al. [23] propose an approach that relies on static analysis of program code to demonstrate that the software does not contain hazardous errors. In [24], the authors automate generation of heterogeneous safety cases, starting from a (manually developed) top-level system safety case, while lower-level fragments are automatically generated from formal verification of safety requirements. In [25], to ensure that a model derived during model-driven development of a safety critical system satisfies all the required properties, the authors use the obtained model checking results. Our approach follows a similar idea. The main difference is in the reliance on the introduced requirements classification to construct both associated formal model and resulting safety argument. Moreover, the automatic tool support created for the proposed approach significantly improves its usability.

Conclusions. In this paper, we have presented an approach and a prototype tool implementation for integrating formal modelling in Event-B into the process

of development and assurance of safety-critical systems. We aimed at providing support for linking requirements and formal models as well as efficient reuse of formal modelling and verification artefacts in safety case construction. The prototype tool implementation provides a platform for dynamic information sharing between safety engineers and verification team. It relies on the idea of linked data promoted by the OSLC standard, which is now rapidly spreading in industry.

To validate the approach, we have undertaken formal development and safety case construction of the steam boiler system. In our work, to test the approach scalability and usability, we have deliberately aimed at representing a large set of complex requirements of the system and then constructing the safety case.

We believe that the proposed approach is beneficial for the development of complex safety-critical systems because it allows the engineers to establish an information continuum between different involved domains. As a future work, we are planning to continue our work on integration by focusing on the integration with techniques for safety analysis as well as different verification tools.

Acknowledgement. The presented work is partially supported by the TEKES project Cyber Trust.

References

1. Abrial, J.R.: Modeling in Event B. Cambridge University Press (2010)
2. (EU-project DEPLOY). <http://www.deploy-project.eu/>
3. Romanovsky, A., Thomas, M. (eds.): Industrial Deployment of System Engineering Methods. Springer, Heidelberg (2013)
4. (EU-project RODIN). <http://rodin.cs.ncl.ac.uk/>
5. OSLC: (Open Services for Lifecycle Collaboration.). <http://open-services.net/>
6. RODIN: Event-B Platform (2009). <http://www.event-b.org/>
7. (EU-project ADVANCE). <http://www.advance-ict.eu>
8. Prokhorova, Y., Laibinis, L., Troubitsyna, E.: Towards rigorous construction of safety cases. Technical Report 1110 (2014)
9. Bishop, P., Bloomfield, R.: A methodology for safety case development. In: Safety-Critical Systems Symposium, Birmingham, UK. Springer (1998)
10. International Organization for Standardization: ISO 26262 Road Vehicles Functional Safety (2011)
11. European Committee for Electrotechnical Standardization: EN 50126 Railway applications - The Specification and Demonstration of Reliability. Availability, Maintainability and Safety (RAMS) (2011)
12. Kelly, T., McDermid, J.: Safety case construction and reuse using patterns. In: Daniel, P. (ed.) Proceedings of the 16th International Conference on Computer Safety, Reliability and Security (SAFECOMP 1997), pp. 55–69. Springer (1997)
13. Goal Structuring Notation Working Group: Goal Structuring Notation Standard (2011). <http://www.goalstructuringnotation.info/>
14. Prokhorova, Y., Laibinis, L., Troubitsyna, E.: Facilitating construction of safety cases from formal models in Event-B. Information and Software Technology **60**, 51–76 (2015)
15. Abrial, J.R.: Steam-Boiler control specification problem. In: Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control, London, UK, pp. 500–509. Springer (1996)

16. Prokhorova, Y., Troubitsyna, E., Laibinis, L.: A Case Study in Refinement-Based Modelling of a Resilient Control System. TUCS Technical Report 1086 (2013)
17. Iliasov, A.: Use case scenarios as verification conditions: event-B/Flow approach. In: Troubitsyna, E.A. (ed.) SERENE 2011. LNCS, vol. 6968, pp. 9–23. Springer, Heidelberg (2011)
18. Wooldridge, M.: An Introduction to MultiAgent Systems. Wiley Publishing (2009)
19. Iliasov, A., Romanovsky, A.: Structured coordination spaces for fault tolerant mobile agents. In: Cheraghchi, H.S., Lindskov Knudsen, J., Romanovsky, A., Babu, C.S. (eds.) Advanced Topics in Exception Handling Techniques. LNCS, vol. 4119, pp. 181–199. Springer, Heidelberg (2006)
20. Gelernter, D.: Generative communication in linda. *ACM Transactions on Programming Languages and Systems* **7**(1), 80–112 (1985)
21. Rodin OSLC Adapter: (Using Instructions). <http://iliasov.org/oslc/>
22. Rushby, J.: Formalism in safety cases. In: Dale, C., Anderson, T. (eds.) Making Systems Safer: Proceedings of the Eighteenth Safety-Critical Systems Symposium, pp. 3–17. Springer, Bristol (2010)
23. Hawkins, R., Habli, I., Kelly, T., McDermid, J.: Assurance cases and prescriptive software safety certification: a comparative study. *Safety Science* **59**, 55–71 (2013)
24. Denney, E., Pai, G., Pohl, J.: Automating the Generation of Heterogeneous Aviation Safety Cases. NASA Contractor Report NASA/CR-2011-215983 (2011)
25. Jee, E., Lee, I., Sokolsky, O.: Assurance cases in model-driven development of the pacemaker software. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 343–356. Springer, Heidelberg (2010)