Formalisation-Driven Development of Safety-Critical Systems

Alexei Iliasov, Alexander Romanovsky
Newcastle University
Newcastle Upon Tyne, UK
{alexei.iliasov, alexander.romanovsky}@ncl.ac.uk

Elena Troubitsyna, Linas Laibinis
Åbo Akademi University
Turku, Finland
{elena.troubitsyna, linas.laibinis}@abo.fi

Abstract—The use of formal modelling and verification is recommended by several standards in the development of highly critical systems. However, the standards do not prescribe a process that enables a seamless integration of formalisation activities into the development process. In this paper, we propose a model and an automated tool support for an iterative formalisation-driven development of safety-critical systems in the Event-B framework. Event-B supports correct-by-construction development and provides the designers with a continuous feedback on the correctness of models and corresponding system requirements, including safety. To automate the proposed formalisation-driven development, we present a prototype of an automated tool support relying on the novel OSLC technology. It allows us to seamlessly integrate derivation of system requirements with formal modelling and proof-based verification.

I. INTRODUCTION

Complexity of modern software-intensive systems and, in particular, safety-critical systems is continuously growing. It often makes testing of such systems to the desired degree of reliability infeasible [1]. Consequently, developers are increasingly relying on formal modelling techniques to verify system correctness [2]. Formal modelling and verification is also recommended by several standards for highly critical systems. Growing maturity of automated tools makes formal verification, including proof-based verification, more accessible and attractive for industrial engineers [2]. Indeed, proofs not only allow the developers to verify model correctness but also spot deficiencies and inconsistencies in the given requirements [3].

In this paper, we propose an iterative formalisation-driven process of safety-critical system development. We also develop a prototype tooling platform that integrates the process of defining requirements for safety-critical systems with formal modelling in Event-B.

Event-B is a top-down state-based framework to formal development [4]. Modelling in Event-B starts from creating an abstract specification that captures the high-level system functionality and properties. In a number of correctness-preserving model transformations – refinements – the developers gradually elaborate on the specification to address lower-level system requirements. Correctness of such models and their refinements in Event-B is verified by proofs. An integrated extendable framework – Rodin platform [5] – provides an automated support for formal development in Event-B.

The approach that we propose in this paper enables formalisation-driven derivation of requirements of safety-critical systems. We start from an informal high-level description of the system behaviour. The incremental definition of requirements proceeds concurrently with model construction, while the conducted proofs verify the corresponding subset of the defined requirements. As a result, spotted incompleteness and inconsistencies immediately lead to correcting the requirements description as well as defining the missing requirements, if needed. The process iteratively progresses until the desired level of detail is reached.

Our prototype of an integrated engineering environment relies on a new industry-driven interoperability standard – OSLC [6]. It allows us to seamlessly integrate derivation of system requirements with formal modelling and verification in Event-B. We demonstrate the proposed approach by an example – an airlock control system. We believe that the proposed formalisation-driven development approach facilitates design of safety-critical systems by providing early feedback on system requirements and establishing a common information space in the engineering of safety-critical systems.

II. BACKGROUND: EVENT-B

Event-B is a state-based formalism that promotes the correct-by-construction approach to system development and verification by theorem proving. In Event-B, a system model is specified using the notion of an abstract state machine [4], [7]. 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 of Event-B models is given in Figure 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:

\[ e \equiv \text{any a where } G_e \text{ then } R_e \text{ end}, \]

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 body of an event is defined by a multiple (possibly nondeterministic) assignment over the system variables. In Event-B, an assignment represents a 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.

The Rodin platform [5] 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.

### III. MODEL/REQUIREMENTS CO-ENGINEERING

In Event-B, complex system models are constructed in an incremental manner. The system behaviour and properties are gradually introduced and elaborated on in the refinement process. The associated proofs provide a feedback that often helps to discover missing, contradictory or ambiguous requirements.

In this paper, we introduce and automate an iterative workflow for safety-critical system development. Each iteration aims at defining and formalising a certain subset of system requirements and incorporating a feedback provided by the formalisation into the requirements definition.

Each iteration of our approach comprises one or more of the following steps:

- defining a subset of the requirements by deriving them from a general system description or by elaborating on the higher-level requirements;
- formalising these requirements as an Event-B model or a refinement of a more abstract model;
- attempting to verify by proofs logical consistency of the model;
- modifying the model(s), if necessary, by, e.g., adding additional invariants or constraints, to make the model(s) provably correct;
- reflecting model modifications in the requirements definition by adding missing requirements or correcting the defined ones;
- elaborating (refining) the requirements by adding new concrete requirements;
- creating the corresponding refined models and proving their correctness with respect to the abstract ones;
- translating the required model modifications into adding missing requirements or modifying existing ones, etc.

The requirements document is developed gradually, by starting with general system requirements and then elaborating on them by adding more concrete and detailed requirements pertaining to certain components or aspects of the system behaviour. A similar consistent detailisation and unfolding of the system is captured by the concept of model refinement.

In our approach, we co-align requirements definition and formal modelling processes in such a way that more concrete requirements are mapped onto refined, i.e., more detailed formal models, while the hierarchical relationships between requirements become specific verification conditions between the corresponding formal models. As a result, the hierarchical requirement development process is aligned with the formal refinement structure. In general, refinement provides a mental framework for a systematic mapping of requirements and enforces strong verification obligations facilitating the proof-driven development process.

The Fig.2 graphically portrays the proposed co-engineering approach to requirements definition and formal modelling. The arrows represent interactions between the requirements elicitation and model development/verification. Moreover, \( \square \) depicts here requirement elicitation or specification development; \( \blacksquare \) is the specification proof effort following a specification change; \( \blacksquare \) represents the integration activity where model changes are reconciled with requirements (typically resulting in new added requirements); finally, \( \blacktriangle \) depicts parallel refinement of requirements and the associated model(s), which may lead to introducing new refined model elements as well as adding the new detailed requirements descriptions.

In the next section, we will demonstrate our approach to formalisation-driven development in Event-B by an example – a specification of an airlock control system.
IV. ILLUSTRATING EXAMPLE

The main function of the airlock is to separate two areas with different air pressures and allow the users to pass safely between the areas (see Fig. 3).

For clarity, let us call the two conjoining areas as external (the left area) and internal (the right one). Let us also assume that the pressure outside is lower than inside. In order to allow a user to pass from inside through the airlock into the external area, the system needs to perform the following steps:

- equalise the chamber pressure to that of the internal environment,
- open the second door to allow the user into the chamber,
- close the second door,
- equalise the pressure in the airlock to that of the external environment,
- open the first door to let the user out.

Moreover, the opposite (dual) scenario needs to be performed to allow the user pass from outside through the airlock into the external area.

The system is equipped with a number of actuators - door motors, a pressure pump, as well as sensors - pressure sensors, door position sensors and buttons. Our goal is to develop control software that would allow a human operator to safely pass through the airlock. In the scope of this demonstration we focus exclusively on safety and liveness properties of the developed system, leaving aside issues of its usability, operation speed, reliability and maintainability.

We can describe these (given above) assumptions about the environment of the system as the following high-level requirements:

ENV1. The airlock system separates two different environments. The pressure of the external environment is lower than that of the internal one.

ENV2. In order to maintain different pressures, the two environments must be physically separated.

The primary function of the system is to allow an operator to travel between internal and external environments.

FUN1. When in operation, the airlock system must be able to let users pass safely between the two environments via the airlock.

In the abstract machine $m0$ we represent the movement of a human operator between the external and internal environments. This succinctly, in an abstract form, models the requirements ENV1, ENV2, and FUN1.

The location of a user is represented by model variable $user$. The model invariant $inv1$ defines it as an element an enumerated set USER_POS0:

$$inv1: user \in USER_POS0$$

where USER_POS0 is defined in the accompanying model context $c0$ as the corresponding axiom $axm1$

$$axm1: USER_POS0 = \{OUT, IN\}.$$  

We satisfy FUN1 by defining events (state transitions) modelling the movement of a user between the environments:

$$go\_in \triangleq \text{when } user = \text{OUT then } user := \text{IN end}$$

$$go\_out \triangleq \text{when } user = \text{IN then } user := \text{OUT end}$$

The model consistency proof obligations [4] require that every model variable is initialised, possibly non-deterministically, establishing the given invariant. To satisfy this, we must make a decision where the human operator may be initially located. This allows us to spot a missing requirement regarding the initial position of the user. We decide that initially the user can be in either environment and hence the following initialisation statement is introduced.

\[ INITIALISATION \triangleq \begin{align*}
\text{begin } 
\end{align*} \text{user } \in USER\_POS0 \text{ end} \]

The initialisation action $user \in USER\_POS0$ is then translated into a new requirement FUN2.

FUN2. A human operator may initiate airlock use from inside or outside environments.

The initial specification has allowed us to identify one missing requirements FUN1 and establish the following mapping between the requirements and Event-B model:

\[ ENV1 \rightarrow c0/axm1 \]

\[ ENV2 \rightarrow c0/axm1, m0/inv1 \]

\[ FUN1 \rightarrow m0/go\_in, m0/go\_out \]

\[ FUN2 \leftarrow m0/INITIALISATION \]

where the notation name1/name2 stands for the name of the machine or the context and name of the element (axiom, invariant, event etc) correspondingly.

A. First refinement

As the next step, we capture the high-level protocol of airlock operation, namely, the three principal airlock modes: awaiting a command, operating in the left-to-right mode, and operating in the right-to-left mode. They correspond to the following high-level requirements:

FUN3. Environments are separated by an airlock.

FUN4. Airlock has operation modes: ALWAIT, ALIN, AOUT.

Our refined specification, defined as a machine $m1$, aims at modelling transitions between the modes. The airlock state is modelled by a new variable, $alck$, defining the current airlock mode according to FUN4:

$$inv1: alck \in AIRLOCK\_STATE$$
The transitions between the modes are specified by the following events:

\[
\text{airlock\_operate} \triangleq \\
\text{when } \text{airlock} = \text{ALWAIT then } \text{airlock} \in \{\text{ALIN}, \text{ALOUT}\} \text{ end}
\]

\[
\text{airlock\_done} \triangleq \\
\text{when } \text{airlock} \neq \text{ALWAIT then } \text{airlock} := \text{ALWAIT} \text{ end}
\]

From the idle mode ALWAIT the airlock can make a transition to the modes ALIN or ALOUT, as modelled by the event airlock\_operate. Correspondingly, the event airlock\_done models the transitions to the idle mode ALWAIT.

\[\text{INITIALISATION} \triangleq \text{begin act2: airlock := ALWAIT end}\]

In the refinement step, we also strengthen the guards of the events go\_in and go\_out introduced in the abstract specification to define the requirements for safe use of the airlock: the user’s movement from one environment to another is only allowed when the airlock is in the corresponding mode:

\[
\text{go\_in} \triangleq \text{when } \ldots \land \text{grd2: airlock = ALIN then } \ldots \text{ end}
\]

The two new guards are translated to new symmetric requirements.

\[\text{FUN8(FUN1). A user may travel to IN only when airlock is in the ALIN mode.}\]

\[\text{FUN9(FUN1). A user may travel to OUT only when airlock is in the ALOUT mode.}\]

Here we have a new situation where requirements detailing proceeds alongside with formal model requirement. In this case, the refinement was induced by the specification part although in general it can be initiated from either side. To emphasise that detailing and refinement are tightly interlinked, on the diagram in Fig. 4 we show the corresponding step as a joint action (the chessboard pattern).

During this stage of co-development we have extended the mapping relation with the following links between requirements and the associated model elements:

\[\text{FUN3 } \rightarrow \text{m1/act1}\]

\[\text{FUN4 } \rightarrow \text{m1/inv1}\]

\[\text{FUN5 } \rightarrow \text{m1/INITIALISATION/act2}\]

\[\text{FUN6 } \leftarrow \text{m1/airlock\_operate}\]

\[\text{FUN7 } \leftarrow \text{m1/airlock\_done}\]

\[\text{FUN8 } \leftrightarrow \text{m1/go\_in/grd2}\]

\[\text{FUN9 } \leftrightarrow \text{m1/go\_out/grd2}\]

\[\text{B. Second refinement}\]

A natural way to continue from this point is to refine the abstract airlock into a more concrete (but still idealised) concept of a pair of doors operated in accord. Since formal refinement imposes fairly strict formal constraints between models, this could lead to synthesising a number of new requirements in the process.

First, we state that the airlock is made of two doors.

\[\text{ENV3. The system has two doors and a chamber. Each door when closed separates the chamber from the appropriate environment.}\]

We also add obvious requirements that the doors may be operated.

\[\text{FUN10. An open door may be closed; a closed door may be opened.}\]

At the specification side, we once again make a new refinement step, resulting in the refined machine \(m2\). This step is necessary since we are going to remove the abstract notion of airlock and replace it with a pair of doors. We start by representing the doors as following model variables

\[\text{inv1: door1 } \in \text{DOOR}\]

\[\text{inv2: door2 } \in \text{DOOR}\]

where DOOR is a constant set (defined in the model context \(c2\)) made of two literals OPEN and CLOSED.

The requirement FUN10 is mapped into four model events. For instance, the opening of \(door1\) is specified as follows:

\[
\text{door1\_open} \triangleq \\
\text{when } \text{door1 } = \text{CLOSED then act1: door1 } := \text{OPEN end}
\]

The events \(door2\_open, door1\_close\) and \(door2\_close\) are defined in a similar manner.

Since the airlock abstraction disappears, so does the variable \(\text{airlock}\). Consequently, the events airlock\_operate and airlock\_done as well as the guard mentioning \(\text{airlock}\) may no longer be present in the model. Instead, we must show formally that the abstract airlock concept is now refined by the two doors model.

In particular, we have to show that new door events are refinements of their abstract counterparts. The event \(door1\_open\) refines the abstract event airlock\_operate for the case of \(\text{airlock} = \text{ALOUT}\), while \(door2\_open\) covers the case of \(\text{airlock} = \text{ALIN}\). Both door closing events refine airlock\_done.

The event refinement leads to a number of action simulation and guard strengthening proof obligations (for details, see Section II), which in this case cannot be proven straight away.

A failure to prove these proof obligations automatically suggests adding the following two new safety invariants:

\[\text{inv5: } \neg (\text{door1 } = \text{OPEN } \land \text{door2 } = \text{OPEN})\]

\[\text{inv6: } \text{door1 } = \text{CLOSED } \land \text{door2 } = \text{CLOSED } \leftrightarrow \text{airlock } = \text{ALWAIT}\]
These are sufficient to prove event refinement and, as a result, they are reflected back in the requirements documents as new requirements

FUN11. door1 and door2 may not be open at the same time.

FUN12. When both doors are closed, the airlock is in the waiting mode.

Note that we do not need to completely precise in the natural language descriptions of requirements as the supporting specification may be consulted to clarify the statement meaning.

The new invariants also require changes to some of the existing events. In particular, a failed proof obligation of invariant preservation of inv5: by door1 open suggests the following new guard:

\[
\begin{align*}
\text{door1\_open} & \triangleq \\
\text{when } \cdots \land \text{grd2: door2 = CLOSED} \text{ then } \cdots \text{ end}
\end{align*}
\]

with a symmetric case for door2 open.

Finally, there is still the matter of event referring to alck in event guards of go in and go out. These are now refined into door2 = OPEN for go in and door1 = OPEN for go out with the additional invariant conditions (necessary to carry out the proof) relating the airlock operation mode with the current user position and door states:

inv3: user = OUT \land door2 = OPEN \Rightarrow alck = ALIN
inv4: user = IN \land door1 = OPEN \Rightarrow alck = AOUT

All these guard changes yield three new requirements:

FUN13(FUN11). A door may be opened only when both doors are currently closed.

FUN14(FUN8). A user may move inside only when door2 is open.

FUN15(FUN9). A user may move outside only when door1 is open.

These changes conclude the current refinement step. As a result, the mapping relation is now extended with the following links.

ENV3 \rightarrow m2/inv1, m2/inv2

FUN10 \rightarrow m2/door1_open,

FUN11 \leftarrow m2/inv5

FUN12 \leftarrow m2/inv6

FUN13 \leftarrow m2/door1_open/grd2, m0/door2_open/grd2

FUN14 \leftarrow m2/go_in/grd2

FUN15 \leftarrow m2/go_out/grd2

FUN8 \leftarrow m2/inv3

FUN9 \leftarrow m2/inv6

Note that the requirements FUN8 - FUN9 are not added at this stage but rather new mapping links are inserted to reflect the fact that refinement-induced invariants m2/inv3 and m2/inv4 now support the previously introduced requirements. A summary of model/requirements co-engineering steps is given in the diagram in Fig. 4.

C. Third refinement

Although the airlock is made of two doors, the user viewpoint of airlock operation is still abstract: to travel through the airlock one needs to open a suitable door and then, in a single instance, move in or out. The models and requirements constructed so far abstract away pressure equalisation as well as operation of the second door. In the third refinement step we refine airlock operation with the notions of a middle chamber and explicit operation of both doors.

inv2: userpos \in \{U_IN, U_MID_OUT\} \leftrightarrow user = IN

Two liveness requirements are added:

LIV1. Starting in an external environment, a user always succeeds in travelling to the internal environment.

and a symmetric requirement LIV2 for moving in the opposite direction. These requirement are mapped into the corresponding LTL/CTL formulas to be verified by the associated Event-B model checker – Pro-B:

\[
G(\{\text{userpos = U_IN}\} \Rightarrow F\{\text{userpos = U_OUT}\})
\]

\[
G(\{\text{userpos = U_OUT}\} \Rightarrow F\{\text{userpos = U_IN}\})
\]

The model checker verification yields several counter-examples, which consequently leads to several model corrections. In turn, new concrete requirements are added to the requirements document. For the lack of space, we omit here the detailed descriptions of these requirements and the appended mapping between them and the related model elements.

V. OSLC-BASED AUTOMATION

Open Services for Lifecycle Collaboration (OSLC) [6] is an open community, the main goal of which is to create specifications for integrating tools, their data and workflows in support of lifecycle processes. OSLC is organised into workgroups that address integration scenarios for individual topics such as change management, test management, requirements management and configuration management. Such topics are called OSLC domains. Each workgroup explores integration scenarios for a given domain and specifies a common vocabulary for the lifecycle artefacts needed to support the scenarios.

In very 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 or capability 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 [8]. OSLC Core consists mostly of standard rules and patterns for using HTTP and RDF (Resource Description Framework) that all the domains must adopt in their specifications. It also defines a small number of resource types that help tools to integrate their activities.

In OSLC, each artefact in the lifecycle – a requirement, test case, source file etc. – is an HTTP resource that is manipulated using the standard methods of the HTTP specification (GET, PUT, POST, DELETE). Each resource has its RDF representation, which allows statements about resources (in particular web resources) in the form of subject/predicate/object expressions, i.e., as linked data. OSLC also supports representations in other formats, like JSON or HTML. Several tools exposing
OSLC-compliant interfaces may be joined, with the help of a notification manager such as a publisher/subscriber server, to form a virtual document to which each individual tools provides a partial interface (Fig. 5).

a) Requirements in OSLC: OSLC Requirements Management (RM) [9] specification is built on the top of the OSLC Core specification. It supports key REST APIs for software Requirements Management systems. The additionally specified properties of OSLC-RM describe the requirements-related resources and the relationships between them.

The meaning of Requirement resource properties are defined in a separate table, together with their multiplicity constraints. Requirement resource properties are not limited to the ones defined in this specification, as Service Providers may provide additional properties.

Using the pre-defined properties in OSLC-RM we can structure the requirements exposed by a requirements management tool, as well as link them with one or several model elements exposed by the associated verification/validation (Rodin). The overall protocol of tool integration is specified by the OSLC Core specification.

b) Implementation and tool support: 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, update and delete external resources.

Eclipse Lyo is an SDK to help the Eclipse community adopt OSLC specifications and build OSLC-compliant tools. In the next section we will discuss our small prototype implementation (using Eclipse Lyo) of OSLC-based integration between a custom-built requirements management tool and the Rodin platform, supporting our co-engineering approach.

VI. TOOLING PLATFORM

The success of the proposed methodology critically depends on the way the dynamics of a development process is affected. Requirement elicitation relies on tight collaboration between domain experts, stakeholders and developers. A requirements document itself serves a concrete medium for communication among these. Putting a formal specification in the midst of this process is likely to negatively affect this communication as most engineers are unused to reading mathematical notation. Hence, from the outset, we were looking for the ways incorporate formal reasoning without disrupting the existing practice of requirements engineering. This means, for instance, that the current tool chain must be preserved and only new side-branches may appear.

The first challenge is to gain access to requirements. For this we rely on a growing trend for interoperable tools. In this work we used the OSLC framework which seems to be rapidly gaining momentum and is backed by a number of large software engineering companies. The role of OSLC is to expose requirements (and, symmetrically, models) in a way that enables other tools to traverse, pull and link, via stable global resource identifiers, to individual requirements or their sub-elements. In its essence, an OSLC-adapted toolset exposes its relevant data as a hierarchical catalogue of objects. The lifetime of a catalogue and its individual elements depends on a tool and, especially, on the kind of data being exposed. In an extreme case, a resource describing a random number generator would change with every read access. More typically, in the context of requirements and software, a resource undergoes periods of rapid changes and then stays stable until the end of the development cycle.

As a prototype experiment, we have developed our own requirements tool (Fig. 7). It uses the generic principle of requirements organised into a tree with further optional cross-links between requirements, and their classifications (by taxonomy, component, developer, etc.). The tool provides a simple form-based UI and, we believe, is a reasonable approximation of some of the more popular industrial tools. The key aspects is that it embeds a web-service that serves OSLC-compliant RDF descriptions of requirements. Every requirement may be referred to by the project name and requirement id:

```
host: port/⟨project-name⟩/⟨requirement-name⟩
```

The link is "live" for as long as a requirement is present, otherwise a report is generated, detailing whether the requirements existed at all and, if it did, when it was removed. The

---

Fig. 5. OSLC enables tight on-line integration of tools resulting in a new kind of artefact represented by cross-linked data of the integrated tools.

Fig. 6. An RDF document served by the Rodin OSLC adapter. In this case it describes the contents of a Rodin project.
Fig. 7. Requirements and model integration steps, from top to bottom: 1) A requirements document is constructed in its own environment; 2) External dependencies on OSLC-compliant providers are declared, the requirements tool is not aware of Event-B and Rodin and integration is purely at the OSLC level; 3) Requirements are linked to external resources via their URIs.
second part of the prototype achieves a similar goal for the Rodin Platform.

We have developed a plug-in for the Rodin platform, which exposes the Event-B model database and proofs as externally referable OSLC resources. It has no human-facing GUI but underlying RDF rendering of live model data can inspected by querying the adapter using with a web browser. Since a browser is typically XML-capable, the adapter returns data in the RDF/XML format (Fig. 6). Once again, every distinguished model element (variable, invariant, refinement) has a unique global identifiers that can be used to cross-link with other OSLC and RDF resources.

Exposition of internal data as OSLC is only the static part of intended collaboration. One needs to make different kinds of tools, e.g., used for requirements and modelling, aware of each, make them react on the respective changes and exchange relevant information when changes are made. One way to bring such dynamics would be have a form of the peer-to-peer connection architecture for every connection, where both parts maintain a server and a client. There are some practical reasons not go this route. One is the that the network address translation widely used to manage TCP/IP networks and connections to the Internet do not easily allow opening a connection from a client to a server. In fact, the dominating network architecture presupposes that connections are always initiated from local networks to dedicated servers. This has forced us to use a centralised approach where a single publisher/subscriber event server is managing all the collaborating tools. In this implementation, individual tools connect to a cloud hosted server to either create a new collaborative project or join an existing one. Then, within a project context, all the the project members can subscribe to and publish resource updates.

From the user perspective, the requirements editor can be cross-linked manually to some model elements. Such cross-links also appear when another user working on the specification part inserts a cross-link to the requirements. We use the OSLC creation functionality to allow engineers to insert model elements from requirements and vice versa. In the former case, the user can only choose the kind of a target element and its location in a model. Then an empty model element of a required type appears in the model with an embedded cross-link to the requirements. This functionality enables tight collaboration between members of a development team even when they are unable to communicate directly.

VII. RELATED WORK AND CONCLUSIONS

In [10], the authors propose an approach to bridge requirements to a formal specification in the context of the B Method. In particular, they have aimed at relating KAOS operations with B operations and defining properties as invariants of the constructed specification [10]. This work focuses on establishing traceability and facilitating formalisation without attempting to support a formalisation-driven development. Moreover, the authors do not rely on model refinement and, hence, building large-scale models would likely to be problematic. The main distinction of our approach from the existing works on applying formal modelling for safety-critical system development is in supporting an evolutionary iterative process to discovering system requirements. Such an approach promotes continuous learning of system properties and inter-dependencies. Moreover, it results in systematic derivation of a layered structure of system requirements that facilitates tracing of their cause-effect chains as well as verification of system correctness and safety.

In this paper, we have proposed a novel approach to formalisation-driven development of safety-critical systems in Event-B. Our approach is supported by the Requirements-Rodin adapter – a prototype tool that creates integrated information environment using linked data. The prototype relies on the OSLC standard that enables tool integration by specifying access to external tool resources. Since it allows for integration between arbitrary tools (open source or proprietary) and is agnostic to the implementation platform, it can support any automated environment for requirements engineering.

The proposed approach established an information continuum between requirements engineering and formal modelling. Its iterative and interactive nature enables tight co-operation between diverse teams and fits modern agile development technologies. Moreover, the approach can also facilitate a construction of the fragments of safety cases, as discussed in [11], interface with safety analysis [12]. It relies on the the experience of using Event-B in the industrial setting [13]. In our future work, we are planning to further extend the scope of the implementation and develop tools for model-based test generation and well as support traceability between models, tests and implementations.

REFERENCES

[8] OSLC-Core: (Open Services for Lifecycle Collaboration. OSLC Core specification.) "Online at http://iliasov.org/oslc/".
[9] OSLC-RM: (Open Services for Lifecycle Collaboration. OSLC Requirements Management (RM) specification.) "Online at http://open-services.net/bin/view/Main/RmSpecificationV2".