COMPUTING SCIENCE

Refinement-based Approach to Co-engineering Requirements and Formal Models

Alexei Iliasov, Linas Laibinis, Elena Troubitsyna, David Adjepon-Yamoah and Alexander Romanovsky

TECHNICAL REPORT SERIES

No. CS-TR-1456 March 2015

TECHNICAL REPORT SERIES

No. CS-TR-1456 March, 2015

Refinement-based Approach to Co-engineering Requirements and Formal Models

A. Iliasov, L. Laibinis, E. Troubitsyna, D. Adjepon-Yamoah, A. Romanovsky

Abstract

Formal modelling is widely recognised to contribute to the rigour and comprehensiveness of requirements. At the same time, a formal specification does not offer the flexibility and legibility of informal requirements, expected by system designers and software engineers. In this paper we propose a method and a supporting platform for tightly integrated co-engineering of a requirements document and the corresponding formal specification. We show that bi-directional transformation between requirements and models affect the practice of requirements construction by, arguably, bringing additional rigour and discipline while retaining the flexibility of informal requirements. We report on the experience of applying the OSLC framework to integrate a requirements engineering tool with the Rodin modelling and verification environment. A prototype implementation illustrates the main steps of the proposed approach.

Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

ILIASOV, A., LAIBINIS, L., TROUBITSYNA, E., ADJEPON-YAMOAH, D., ROMANOVSKY, A.,
Refinement-based Approach to Co-engineering Requirements and Formal Models
[By] A. Iliasov, L. Laibinis, E. Troubitsyna, D. Adjepon-Yamoah, A. Romanovsky
Newcastle upon Tyne: Newcastle University: Computing Science, 2015.
(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1456)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1456

Abstract

Formal modelling is widely recognised to contribute to the rigour and comprehensiveness of requirements. At the same time, a formal specification does not offer the flexibility and legibility of informal requirements, expected by system designers and software engineers. In this paper we propose a method and a supporting platform for tightly integrated co-engineering of a requirements document and the corresponding formal specification. We show that bi-directional transformation between requirements and models affect the practice of requirements construction by, arguably, bringing additional rigour and discipline while retaining the flexibility of informal requirements. We report on the experience of applying the OSLC framework to integrate a requirements engineering tool with the Rodin modelling and verification environment. A prototype implementation illustrates the main steps of the proposed approach.

About the authors

Alexei Iliasov is a Researcher Associate at the School of Computing Science of Newcastle University, Newcastle-upon-Tyne,UK. He got his PhD in Computer Science in 2008 in the area of modelling artefacts reuse in formal developments. His research interests include agent systems, formal methods for software engineering and tools and environments supporting modelling and proof .

Linas Laibinis is Adj.Professor at the Department of Information Technologies of Abo Akademi University, Finland. He got his PhD in Computer Science in 2000 on mechanised formal reasoning about computer programs. His research interests include interactive environments for proof and program construction, as well as application of formal methods to modelling and development of resilient and distributed computer-based systems.

Elena Troubitsyna is an Assoc. Prof. Abo Akademi University. She got her PhD in 2000 on design methods for dependable systems. Her research interests include application of rigorous modelling and analysis techniques to development and verification of resilient systems. She has been especially active in the area of fault tolerant and dependable computing. Elena Troubitsyna regularly serves at the program committees of resilient and dependable computing conferences.

David Ebo Adjepon-Yamoah is a PhD student at Newcastle University (Centre for Software Reliability, School of Computing Science). His research interests include system engineering, system architecture, system dependability, formal methods, and cloud computing.

Alexander (Sascha) Romanovsky is a Professor in the Centre for Software and Reliability, Newcastle University. His main research interests are system dependability, fault tolerance, software architectures, exception handling, error recovery, system structuring and verification of fault tolerance. He received a PhD degree in Computer Science from St. Petersburg State Technical University and has worked as a visiting researcher at ABB Ltd Computer Architecture Lab Research Center, Switzerland and at Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. In 1993 he became a postdoctoral fellow in Newcastle University, and worked on the ESPRIT projects on Predictable Dependable Computing Systems (PDCS), Design for Validation (DeVa) and on UK-funded projects on the Diversity, both in Safety Critical Software using Off-the-Shelf components. He was a member of the executive board of EU Dependable Systems of Systems (DSoS) Project, and between 2004 and 2012 headed projects on the development of a Rigorous Open Development Environment for Complex Systems (RODIN), and latterly was coordinator of the major FP7 Integrated Project on Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity (DEPLOY). He now leads work on fault tolerance in Systems of Systems within the COMPASS project and is Principal Investigator of Newcastle's Platform Grant on Trustworthy Ambient Systems.

Suggested keywords

CO-ENGINEERING
REQUIREMENTS
FORMAL MODELS
REFINEMENT

Refinement-based Approach to Co-engineering Requirements and Formal Models

Alexei Iliasov, David Adjepon-Yamoah, Alexander Romanovsky
Newcastle University
Newcastle Upon Tyne, UK
{alexei.iliasov, d.e.adjepon-yamoah, alexander.romanovsky}@ncl.ac.uk

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

***Abstract*—Formal modelling is widely recognised to contribute to the rigour and comprehensiveness of requirements. At the same time, a formal specification does not offer the flexibility and legibility of informal requirements, expected by system designers and software engineers. In this paper we propose a method and a supporting platform for tightly integrated co-engineering of a requirements document and the corresponding formal specification. We show that bi-directional transformation between requirements and models affects the practice of requirements construction by, arguably, bringing additional rigour and discipline while retaining the flexibility of informal requirements. We report on the experience of applying the OSLC framework to integrate a requirements engineering tool with the Rodin modelling and verification environment. A prototype implementation illustrates the main steps of the proposed approach.**

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]. 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 correctness of models but also spot deficiencies and inconsistencies in the requirements [3].

However, since models and requirements are often developed by different engineering teams, communication between formal modelling and requirements engineering activities is not straightforward [3]. To alleviate this problem, we propose an integrated approach relying on a new industry-driven interoperability standard – OSLC [4] – to enable automated co-engineering of requirements and formal models in Event-B.

Event-B is a top-down state-based framework to formal development [5]. 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 models and their refinements in Event-B is verified by proofs. An integrated extendable framework – Rodin platform [6] – provides an automated support for formal development in Event-B.

Refinement is a methodology supporting structured representation (and consequent validation) of requirements in formal models [7]. However, it typically assumes that the requirements are thoroughly described and classified before the actual modelling starts. The developed models and proofs then aim at confirming that the defined requirements are indeed complete and consistent. Such a methodology does not allow the engineers to fully exploit the benefits of formal modelling [8].

In this paper, we propose an alternative approach that enables and automates co-evolution of requirements and formal models. In our work, the starting point is an informal high-level description of the system behaviour. The incremental definition of requirements proceeds concurrently with model construction, while the conducted proofs verify each 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. The process iteratively progresses until the desired level of detail is reached.

By far and large, majority of requirements documents are written in natural language [9]. In our approach, we aim at retaining flexibility of the natural language requirements description. Hence, to facilitate representation of the informally described requirements in formal models, we merely provide the mapping guidelines, i.e., do not attempt to restrict the requirements representation. To maintain the link between the dynamically changing requirements description and the associated formal models, we have created a prototype Requirements-Rodin adapter [10]. It relies on OSLC – Open Services for Life Cycle Collaborations [4] – a newly introduced open standard for integrated information engineering. The standard allows the engineers to achieve inter-operability between engineering tools by specifying the access to the external resources of those tools as linked data [11], [12].

The proposed approach is illustrated by a case study – an airlock control system. We demonstrate how the requirements description and the associated Event-B models evolve through their co-engineering and highlight the feedback provided by the formalisation. The Requirements-Rodin adapter automates creation and maintenance of an information continuum between the requirements engineering and Event-B modelling activities.

We believe that the approach proposed in this paper amplifies the benefits of formal modelling, improves communication between different engineering teams and establishes common information space in the engineering of complex systems.

The paper is structured as follows. Section II presents the essentials for formal modelling and verification in Event-B. In Section III, we describe our approach to co-engineering of requirements and formal models. Section IV demonstrates the approach in detail by presenting a small case study – an airlock control system. In Section V, we briefly overview our chosen tool integration framework – OSLC. Section VI presents our prototype implementation of the proposed tool integration. Finally, in Section VII, we overview the related work and give some concluding remarks.

II. Modelling and Verification in Event-B

Event-B is a state-based formal approach that promotes the correct-by-construction approach to system development and formal verification by theorem proving. In Event-B, a system model is specified using the notion of an *abstract state machine* [5], [13]. 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.

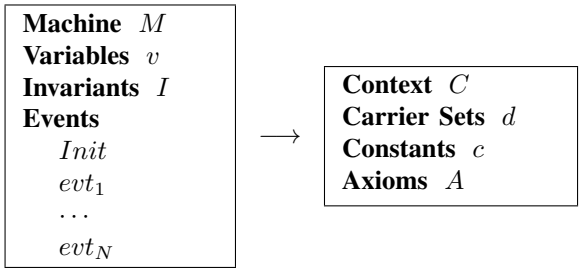

Fig. 1. Event-B machine and context

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:

$$e \mathrel{\widehat{=}} \textbf{any } a \textbf{ where } G_e \textbf{ then } R_e \textbf{ end},$$

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.

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract specification that nondeterministically models the 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*. When data refinement is performed, we define *gluing invariants* as a part of the invariants of the refined machine. They define the relationship between the abstract and concrete variables.

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 the relevant verification conditions – proof obligations.

An Event-B model should satisfy a number of such proof obligations. The most important class of them include *invariant preservation* properties. More precisely, each event e of the abstract Event-B model should preserve the given model invariant I:

$$A(d,c),\ I(d,c,v),\ G_e(d,c,x,v),\ R_e(d,c,x,v,v') \ \vdash\ I(d,c,v') \qquad \text{(INV)}$$

where A are model axioms, I are the model invariants, d and c are model constants and sets respectively, x are the event's local variables and v, v' are the variable values before and after event execution.

Moreover, the invariant I must be established after model initialisation:

$$A(d,c),\ R_{Init}(d,c,v') \ \vdash\ I(d,c,v') \qquad \text{(INIT)}$$

In turn, each refinement step generates additional proof obligations ensuring that the transformation is performed in a correctness-preserving way. For brevity, here we show only a couple of the most essential ones, which we will use in the paper.

Let a shorthand $H(d,c,v,w)$ stands for the hypotheses $A(d,c),\ I(d,c,v),\ I'(d,c,v,w)$, where $I,\ I'$ are respectively the abstract and refined invariants, and v, w are respectively the abstract and concrete variables. The event guards in a refined model can be only strengthened in a refinement step:

$$H(d,c,v,w),\ G'_e(d,c,x,w) \ \vdash\ G_e(d,c,x,v) \qquad \text{(REF\_GRD)}$$

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

Moreover, the *simulation refinement* property requires to show that the "execution" of a refined event is not contradictory with its abstract version:

$$H(d,c,v,w),\ G'_e(d,c,x,w),\ R'_e(d,c,x,w,w') \\ \vdash \exists v' \cdot R_e(d,c,x,v,v') \ \wedge\ I'(d,c,v',w') \qquad \text{(REF\_SIM)}$$

where R_e, R'_e are respectively the abstract and concrete next-state relations of the same event evt_i.

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.

Rodin is an open platform, modelling and verification in which is enhanced by a number of extensions (plug-ins). The extensions provide us with the different ways to represent models. They also give us an access to various verification engines (theorem provers, model checkers, SMT solvers). For instance, model liveness properties may be expressed in LTL/CTL and then verified in the associated model checker – ProB, while model use cases can be represented graphically and then translated into theorems to be proved by the Rodin theorem proving engine. In the rest of the paper, we use Event-B models as a frontend for a variety of models used in formal verification.

III. Co-engineering of requirements and formal models

A. Co-development process

In this section we present our general approach for requirements elicitation via co-engineering of a requirements and the formal models in Event-B.

In his seminal book, Lamsweerde [9] introduces iterative process of requirements engineering. In this process, each iteration aims at improving quality of the requirements definition by incorporating acquired knowledge of domain and the system as well as the verification feedback. Our approach follows the same idea and adopts it to align with the refinement approach to formal modelling. We also slightly modify the starting point of the requirements engineering process and start from a conceptual requirement description that is made detailed through the iterative co-development with formal modelling. This process is similar to goal decomposition, i.e., the initial iterations focus on defining and verifying higher-level requirements, while the later iterations introduce increasingly detailed requirements.

Each iteration of our approach consists 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 refinement of more abstract model;
- attempting to verify by proofs logical consistency of the model;
- modifying the model, if necessary, by, e.g., adding additional invariants or constraints, to make the models provably correct;
- reflecting model modifications in the requirements definition by adding missing requirements or correcting the defined ones;
- repeating the process ...

Essentially, our approach proposes structured requirement elicitation, which is aligned with top-down refinement of formal models in Event-B. In other words, the development of a requirements document and the related formal models proceeds hand-in-hand – changes in one are reflected in the other and vice versa. Every time when the requirements engineers define a new subset of requirements in the requirements document, formalisation of this subset is triggered in the verification team. Verification of the created formal models often necessitates the model changes, i.e., spots problems with the introduced subsset of the requirements. In its turn, it triggers changes in the requirements definition – the missing requirements are added or erroneous are corrected. Such a process can be seen as as "proof-driven" requirements discovery.

To facilitate co-engineering of the requirements description and the formal models, we define recommendations on how to map the requirements onto the associated model elements. Our mapping helps to ensure that each informal requirement is reflected by a certain modelling artefact or a group of them. On the other hand, every modelling decision must be justified from the requirements point of view.

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.

The hierarchical development approach results in spreading representation of the requirements over a number of different models. Different nature of the requirements would motivate the engineering to choose the formalisation that enables the most efficient verification (e.g., theorem proving, model checking or SMT solving). We believe that such a flexibility is beneficial for the requirements engineering 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, ▭ depicts here requirement elicitation or specification development; ▬ is the specification proof effort following a specification change; ▤ represents the integration activity where model changes are reconciled with requirements (typically resulting in new added requirements); finally, ⊏⊐ 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.

B. Construction of the mapping between requirements and formal models

To facilitate the proposed co-engineering process, we define the recommendations for mapping requirements onto the formal models. In general, unambiguous translation of textual requirements into the corresponding elements of formal methods, i.e., a formalisation of requirements, is a challenging unresolved problem. In the related work section, we overview various approaches to achieve it.

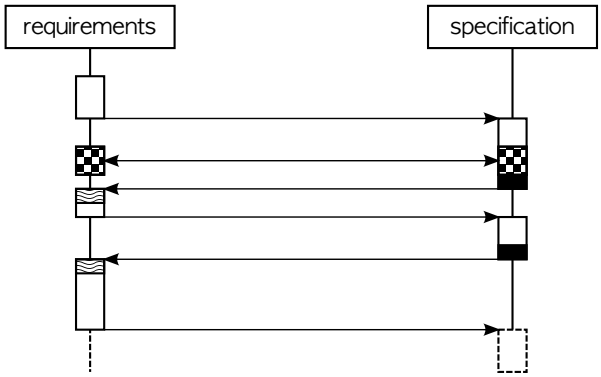

Fig. 2. Requirements and specification developed in parallel

In our approach, we rely on our extensive expertise in Event-B modelling that resulted in creating classification of the requirements and defining the guidelines for their modelling [14]. The most typical requirements can be classified as safety requirements (SRs) about global system properties, requirements about flow of control (i.e., order of function execution), termination conditions, temporal properties etc.

Formally, the mapping is a function, F_M, of the type:

$$F_M : \ Reqs \ \rightarrow \ \mathcal{P}(\mathit{MElem})$$

where $\mathcal{P}(T)$ stands for all possible subsets of the type T, and *MElem* represents all possible model elements. In other words, a single requirement is mapped to a set (collection) of model elements. Here model elements are referable elements of Event-B formal models, such as model axioms, variables, invariants, theorems, model events, event guards and actions, etc. Moreover, the associated model expressions such as LTL/CTL formulas used for model checking or use case diagrams to be verified are also considered as model elements that can be referred to.

As mentioned above, requirements from different classes may be mapped (translated) into the associated formal model in differing ways. For instance, a fragment of the mapping

$$SR_1 \ \mapsto \{\mathit{safety\_inv}_2, \ \mathit{safety\_inv}_3\}$$

associates the safety requirement labeled SR_1 with the corresponding model invariants, labeled as $\mathit{safety\_inv}_2$ and $\mathit{safety\_inv}_3$. The requirement label here refers to the associated textual requirement description, while the invariant labels refer to specific logical sequents (theorems) to be proved for a model to ensure that the safety property is preserved.

In general, the following guidelines for constructing the mapping have been tested to work in practice:

1) the requirements expressing safety properties are translated into safety invariants or event guards (preconditions) of the related formal models;
2) liveness properties become LTL/CTL temporal properties to be verified by the associated model checker;
3) system properties are represented by state transitions (model events) as well as guards or post-conditions (the constructed theorems) of those events;
4) the requirements expressing sequential properties (such as the desired system control flow) and use cases become theorems on the event order to be verified;
5) design decisions become either model events or guards or post-conditions of particular events.

In all cases, new model variables may be also introduced. If the mapping relates a requirement with such artefact as an invariant, a theorem, or an LTL/CTL formula, its verification gives us immediate proving feedback on the requirement. In other cases, like adding new events or variables, we rely on the underlying Event-B proof semantics (i.e., the pre-defined proof obligations for model consistency and refinement) to provide such a feedback. We will demonstrate building of a concrete mapping between a requirements documents and the corresponding Event-B models on a case study in Section IV.

We require certain consistency (well-formedness) conditions to be fulfilled by a constructed mapping. First, we expect mapping to be total and surjective, which means that any requirement is mapped to some model elements, while any model element is associated with at least one informal requirement. The consistency of hierarchical structures of requirements and model refinements must be also respected: more concrete requirements must be mapped onto more concrete refined models. Mathematically, this means that a mapping is an order preserving transformation between the corresponding data structures.

C. Tool support

Development of the requirements engineering field shows that it is critical to ensure flexibility in the requirements elicitation process [9]. Therefore, while devising the tool support for the proposed approach, we aim at creating a platform for collaborative yet non-restrictive co-engineering. Our goal is to establish an information continuum between two teams: the team of domain experts working on the requirements document and the verification experts responsible for construction and verification of formal models. The desired integration should be loose and non-intrusive, i.e., should support the existing modus operandi in each of the teams. However, the integration should also enable highly-interactive collaboration, i.e., trigger the actions on each side by exposing the relevant information to the engineers.

OSLC [4] – *Open Services for Lifecycle Collaboration* – provides the means for such a kind of integration. In its essence, OSLC ensures interoperability between arbitrary engineering tools by specifying a number of constraints that resources, which are externally exposed by the tools, must preserve and a basic protocol allowing for the tools to integrate their activities on these resources.

In the next section, we will demonstrate our approach by a case study – an airlock control system. Then in Section V we will overview the OSLC integration framework in more detail, before briefly discussing (in Section VI) our OSLC-based prototype implementation illustrating the proposed tool integration.

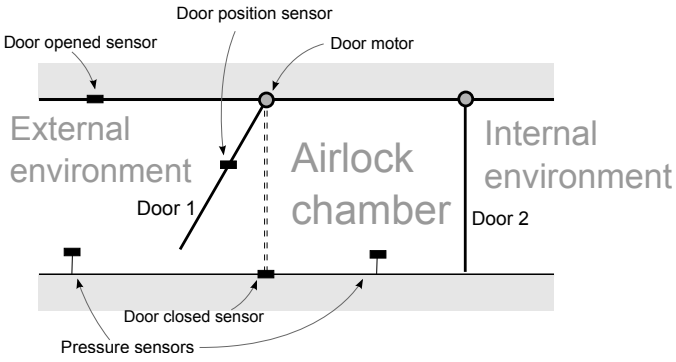

Fig. 3. The airlock system

IV. Approach Demonstration: an Airlock Control System

We illustrate the approach with a case study of an airlock system control. The main function of the airlock is to separate two areas with different air pressures and allow 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 positions 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.

At this stage we switch to the specification part of our approach and try to capture the purpose of the system as an abstract Event-B model. We are going to follow the top-down development methodology where a well-chosen abstraction "frames" the further model refinement steps, while the refinement proof obligations suggest missing model elements. Hence, we initially abstract away of the notion of airlock as a pair doors and represent it as a monolithic door-like entity. In this way we can also omit details of the airlock chamber and pressure control, only to introduce them later as refinements of both requirements and associated formal models.

The sole phenomenon we capture in the abstract machine *m0* is the movement of a human operator between the external and internal environments. This succinctly, in an abstract form, encodes the requirements ENV1, ENV2, and FUN1.

The location of a user is represented by model variable *user* defined to be an element an enumerated set USER_POS0:

$$\text{inv1}: \; user \in \text{USER\_POS0}$$

where USER_POS0 is defined in the accompanying model context *c0* as

$$\text{axm1}: \; \text{USER\_POS0} = \{ \text{OUT}, \; \text{IN} \}.$$

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

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

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

Model consistency proof obligations [5] require that every model variable is initialised, possibly non-deterministically, establishing the given invariant (for details, see the proof obligation (INIT) in Section II). To satisfy this, we must make a decision where the human operator may be initially located. It is clear from the description that it could be either environment and hence the following initialisation statement is introduced.

$$INITIALISATION \triangleq \textbf{begin} \;\; user :\in \text{USER\_POS0} \;\; \textbf{end}$$

The initialisation action $user :\in \text{USER\_POS0}$ is then translated into a new requirement FUN2, thus adding to the overall list of requirements.

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

On top of requirements document and Event-B model we must also maintain a consistent mapping relation between requirements. As a result of the co-development presented so far, we constructed the following mapping:

ENV1 $\rightarrow$ *c0*/axm1
ENV2 $\rightarrow$ *c0*/axm1, *m0*/inv1
FUN1 $\rightarrow$ *m0*/*go_in*, *m0*/*go_out*
FUN2 $\leftarrow$ *m0*/*INITIALISATION*

The overall co-engineering process for the conducted case study is graphically portrayed in Fig. 4.

A. First refinement

As the next step, we shall introduce an airlock abstraction that operates much like a simple door. Its role here is to 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. In the requirements document, we introduce the following (abstract) requirements:

FUN3. Environments are separated by an airlock.

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

We also *elaborate on* the abstract requirement FUN2 with the following statement:

FUN5(FUN2). Airlock is initially in the ALWAIT mode.

In the specification part, we introduce a new *refinement step* – (refinement machine $m1$) to signify that the existing abstraction is extended with a new phenomena. In this refinement, the airlock state is modelled by a new variable, *alck*, defining the current airlock mode, as prescribed by the requirement FUN4.

$$\text{inv1}: \ alck \in \text{AIRLOCK\_STATE}$$

The requirement FUN5 is reflected in the following new initialisation statement:

$$INITIALISATION \triangleq \textbf{begin} \ \text{act2}: \ alck := \text{ALWAIT} \ \textbf{end}$$

At the basic level, the airlock behaviour model must cover the transitioning between the three airlock modes. This results in the following new specification events.

$$airlock\_operate \triangleq$$
$$\textbf{when} \ \ alck = \text{ALWAIT} \ \ \textbf{then} \ \ alck :\in \{\text{ALIN}, \text{ALOUT}\} \ \ \textbf{end}$$

$$airlock\_done \triangleq$$
$$\textbf{when} \ \ alck \neq \text{ALWAIT} \ \ \textbf{then} \ \ alck := \text{ALWAIT} \ \ \textbf{end}$$

The events elaborate upon the requirement FUN5 and are mapped back into two new requirements, elaborating on FUN5:

FUN6(FUN5). The airlock may be switched to ALIN or ALOUT mode.

FUN7(FUN5). The airlock may be switched to ALWAIT mode.

The model makes it clear that the airlock behaviour must be harmonised with user movement. We simply state that when the user is moving between locations, the airlock is put in the corresponding mode. For instance, the case when the user moves inside is addressed by adding new guard $alck = \text{ALIN}$:

$$go\_in \triangleq \textbf{when} \ \ \cdots \wedge \text{grd2}: \ alck = \text{ALIN} \ \ \textbf{then} \ \ \ldots \ \ \textbf{end}$$

The correctness of event refinement is verified by discharging the associated proof obligations for guard strengthening and event simulation (see Section II).

The two new guards are translated to new symmetric requirements.

FUN8(FUN1). A user may travel to IN only when airlock is in the ALIN mode.

FUN9(FUN1). A user may travel to OUT only when airlock is in the ALOUT mode.

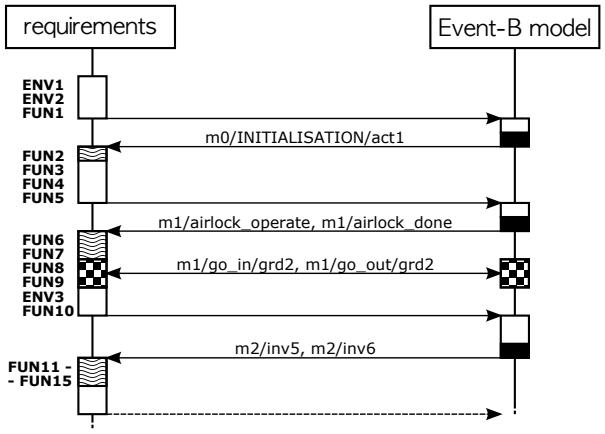

Fig. 4. Requirements and model co-engineering time line.

Here we have a new situation where requirements detalisation 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 detalisation and refinement are tightly interlinked, on the diagram in Fig. 4 we show the corresponding step as a joint action (the chessboard pattern).

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

FUN3 $\rightarrow$ $m1/alck$
FUN4 $\rightarrow$ $m1/\text{inv1}$
FUN5 $\rightarrow$ $m1/INITIALISATION/\text{act2}$
FUN6 $\leftarrow$ $m1/airlock\_operate$
FUN7 $\leftarrow$ $m1/airlock\_done$
FUN8 $\leftrightarrow$ $m1/go\_in/\text{grd2}$
FUN9 $\leftrightarrow$ $m1/go\_out/\text{grd2}$

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.

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.

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:

$door1\_open \triangleq$
when $door1 = \text{CLOSED}$ **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 $alck$. Consequently, the events $airlock\_operate$ and $airlock\_done$ as well as the guard mentioning $alck$ 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 $alck = \text{ALOUT}$, while $door2\_open$ covers the case of $alck = \text{ALIN}$. 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:

inv5: $\neg (door1 = \text{OPEN} \wedge door2 = \text{OPEN})$
inv6: $door1 = \text{CLOSED} \wedge door2 = \text{CLOSED} \Leftrightarrow alck = \text{ALWAIT}$

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:

$door1\_open \triangleq$
when $\cdots \wedge$ grd2: $door2 = \text{CLOSED}$ **then** ... **end**

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 $door2 = \text{OPEN}$ for go_in and $door1 = \text{OPEN}$ 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: $user = \text{OUT} \wedge door2 = \text{OPEN} \Rightarrow alck = \text{ALIN}$
inv4: $user = \text{IN} \wedge door1 = \text{OPEN} \Rightarrow alck = \text{ALOUT}$

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	$\rightarrow$	$m2/door1\_open$, $m2/door1\_close$, ...
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.

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 then, in a single instance, move in or out. The model and requirements abstract away pressure equalisation and operation of the second door. In the new refinement step we refine airlock operation with the notions of a middle chamber and explicit operation of both doors. First, we add requirements that the airlock must be visited when using the airlock:

FUN16. User moves from U_OUT to U_MID_IN.

FUN17. User moves from U_OUT to U_MID_IN.

This translates to the following new events:

$go\_mid\_in \triangleq$
when
$userpos = \text{U\_OUT} \wedge door1 = \text{OPEN}$
then
$userpos := \text{MID}$
end

$go\_mid\_out \triangleq$ **when** ... **then** ... **end**

To relate the in/out position to the in/middle/out, we put the following invariant statements in the model:

inv2: $userpos \in \{\text{U\_IN}, \text{U\_MID\_OUT}\} \Leftrightarrow user = \text{IN}$
inv3: $userpos \in \{\text{U\_OUT}, \text{U\_MID\_IN}\} \Leftrightarrow user = \text{OUT}$

All of the model changes above translate into adding a liveness requirement

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 are translated into the following LTL statements over the machine states that are forwarded to be verified by the associated Event-B model checker – Pro-B:

$\mathbf{G}(\{userpos = \text{U\_IN}\} \Rightarrow \mathbf{F}\{userpos = \text{U\_OUT}\})$
$\mathbf{G}(\{userpos = \text{U\_OUT}\} \Rightarrow \mathbf{F}\{userpos = \text{U\_IN}\})$

The model checker verification yields several counterexamples, which consequently leads to several model corrections. In turn, new concrete requirements are added to

the requirements document. All in all, this step introduces nine new functional requirements synthesised during model validation. 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.

The presented case study illustrates our proposed co-engineering approach. However, to improve its applicability and usability, we need tools supporting integrated co-engineering work of different developer teams. In the next section we present the basis for such tool integration – the OSLC framework.

V. OSLC

Open Services for Lifecycle Collaboration (OSLC) [4] 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 [15]. 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.

The central organising concept of OSLC is *ServiceProvider*, enabling tools to expose resources and allowing consumers to navigate to all of the resources, and create new ones. Two fundamental properties of a ServiceProvider are:

1) oslc:creation: the URL of a resource to which you can POST representations to create new resources.
2) oslc:queryBase: the URL of a resource that you can GET to obtain a list of existing resources.

ServiceProviders have a third important property – dialog, describing invocation of HTML web user interface dialogs of one tool by another.

a) Requirements in OSLC: OSLC Requirements Management (RM) [11] 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. A small excerpt from this table from the OSLC-RM specification is given on Fig.5.

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 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

Prefixed Name	Occurs	Read-only	Value-type	Represen-tation	Range	Description
Relationship properties: This grouping of properties are used to identify relationships between resources managed by other OSLC Service Providers						
oslc_rm:elaboratedBy	zero-or-many	False	Resource	Reference	any	The subject is elaborated by the object. For example, a user requirement is elaborated by use case.
oslc_rm:elaborates	zero-or-many	False	Resource	Reference	any	The object is elaborated by the subject.
oslc_rm:specifiedBy	zero-or-many	False	Resource	Reference	any	The subject is specified by the object. For example, a requirement is elaborated by a model element .
oslc_rm:specifies	zero-or-many	False	Resource	Reference	any	The object is specified by the subject.

Fig. 5. Excerpt from OSLC-RM resource properties

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. 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:

$$\texttt{host} : \texttt{port}/\langle\text{project-name}\rangle/\langle\text{requirement-name}\rangle$$

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 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, 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 kinds of tools – for requirements and modelling – aware of each, make them react on 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 crossed 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.

Instructions on how to use the developed OSLC adapter for the Rodin platform can be found in [10]. The adapter adds an embedded http server (jetty) and implements, with the help of the Lyo framework and Apache Wink, a RESTful java servlet that provides access to the Rodin projects and models.

VII. Related Work and Conclusions

A. Related work

The main goal of requirements engineering is to derive structured, complete and consistent set of requirements, which forms the basis for further system development. Often this is achieved via formal modelling and verification. Consequently, the research on methodologies for translating informal requirements into formal specifications is vast. One strand of the research relies on natural language processing [16] to automatically extract formal specifications. Another strand focuses on structuring requirements to facilitate their formalisation. Fraser et al. [17] are pioneers in this area. They proposed guidelines for developing VDM specifications from structured analysis of requirements. Giese et al [18] demonstrated how

to relate informal requirements in the form of UML use cases to formal representation in OCL. The most prominent work in this area is the KAOS framework [9], [19]. KAOS is a goal-oriented framework that aims at formalising hierarchy of goals and corresponding sub-goals in Linear Temporal Logic. Another intensifying strand of research relies on ontologies for imposing a structure on informal requirements and then proposing patterns for translating them [20].

Our research builds on the results of this work emphasising, however, a co-engineering approach. Firstly, we take a different point of departure and start from an informal high-level system description, i.e., without attempting to restrict the form of the requirements representation or define a structure over them. Secondly, we propose to define requirements in small increments and immediately verify them. Therefore, our approach supports shorter iterations of requirements elicitation and checking.

An initial approach to bridge requirements to specification in the context of the B Method have aimed at relating KAOS operations with B operations and defining properties as invariants of the specification [21]. Further development in this area has continued along providing a formalisation of the requirements defined as hierarchy of goals to Event-B [22]. This work focused on establishing traceability and facilitating formalisation without attempting to support co-development. Moreover, the authors do not rely on refinement and, hence, building large-scale models would likely be problematic.

B. Conclusions

In this paper, we have proposed a novel approach to requirements elicitation. It is based on co-development of requirements description and the associated formal models. The approach is supported by a prototype tool based on the new industry-driven standard for tool integration – OSLC. In our work, we aimed at providing both requirements engineering and formal modelling teams with flexibility and highly interactive development environment. Requirements engineers incrementally introduce definitions of the requirements. This is reflected in the corresponding refinements of the formal model. Proof-based verification provides a quick feedback leading to requirements adjustment. In such an approach, the role of proofs is enhanced to guide development of the requirements.

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.

In this paper, we have reported on the methodology and the prototype tool development. To evaluate the proposed approach, we are currently setting an industrial pilot. Collecting the feedback and refining the proposed approach constitute our next steps.

REFERENCES

[1] Rushby, J.: Disappearing Formal Methods. In: High-Assurance Systems Engineering Symposium, Association for Computing Machinery (2000) 95–96

[2] Woodcock, J., Larsen, P., Bicarregui, J., Fitzgerald, J.: Formal Methods: Practice and Experience. ACM Computing Surveys **41(4):19:1** (2009) 19–36

[3] Romanovsky, A., Thomas, M., eds.: Industrial Deployment of System Engineering Methods. Springer-Verlag Berlin Heidelberg (2013)

[4] OSLC: (Open Services for Lifecycle Collaboration.) "Online at http://open-services.net/".

[5] Abrial, J.R.: Modeling in Event-B. Cambridge University Press (2010)

[6] RODIN: Event-B Platform. http://www.event-b.org/ (2009)

[7] Abrial, J. In: A Mechanical Press Controller Development. Volume Volume 22: Engineering Methods and Tools for Software Safety and Security of NATO Science for Peace and Security Series - D: Information and Communication Security. (2009) 1–42

[8] Gmehlich, R., Grau, K., Iliasov, A., Jackson, M., Loesch, F., Mazzara, M.: Towards a Formalism-Based Toolkit for Automotive Applications. Volume 1311.6145., CoRR (2013)

[9] Lamsweerde, A.v.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley (2009)

[10] Rodin OSLC Adapter: (Using Instructions) "Online at http://open-services.net/bin/view/Main/OslcCoreSpecification".

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

[12] OSLC-CM: (Open Services for Lifecycle Collaboration. OSLC Change Management (RM) specification.) "Online at http://open-services.net/bin/view/Main/CmSpecificationV2".

[13] Metayer, C., Abrial, J., Voisin, L., eds.: Rodin Deliverable D7: Event-B language. Project IST-511599, School of Computing Science, Newcastle University (2005)

[14] Prokhorova, Y., Laibinis, L., Troubitsyna, E.: Facilitating Construction of Safety Cases from Formal Models in Event-B. Information and Software Technology **60** (2015) 5176

[15] OSLC-Core: (Open Services for Lifecycle Collaboration. OSLC Core specification.) "Online at http://iliasov.org/oslc/".

[16] Rolland, C., Proix, C.: A Natural Language Approach for Requirements Engineering. In: Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE. (2013) 35–55

[17] Fraser, M.D., Kumar, K., Vaishnavi, V.K.: Informal and Formal Requirements Specification Languages: Bridging the Gap. IEEE Trans. Software Eng. **17**(5) (1991) 454–466

[18] Giese, M., Heldal, R.: From Informal to Formal Specifications in UML. In: UML 2004 - The Unified Modelling Language: Modelling Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004. Proceedings. (2004) 197–211

[19] Dardenne, A., Lamsweerde, A.v., Fickas, S.: Goal-Directed Requirements Acquisition. Sci. Comput. Program. **20**(1-2) (1993) 3–50

[20] Li, F., Horkoff, J., Mylopoulos, J., Guizzardi, R.S.S., Guizzardi, G., Borgida, A., Liu, L.: Non-functional requirements as qualities, with a spice of ontology. In: IEEE 22nd International Requirements Engineering Conference, RE 2014, Karlskrona, Sweden, August 25-29, 2014. (2014) 293–302

[21] Ponsard, C., Dieul, E.: From Requirements Models to Formal Specifications in B. In: Proceedings of the CAISE*06 Workshop on Regulations Modelling and their Validation and Verification ReMo2V '06, Luxemburg, June 5-9, 2006. (2006)

[22] Aziz, B., Arenas, A., Bicarregui, J., Ponsard, C., Massonet, P.: From Goal-Oriented Requirements to Event-B Specifications. In: First NASA Formal Methods Symposium - NFM 2009, Moffett Field, California, USA, April 6-8, 2009. (2009) 96–105