Abstract

This paper discusses the need and leveraging potential of formal modeling and web technology for progressing towards the goal of automating the establishment, maintenance and assessment of the completeness of traceability and the consistency of the requirements. The generic Augmented Lifecycle Space method, devised in an earlier paper, is applied as the approach to improve the capability of software processes requiring bidirectional traceability as well as consistency of the requirements in either homogeneous or heterogeneous development environments capitalizing on the emerging Open Services for Lifecycle Collaboration (OSLC) initiative. One of the important features of the presented new approach is that it allows for the so called “graceful integration” of formal modeling. Formal modeling is fundamentally necessary for securing completeness and consistency, but customarily rejected due to the usually prohibiting up-front effort needed to formally process all artifacts of an already established traditional system; Graceful integration can considerably lower this threshold.

Keywords

Application lifecycle management - Process assessment - Process improvement - Formal modeling - Open services for lifecycle collaboration - Tools integration - Heterogeneous tool environment - Requirements traceability - Requirements consistency
Graceful Integration of Process Capability Improvement, Formal Modeling and Web Technology for Traceability

Miklós Biro1,3, Felix Kossak1, József Klespitz2, and Levente Kovács2

1 Software Competence Center Hagenberg, Hagenberg, Austria
{miklos.biro, felix.kossak}@scch.at
2 Óbuda University, Budapest, Hungary
klespitz.jozsef@phd.uni-obuda.hu,
kovacs.levente@nik.uni-obuda.hu
3 Johannes Kepler Universität, Linz, Austria

Abstract. This paper discusses the need and leveraging potential of formal modeling and web technology for progressing towards the goal of automating the establishment, maintenance and assessment of the completeness of traceability and the consistency of the requirements. The generic Augmented Lifecycle Space method, devised in an earlier paper, is applied as the approach to improve the capability of software processes requiring bidirectional traceability as well as consistency of the requirements in either homogeneous or heterogeneous development environments capitalizing on the emerging Open Services for Lifecycle Collaboration (OSLC) initiative. One of the important features of the presented new approach is that it allows for the so called “graceful integration” of formal modeling. Formal modeling is fundamentally necessary for securing completeness and consistency, but customarily rejected due to the usually prohibiting up-front effort needed to formally process all artifacts of an already established traditional system; Graceful integration can considerably lower this threshold.

Keywords: Application lifecycle management · Process assessment · Process improvement · Formal modeling · Open services for lifecycle collaboration · Tools integration · Heterogeneous tool environment · Requirements traceability · Requirements consistency

1 Introduction

The ultimate need addressed in this paper is functional safety extensively discussed in the IEC 61508 series of international standards setting out “the requirements for ensuring that systems are designed, implemented, operated and maintained to provide the required safety integrity level (SIL)” [19]. The use of systems in all industries, including automotive, transportation, medical, manufacturing, nuclear industries for instance, involves a foreseen level of risk of causing harm that must always be reduced to a level “as low as reasonably practicable” (ALARP) as required by ethics, regulatory regimes, and standards.
The key requirements of processes producing safety-critical systems, addressed in this paper, are the completeness of bidirectional traceability and the consistency of the requirements of the system under assessment which appear in all process assessment and improvement models [24]. As already highlighted in [10], traceability is fully recognized as a key issue by the agile community as well [2, 3].

The addressed software development processes naturally involve artifacts that can only be created by humans (customers, sales, marketing, etc.). Yet, there are other artifacts which can hardly be managed manually, including, for example, the documentation of low level test results or results of automated testing (e.g. static and/or dynamic code analysis). Similarly, the number of relationships, including traceability links, between the different artifacts becomes prohibitive even in the simplest practical cases [8, 9], so the handling and maintenance requires automated support which is the direct goal of our research [29, 30].

The commercially available tools supporting the above mentioned processes are called Application Lifecycle Management systems (ALMs) which do not only cover the implementation, but the whole process starting from the initial idea, closing with the end of the product’s life [12]. When a company chooses to set up an ALM, it can choose among numerous off-the-shelf or third party software systems and/or can decide to develop needed elements and optionally complement them with other management tools. And adapting Murphy’s law to this case, whatever can happen will happen, so many companies are effectively using a heterogeneous variety of ALM tools as pointed out for example in [21, 26, 28].

People and process challenges as well as fundamental logical and technical barriers for assessing and improving the completeness of traceability and verifying the consistency of the requirements in either homogeneous or heterogeneous tool environments were discussed in [10] also referring to [13, 16]. The Augmented Lifecycle Space approach was introduced in [10] as a generic method building on an arbitrary model containing traceability and consistency requirements, allowing the automation of the assessment, and facilitating the improvement of the completeness of traceability as well as the verification of the consistency of the requirements in either homogeneous or heterogeneous tool environments.

In this paper, we present the initial development phases of the system we call Requirements Traceability and Consistency checking Tool (RTCT) planned to integrate the latest achievements independently evolving in the fields of process capability improvement, formal modeling, and web technologies. It is interesting to remark that these fields clearly match the process areas of organization (O), methodology (M) and technology (T) devised in the historically significant European BOOTSTRAP software process assessment and improvement methodology [7].

Beyond the general goals, Sect. 2 presents use cases that RTCT must be able to handle exploiting OSLC technology and leading to the capability of automatically checking bidirectional traceability and consistency of the requirements in a possibly heterogeneous lifecycle management system under assessment, and proving that all of its necessary artifacts are available and properly linked.

The traceability and consistency requirements shall be based on the V-model of the software development lifecycle also applied in the Automotive SPICE standard [5] as depicted in Fig. 1. All of the arrows represent bidirectional traceability
One of the features of the new approach, of high importance for actual applications but rarely considered by promoters of formal modeling, is that it is possible to introduce the newly designed requirements management system incrementally, so that only new or changed requirements are affected and traced. This means that only new or changed requirements need to be formally modeled at the start, making the operational introduction of the new system practically feasible.

Let us use the term “graceful integration” for the above capability of incrementally introducing formal modeling. Formal modeling is fundamentally necessary for securing completeness and consistency, but customarily rejected due to the usually prohibiting up-front effort needed to formally process all artifacts of an already established traditional system; Graceful integration can considerably lower this threshold. In contrast to the established term “graceful degradation” for the ability of a system to maintain functionality when portions of the system break down, “graceful integration” means the ability of a system to be incremented so that its functionality is improved without significant disruption.

For consistency checking, a formalized representation of all relevant requirements will have to be available however. The need for the feature of the incremental introduction of the formal modeling based system originates from an actual safety-critical software development environment providing the fundamental industrial background of this research.

The proposed RTCT prototype shall have the following features:

- Create a collection of artifacts (typically corresponding to a project; possible artifacts include requirements and test cases, for example).
- Add an artifact to a collection or actually a link to an artifact/URI.
• Add a traceability link between artifacts.
• Show which test cases (etc.) are linked with a requirement and that each require-
ment is linked with at least one test case (etc.).
• Show which requirement(s) a test case or another artifact belongs to.
• Check whether all requirements of a given collection are mutually consistent.
• Check whether all required artifacts are linked with a given requirement.

The desired requirements traceability tool (RTCT) is illustrated in more detail in the
next section. Single elements of the envisaged RTCT software have already been
investigated by different researchers. The present work is based on [10] which also
gives an overview over the topic.

Regarding models for trace completeness, [27] study models for realizing
requirements traceability but do not consider process models as a source for trace
completeness checking. The same holds e.g. for [18], who make suggestions for
document-based requirements tracing.

Regarding requirements consistency checking, [20] state that “only few require-
mments tools are available in the market to provide facilities for the validation process
especially for verifying and checking the requirements qualities, such as the com-
pleteness and consistency”. This paper gives a good overview over approaches and
tools for heuristic and formal requirements consistency checking as well as mixed
approaches. However, most of the cited approaches or tools do not seem applicable for
the setting envisaged by us. Some approaches address high-level structure consistency
only, some will obviously yield highly incomplete results e.g. because they rely on
natural language analysis, and others appear to be rather demanding for users and
require special modelling that cannot be reused in implementation.

An exception is [Sousa et al., 2010], who use the formal method B for requirements
consistency checking in a way which fits our approach in principle, except for ques-
tions of tool integration. Interestingly, they consider use cases rather than imperative,
constraint-like requirements statements. They see consistency checking as checking
that the actions of use cases preserve invariants (i.e. constraints); however, at least in
the paper, they do not consider consistency (feasibility) of the invariants themselves.

The structure of the rest of this paper is as follows. In Sect. 2, we describe the
envisioned Requirements Traceability and Consistency checking Tool (RTCT) by
selected use cases. We discuss the most important and novel features in more detail and
provide an example to better understand key ideas. In Sect. 3, we comment on
available technologies for implementation. We conclude in Sect. 4.

2 RTCT: General System Description and Use Cases

2.1 General Goal Description
The general goal we want to address in our work is the improvement of tool support for
requirements engineering. The specific goal of this paper is to describe major
requirements for a requirements traceability and consistency checking tool (RTCT)
with a user interface, in the sequel referred to as RTCT-UI, which allows to
• Manage artifacts involved in the development of software or combined hardware-software systems and possibly residing on different platforms, of different types and at least including requirements, design and implementation artifacts, test cases, and test results;
• Manage traceability links between requirements and other artifacts (possibly including other requirements, especially in the case of decomposition or refinement, including formalization);
• Check whether all requirements pertaining to one product are mutually consistent; and
• Check whether each requirement is linked to all expected other artifacts according to a given requirements traceability model.

We will further discuss approaches of implementing such a tool (at least in prototypical form). The described tool may be considered as part of or add-on to a more comprehensive requirements engineering tool, or even as part of a comprehensive systems engineering tool. We will describe the major requirements in the form of use cases, rather than in the form of a formal specification, in order to provide a well readable impression which is not cluttered by technicalities.

2.2 General Constraints

The RTCT software shall be built under the following constraints:

• The Augmented Lifecycle Space approach shall be implemented (see [10]).
• Consequently, OSLC shall be employed to integrate different platforms and the OSLC standard, including its terminology, shall be used wherever applicable (see [OSCL, 18]).
• It shall be possible to introduce the new system incrementally, that is, only for new projects or new requirements for existing projects.

2.3 RTCT Use Cases

The following use cases informally describe the key features of the RTCT prototype to be implemented. Selected features and an example illustrating the rationale of the use cases will be discussed in more detail in subsequent sections. There are features of the system which are naturally needed and included for the sake of completeness. Those features, which determine the novelty value of RTCT with respect to the features of existing systems, are marked with a bold asterisk (*) at the beginning of the corresponding text.

U1. Create a new artifact collection. A user creates a new collection (project) with a custom name by means of RTCT-UI.

• The new collection is displayed in RTCT-UI.

U2. Add a new requirement to an empty collection. A user searches a requirement artifact within a given search space (possibly comprising different platforms) and adds it (as a link) to a selected, empty collection.
• Available meta-data are displayed (such as title, artifact type: Requirement, etc.). If relevant meta-data are missing, the user is prompted to enter them.
• Based on a given lifecycle model (e.g. the V-schema), RTCT determines which other artifacts are required to be linked to the new requirement (such as implementation artifacts to fulfill and test cases to test the requirement).
• For each required artifact, RTCT either creates a workflow for the creation of the artifact, or creates a stub artifact from a template, or creates a stub artifact plus a workflow for further work.

U3. Add a new requirement to a non-empty collection. A user adds a requirement artifact to a collection which already contains at least one other requirement artifact.
- As in (U2), with the following addition:
  * If the new requirement is formal, RTCT automatically checks its consistency with all other formal requirements in the same collection. If consistency cannot be proved, or an inconsistency is detected, RTCT issues a warning to the user.

U4. Add an artifact other than a requirement to a collection. A user adds an artifact other than a requirement to a selected collection which already contains at least one requirement artifact.
- Available meta-data are displayed (such as title, artifact type: Test case, etc.). If relevant meta-data are missing, the user is prompted to enter them.
- The user is asked to which existing requirement (or intermediate artifact) the new artifact should be linked. A respective bidirectional link is established after selection.

U5. * Check completeness of links. A user shall be able to check for a selected requirement, or for all requirements of a selected collection, whether all required links already exist. The check is triggered by a user and performed by RTCT.

U6. * Check consistency of requirements. A user shall be able to check for a selected collection whether all its formal requirements are mutually consistent. The check is triggered by a user and performed by RTCT.

- A copy of the selected requirement is made by RTCT for further work by the user, with a new unique requirement identifier (URI), and/or a workflow for creating/editing the new requirement is created.
- Alternatively, the user can add an existing requirement as the formalized version of the selected requirement.
- A new bidirectional traceability link of type “refines” (respectively “isRefinedBy”) is automatically generated between the selected requirement and the new (refining) requirement. The list of required traceability links is shifted from the refined requirement to the refining requirement, unless the refinement was required by the model used.
U8. * Decompose a requirement. A user selects a requirement to decompose.

- A user-specified number of copies of the selected requirement is made by RTCT for further work by the user, with new unique requirement identifiers (URIs), and/or a workflow for creating/editing the new requirements is created.
- Alternatively, the user can add existing requirements as decompositions of the selected requirement.
- New upstream bidirectional traceability links are automatically generated as in U7, for each new requirement, but of type “decomposes” (respectively “isDecomposedBy”). The list of required traceability links of the selected requirement is deleted (unless the decomposition was required by the model used) and new lists of required traceability links are created for each new requirement (further steps as in U2).
- Advanced (OPTIONAL): RTCT allows the user to search for existing blocks for reuse in the decomposition.


- As in U7, with the following differences:
  - A property “isFormal” of the new artifact is automatically set to TRUE.
  - The type of the upstream traceability link is automatically set to “formalizes”, the corresponding oncoming link to “isFormalizedBy”.
- When the new, formal requirement is committed, an automatic consistency check with existing formal requirements in the same collection is performed.

In the following, we will describe key and novel features contained in these use cases in more detail.

2.4 Traceability Link Completeness

In order to determine whether a requirement has been fully satisfied by implementation and testing, we need to know not only which design, implementation and testing artifacts (and possibly other artifacts) relate to the requirement in question, but also whether all artifacts we should expect to relate to this requirement actually exist and are linked to the requirement. [32] have presented empirical evidence that traceability completeness actually has a major impact on software quality.

Which traceability links to other artifacts we should expect, or actually require, can be determined from any appropriate model addressing traceability (see [10] for details). For instance, the version of the V-model applied in the Automotive SPICE standard (see Fig. 1 above) explicitly depicts required traceability links between artifacts associated with different project phases.

2.5 Stub Artifacts and Automatic Workflow Creation

Considering the model addressing traceability, it can be automatically determined which artifacts are necessary to satisfy a traceability requirement. In order to make sure that all of these artifacts are actually created and linked, we can offer the user at least
twofold support: First, we can create stub artifacts of the required types from templates, and secondly, we can create a workflow (or separate workflows for all required artifacts) which have to be allocated to appropriate development project members for completion. Both measures can be combined or applied alternatively. With traceability link completeness checking and automatic workflow generation for missing links, we implement the Augmented Lifecycle Space approach introduced in [10].

2.6 Requirements Consistency Checking

If requirements are formalized, we can move a step further in the early discovery of inconsistencies which would otherwise be usually detected only during design or implementation. It is well known that not only a major proportion of the deficiencies in software is introduced in the early project phases including requirements analysis, but that those deficiencies are typically the most costly to remediate. When such deficiencies are only detected during design or even implementation, then the requirements analysis phase and all subsequent lifecycle phases have to be at least partly reenacted, while elements of the already performed design and implementation phase may become obsolete. (Cf. [20] on the importance of requirements consistency.)

Formally modeled requirements within a project – or, more generally, any given collection – can be automatically checked for mutual logical consistency at least partially, provided they use a common data model, such as common variables. For instance, if we add a requirement which requires a certain pressure to be within a given window, but another requirement has already ruled out such a pressure, we must use the same pressure variable to detect this inconsistency. Standard verification can be performed e.g. with model checking or static analysis like satisfiability checking (SAT-checking). Complete models comprising all requirements in a more systematic way can be checked more comprehensively, using e.g. semi-automatic verification tools or more sophisticated model checking using custom temporal properties to be checked (see [20], Sect. 2.2.2 for some possible approaches; see also the example in Sect. 2.8 as well as Sect. 3 on implementation below).

Model checking may become difficult for large models, but may still be possible using appropriate model decomposition. Semi-automatic verification can take considerable effort, but may still pay off in many cases considering the potential cost of errors which are detected late or even after deployment. Note that others have considered heuristic, non-formal consistency checking, see e.g. [20] (Sect. 2.2.1); however, we do not consider this approach to be reliable and sufficiently automatable.

2.7 Decomposition, Refinement, and Formalization

A complication which is not fully visible in a model addressing traceability is the need for decomposition and refinement. We can see in Fig. 1 that stakeholder requirements must be refined to system requirements, for example. Stakeholder requirements are usually not precise enough, are incomplete (tacitly assuming domain knowledge, for instance), and may even be inconsistent, as they are often compiled from the desires of different stakeholders with different roles and professional and educational
backgrounds. Yet they form the basis for the system requirements, and thus the respective traceability links have to be established.

Beyond this refinement step, however, often more of such steps are required, as well as decomposition. Decomposition may be necessary, for instance, when comprehensive or general requirements will affect different parts or aspects of the desired system and/or will require different test cases, if overview would become lost during refinement of such a comprehensive requirement, or if consistency checking can be facilitated by decomposition.

Consider, for example, the following requirement: **Rx. RTCT-UI SHALL allow to search for and add new artifacts to collections and to add new relations between existing artifacts. New artifacts may exist on external data stores (repositories) and shall remain there, except from a local reference and possibly a local copy of metadata.** This requirement may be decomposed as follows:

- **Rx.1.** RTCT-UI SHALL allow to search for and add new artifacts to collections.
- **Rx.2.** RTCT-UI SHALL allow the addition of new relations between existing artifacts.
- **Rx.3.** If an artifact residing in an external repository is added to a collection in RTCT, then only a link SHALL be added to the collection as well as (OPTIONALLY) a local copy of metadata. The artifact itself SHALL remain in its original location and SHALL NOT be copied.

Formalization is a special case of refinement. The core of the result must be entirely formulated in a formal language suitable for automated verification (including model checking). A formal requirement must also be explicitly flagged as such in order to signal that it shall be checked for consistency with other formal requirements within the same collection (typically corresponding to a project). Whenever a new formal requirement is added to a collection, pairwise consistency checks shall be performed with all other formal requirements already in that collection.

Also certain types of formal requirements (especially model-based formalizations) may still be refined in order to allow for simulation for the sake of validation. Thereby already abstract, specification-level models can be interpreted and animated to allow domain experts assess whether the specified system behaviour meets their expectations. This way, many otherwise costly misunderstandings can be detected at an early stage.

### 2.8 An Example

We illustrate the important use cases by an example taken from [23] concerning the development of software for active medical devices. Let’s consider a hypothetical project, “Hemodialysis Safety System (HDSS)”. We assume an RTCT collection “HDSS” already exists and we have to increment it with the following requirement: **The software shall monitor the critical flow in the extra-corporeal circuit and if no flow is detected for more than 120 s then the software shall stop the critical flow pump and execute an alarm signal.**

We add an artifact containing this text to our collection. We are asked for metadata. Let us give a title: “Monitor critical flow”, artifact type: “Software requirement”, set flag “is-formal” to FALSE. Other metadata can be set automatically: A URI must
already exist anyway for RTCT so that it can access the artifact; a serial number (let’s say “R.HDSS.126”), can be automatically generated, while time, author, etc. are readily available to the tool.

Now RTCT will ask us whether this new artifact has an upstream dependency (e.g. was derived from another requirement). As this is a software requirement, it should depend on some system requirement (see the V-schema depicted in Fig. 1 above), so we select an existing system requirement, let’s say “R.HDSS.16”, and give its type by “refines/refined-by” (or, alternatively, “decomposes”/“decomposed-by”). By the act of selection, the new requirement replaces a stub software requirement which was automatically generated when system requirement R.HDSS.16 was added, and the respective workflow is marked as closed.

Next, RTCT determines that the new requirement requires two downstream links: one to a software qualification test case, and another to a software architecture artifact. The respective stub artifacts are created and bidirectional links are established; workflows are created. The resulting artifact dependencies are depicted in Fig. 2.

In order to check the consistency of the new requirement with existing requirements, it needs to be formalized. A formalization of this informal requirement in Event-B, as given in [23], consists of two different files (artifacts), called a context and a machine:
CONTEXT
C0
SETS
criticalFlowPumpingValues, Alarms
CONSTANTS
Start, Stop, noFlowMaxTime, ALM382, NULL
AXIOMS
tec1 partition (criticalFlowPumpingValues, {Start}, {Stop})
tec2 partition (Alarms, {ALM382}, {NULL})
typ1 noFlowMaxTime ∈ N
pro1 noFlowMaxTime = 120
END

MACHINE
M0
SEES
C0
VARIABLES
noFlowDetectionTime, alarm, criticalFlowPumping
INVARIANTS
inv1 noFlowDetectionTime ∈ N // Typing
inv2 noFlowDetectionTime > noFlowMaxTime ⇒ alarm = ALM382
/* If no flow is detected in 120s then the alarm should be executed */
inv3 alarm ∈ Alarms // Typing
inv4 criticalFlowPumping ∈ criticalFlowPumpingValues // Typing
EVENTS
Event INITIALISATION // Initialization values
Then
act1 noFlowDetectionTime := 0
act2 alarm := NULL
act3 criticalFlowPumping := Stop
End
Event stopCriticalFlowPumping // Stop critical flow pumping event
Where
grd noFlowDetectionTime > noFlowMaxTime ∧ criticalFlowPumping = Start
Then
act1 alarm := ALM382
act2 criticalFlowPumping := Stop // Stop critical flow pumping
End
Event startCriticalFlowPumping // Start critical flow pumping event
Where
grd criticalFlowPumping = Stop
Then
act1 criticalFlowPumping := Start
End
Event flowDetectionClock // The clock to simulate the time for flow detection
Where
grd noFlowDetectionTime < noFlowMaxTime ∧ criticalFlowPumping = Start
Then
act1 noFlowDetectionTime := noFlowDetectionTime + 1
End
END
(Note that in the machine above, the event “startCriticalFlowPumping” is not explicitly specified in the informal requirement; we will not pursue this issue here, however). We could either add these two artifacts to the collection separately or add the whole Event-B model (such as generated by the Rodin tool), e.g. in the form of a ZIP-file. We opt for the latter. So we add a new artifact: we select the URI where this ZIP-file can be found and add appropriate metadata; in particular, we set “is-formal” to TRUE. Let us call the new requirement “R.HDSS.127”. RTCT will ask us for an (optional) upstream dependency. Now we select R.HDSS.126 (the informal counterpart) and set the link type to “formalizes/formalized-by”.

Now two things should happen automatically: First, existing downstream links of parent R.HDSS.126 have to be rerouted to the new R.HDSS.127: now the only downstream link of R.HDSS.126 is to R.HDSS.127, of type “formalized-by”, and the previously generated links to a test case and an architecture artifact are now downstream links of the new R.HDSS.127. Figure 3 shows how the traceability graph has changed by this step.

![Artifact dependencies after adding a formalization of R.HDSS.126.](image-url)

Fig. 3. Artifact dependencies after adding a formalization of R.HDSS.126.

Secondly, as R.HDSS.127 is a formal requirement, RTCT will need to check its mutual consistency with all other formal software requirements which are already within the collection HDSS. The outcome of this check will be reported to us. If the system requirement R.HDSS.16 (the parent of R.HDSS.126) also has a formal counterpart, also consistency with this formal parent can be checked.
For instance, there might exist a requirement stating that “the critical flow pump MUST NOT be stopped during treatment”, obviously contradicting the new requirement that “if no flow is detected for more than 120 s then the software shall stop the critical flow pump …”. This consistency can only be detected if both requirements, in their formal versions, use the common variable “criticalFlowPumping” (plus a few other variables and constants).

Given that R.HDSS.127 is formal, a potential third step would be to automatically generate one or more test cases for it. There are tools which are capable of test case generation from formal models, e.g. Asmeta (see the next section). We could have proceeded differently, by the way. For instance, requirement R.HDSS.126 could have been decomposed before formalization (though this might seem rather artificial in this particular case). We could (in theory) decompose into monitoring, stopping the pump at a particular monitoring event, and executing an alarm at the same monitoring event.

In such a case, three new requirements would be added, each with links of type “decomposes” to R.HDSS.126. In this case, the existing downstream links of R.HDSS.126 as well as their targets would become invalid; instead, for each of the new requirements, new stub artifacts – test cases and architecture artifacts – together with the respective traceability links as well as workflows will have to be created. Figure 4 depicts the respective portion of the new traceability graph. Note that Test Case HDSS.1161 and Architecture artifact HDSS.126 have disappeared.

![Figure 4](image-url)
3 Implementation

As OSLC is based on web technologies, RTCT is best implemented as a web service. The core implements an OSLC client which can access different artifact repositories via OSLC adapters. The user interface of RTCT (RTCT-UI) is best implemented as a web interface using the RTCT service. There exists an OSLC reference Software Development Kit (SDK) called OSLC4J and provided by the project Eclipse Lyo [22]. This is explicitly recommended on the OSLC web page [25]. It is Java-based and builds on the Wink library for RESTful web services [35]. It provides a library for implementing both OSLC servers and clients.

The RTCT core must also be able to access resources like a lifecycle model and a verification tool (e.g. a model checker). A Java-based implementation platform is helpful for this purpose as there exist Java libraries for tasks like verification, but also e.g. for test case generation from abstract models. Examples are the tool Asmeta [4], which is based on the Abstract State Machine (ASM) method [11, 17], or the tool Rodin [33] for Event-B [1]. For various other tools, e.g. Python interfaces or even OSLC adapters exist.

Regarding formal modelling, there exist several implementations on the market which address parts of the issues targeted by the above use cases. One of the market leaders is Mathworks, with the related systems MatLab Simulink® and Polyspace®. Simulink models are not only implementations, but also abstractions of problems [15]. As Simulink models can usually not run on the target device, code has to be generated. Analysis can be performed in three different ways. First, the compliance with (corporate) guidelines can be checked with the Verification and Validation toolbox. This toolbox contains a rule base which complies with the most relevant standards (i.e. DO-178, ISO 26262, and IEC 61508). These rules contain design patterns corresponding to best development practices. By following and checking modelling guidelines, unsafe situations can be avoided and inherently safe design can be achieved (similarly to using MISRA rules in the programming language C).

For example, data flow can be restricted in a certain way to improve readability. If data flow is detected which does not comply with the restriction, a warning is given. Similarly, initial conditions have to be defined, and the lack of any initial condition again leads to a warning. This kind of analysis stays at the level of simple rule checks or pattern analysis, which is certainly useful but its capability is limited.

Second, the Simulink model can be interpreted as a formal description: the blocks have well defined (formal) descriptions together with the necessary boundary conditions. The Simulink Design Verification and Validation toolbox is capable of analyzing the model in such a manner and highlighting the weaknesses. It cannot only find certain errors but test cases can be generated with its help, where the use of formal methods guarantees the full test coverage at least at model level. Furthermore, dependencies can be highlighted, which helps identify the implementation artifacts corresponding to particular requirements.

As formerly mentioned, usually code is generated from the Simulink model. The generated code may fulfill certain standards [14] guaranteed by MathWorks (the manufacturer). Still, Polyspace can be utilized to check the generated code. This third
analysis again uses formal methods as a tool to find bugs which were inserted during code generation or were overlooked previously. These three steps provide an analysis as complete as can be expected, and they also support the documentation process of certain standards (mostly DO-178, ISO 26262 and IEC 61508). MathWorks also realized the importance of integration. Therefore, Simulink models can be linked directly to numerous tools used for requirements management. These tools include, among others, Rational DOORS. Yet it is important to highlight that Matlab uses a unique method via URIs to create the connection. Using OSLC compliant connections would further improve the usefulness of these well-build tool packages. There actually exist OSCI adapters for Simulink by third parties, see [6, 31], giving evidence for the interest for the integration of Simulink in tool chains.

In summary, the major building blocks necessary for creating the RTCT system exist, and have been independently developed in the fields of process capability improvement, formal modeling, and web technologies. The addressed challenge is the achievement of the collaboration of these building blocks, allowing the incremental improvement of the efficiency and effectiveness of safety-critical software development in practice, exploiting OSLC (Open Services for Lifecycle Collaboration).

4 Conclusion

We have presented major steps in realizing the Augmented Lifecycle Space approach for requirements management presented in [10] in a tool prototype which we call Requirements Traceability and Consistency checking Tool (RTCT). The approach considers cross-platform traceability exploiting OSLC, trace completeness checking based on a process model, and requirements consistency checking in order to considerably ease the requirements management process, to reduce major and costly defects in the requirements engineering phase, and to better support the verification of requirements satisfaction and thereby certifiability of the respective processes and products.

RTCT should not be seen as a standalone application, but as a kind of middleware providing particular services for engineering software that shall ultimately be integrated with other ALM-related software.

One of the important features of the new approach, of high importance for actual applications, is that it allows for the so-called “graceful integration” of formal modeling. Formal modeling is fundamentally necessary for securing completeness and consistency, but customarily rejected due to the usually prohibiting up-front effort needed to formally process all artifacts of an already established traditional system; Graceful integration can considerably lower this threshold.

In future work, amongst others, systems engineering aspects could be considered in RTCT, and RTCT could be integrated into a toolchain covering much of systems engineering activities. For instance, SysML requirements diagrams [34] could be used to depict the dependencies between requirements, and SysML associations can be used to depict traceability links between requirements and other artifacts.
Acknowledgement. The authors are grateful for the support of Research and Innovation Center of Óbuda University. The work is supported by the European Research Council Starting Grant ERC-StG 679681.

The research reported in this paper has been supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

References


Please ensure you fill out your response to the queries raised below and return this form along with your corrections.

Dear Author,

During the process of typesetting your chapter, the following queries have arisen. Please check your typeset proof carefully against the queries listed below and mark the necessary changes either directly on the proof/online grid or in the ‘Author’s response’ area provided below.

<table>
<thead>
<tr>
<th>Query Refs</th>
<th>Details Required</th>
<th>Author’s Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>AQ1</td>
<td>Please confirm if the corresponding author is correctly identified. Amend if necessary.</td>
<td></td>
</tr>
</tbody>
</table>