Supporting Requirements Validation: the EuRailCheck tool

R. Cavada, A. Cimatti, A. Mariotti, C. Mattarei, A. Micheli,
S. Mover, M. Pensallorto, M. Roveri, A. Susi, and S. Tonetta
Fondazione Bruno Kessler, IRST
Via Sommarive, 18, 38123 Povo - Trento, Italy
{*cavada,cimatti,mariotti,mattarei,amicheli,mover,pensallorto,roveri,susi,tonettas*}*@fbk.eu*

Abstract

We present the EuRailCheck tool, which supports the formalization and the validation of requirements, based on the use of formal methods. The tool allows the user to analyze the requirements in natural language and to categorize and structure them. It allows to formalize the requirements into a subset of UML enriched with static and temporal constraints for which we defined a formal semantics. Finally, the tool allows to apply model checking techniques specialized for the validation of formal requirements.

The tool has been developed and validated within a project funded by the European Railway Agency for the validation of the European Train Control System specification. By now, the tool has been successfully used by about thirty railway experts of different companies.

1. Motivations

EuRailCheck is a tool developed within a project funded by the European Railway Agency (ERA) for the formalization and the validation of the European Train Control System (ETCS) specification. ETCS defines a common train control system to be implemented in all European countries in order to guarantee an uninterrupted movement of trains across the borders. It is therefore of paramount importance to assure that the specification is consistent and is interpreted without ambiguity by different engineers in different countries. The main issues of this project are that: first, the ETCS documents are written in natural language, and thus may contain a high degree of ambiguity; second, in order to correctly interpret the specification, a deep knowledge in the railway domain is necessary, but is difficult to be combined with a background in formal methods. For these reasons, we developed a methodology and a supporting tool that maximize the usability for domain experts by: enriching the formal language used in the formalization phase with graphical notations and natural language expressions; building the tool on top of a standard engineering tool for software development, thus hiding the direct interaction with the verification engines.

S. Tonetta is supported by the Provincia Autonoma di Trento (project ANACONDA).

2. Tool supported methodology

Our methodology has been presented in [2]. It consists of three main steps:

- *Fragmentation and categorization.* Atomic fragments of the requirements document are identified, categorized and structured.
- *Formalization.* The categorized requirement fragments are formalized with a set of concepts and diagrams in UML, and through additional static and temporal constraints.
- *Formal validation.* It consists of the definition of a series of validation problems and the analysis of the results given by an automatic validation check.

Each step of the methodology is supported by a specific component of the EuRailCheck tool that is centered on the IBM Rational Software Architect (RSA) component and on the IBM Eclipse Platform the RSA application is built on. The developed Eclipse plug-ins allow connecting to other tools (MS Word, IBM RequisitePro and NuSMV or CEGAR model checkers) to complement the functionalities needed to support the whole specification/validation process. A view of the software components of EuRailCheck tool is given in Figure 1.

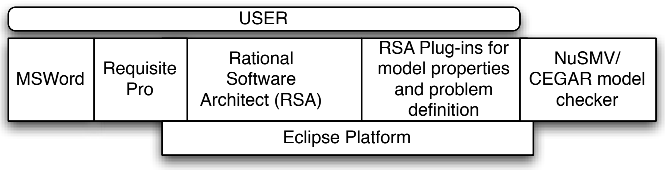
USER
MSWord
Requisite Pro
Rational Software Architect (RSA)
RSA Plug-ins for model properties and problem definition
NuSMV/ CEGAR model checker
Eclipse Platform

Figure 1. The EuRailCheck architecture.

2.1. Fragmentation and Categorization

2.1.1. Concepts. A *requirement fragment* is a part of the specification that identifies an atomic aspect of the system. This can be the name of a concept, a fragment of a sentence that contains a constraint on the system, a figure, a comment. Requirement fragments are grouped according to their purpose in the specification. Possible categories are:

Glossary, *Architecture*, *Behavioral*, *Functional*, *Scenario*, *Property*, *Annotation*.

The *Dependencies* between two requirement fragments A and B are: *Strong Dependency* links (A cannot exist without B); *Weak Dependency* links (A can exist without B). *Refinement* links (A redefines some notions of B at a lower level of abstraction). These dependencies are used in the formalization, to establish links among the formalized counterparts, and in the formal validation, to identify a well formed verification task.

2.1.2. Process. The steps for this informal categorization and analysis are:
M1.1 The user identifies and categorizes the informal requirement fragments.
M1.2 The user can create dependencies among the categorized requirement fragments.

2.1.3. Tool support. The tool supports the activity *M1.1* via the RequisitePro component that allows the user to choose the set of requirements under analysis directly on the documents, highlight the requirement fragments and categorize them according to the category taxonomy described before. The result is a database of categorized requirement fragments. Step *M1.2* is supported by a developed plug-in that allows the user to establish the dependencies between the categorized requirement fragments.

2.2. Formalization

2.2.1. Concepts: UML and CNL. The language used to formalize the requirements is a fragment of first-order temporal logic described in [3]. The formal language is enriched with a specification language that uses constructs of a semi-formal language to improve the usability. We have adopted concepts and diagrams from the UML language such as *classes*, *class diagrams*, *state machines*, *sequence diagrams* constructs. In order to specify static and temporal constraints over the entities in the model, we extended UML with a Controlled Natural Language (CNL). On the lines of [6], we use natural language expressions that can be mapped to temporal formulas. CNL grammar is based on the subset of the Property Specification Language [4], which is a standard in hardware design. It mixes linear temporal logic with regular expressions. A CNL constraint is attached to a class and can predicate over the attributes of that class and the associated classes. CNL constraints are classified in: *Initial*, defining constraints that are valid initially; *Invariant*, defining a constraint expected to be always valid over time; *Behavior*, defining a general constraint expressing admissible behaviors; *Scenario* describing behaviors that are expected to be admitted by the formalized requirement fragments; *Property*, defining constraints that every possible admissible behavior should satisfy (i.e., it defines a set of behaviors that are not admissible).

2.2.2. Process. The formalization phase consists of the following two activities.
M2.1 Formalization of each categorized requirement fragment identified in the fragmentation and categorization by specifying the corresponding UML concepts and diagrams, and/or the CNL constraints. In particular, *classes* and *class diagrams* to formalize the requirements that have been classified as Glossary; *state machines* to formalize requirements classified as Functional; *sequence diagrams* to represent those requirements classified as Scenarios that describe the interaction among a set of objects; CNL to specify the Behavioral, the Property requirements and the remaining Scenario requirements.
M2.2 Linking of the UML and CNL elements introduced to the textual requirements. The link is used for requirements traceability of the formalization against the informal textual requirements, and to select directly from the textual requirements document a categorized requirement fragment to validate.

2.2.3. Tool support. The activities are supported by the RSA component and the developed plug-ins. Here the user can refer to the database of the categorized requirement fragments to translate them into the corresponding set of UML concepts directly exploiting RSA's UML editor. Moreover, the tool provides the domain expert with an editor, equipped with a syntax checker, to specify the CNL constraints associated to the UML constructs. Once specified, the set of formal constructs can be linked to the categorized requirement fragments they refers to in a simple drag and drop manner. The result of the phase is the formalization of all the categorized requirement fragments and the specification of the information related to the traceability between each categorized requirement fragment and its formal counterpart.

2.3. Validation

2.3.1. Concepts: checks and diagnostic information. This phase aims at improving the quality of the requirements. This goal is achieved by performing several analysis, based on the use of formal techniques, which may help to pinpoint flaws that are not trivial to detect in an informal setting. The different analysis that are possible over the formalized requirements are:
Logical Consistency to formally verify the absence of logical contradictions in the considered formalized requirement fragments (e.g. the absence of two fragments mandating mutually incompatible behaviors).
Scenario compatibility to verify whether a scenario is admitted given the constraints imposed by the considered formalized requirement fragments.
Property checking to verify whether an expected property is implied by the considered formalized requirement fragments.

The above checks not only produce a yes/no answer, but they can also provide the domain expert with diagnostic information of different forms:
Traces. When consistency and scenario checking succeeds, it is possible to produce a trace witnessing the consistency, i.e. satisfying all the constraints in the considered formalized requirement fragments. Similarly, when a property check fails the tool provides a trace witnessing the violation of the property by the formalized requirement fragments.
Unsatisfiable core. If the specification is inconsistent or the scenario is incompatible, no behavior can be associated to the considered formalized requirement fragments; in these cases, the tool can also generate diagnostic information in the form of a minimal inconsistent subset. This information can be given to the domain expert, to support the identification and the fix of the flaw.

2.3.2. Process. The validation process consists of three main steps:
M3.1. The user chooses a set of requirements to focus the validation on particular aspects of the specification.
M3.2. The user defines a set of problems, each one consisting of a set of objects and a set of scenarios and properties.
M3.3. The user checks the defined problems and analyzes the results.

The above validation steps can be iterated arbitrarily, by correcting formalized requirement fragments and/or the corresponding categorized requirement fragments if necessary, creating new scenarios, new properties, and by analyzing different aspects of the requirements specification.

2.3.3. Tool support. The definition of the set of requirements of interests (*M3.1*) is supported by a plug-in that takes care of checking the completeness of the set with regards to the dependencies defined in *M1.2*. The user may define a problem with a special class whose attributes are considered the objects of the problems, and the attached constraints are considered the scenarios and the properties of the problem. Thank to the links with the formal elements created in *M2.2*, the selected informal requirement fragments correspond to a set of formalized requirement fragments. With this formal model and a given problem, the tool automatically translates the problem into an equi-satisfiable problem for the model checker: the problem admits a model if and only if the model checker finds a trace. The trace is mapped back to the tool and is visualized to the user. If the problem is unsatisfiable, the user may choose to look for an unsat core. This is presented to the user as a list of formalized requirement fragments that caused the inconsistency.

3. Conclusions

In this paper we described the EuRailCheck tool, which supports an end-to-end methodology for the analysis of requirements. The tool guarantees traceability, by allowing for a direct correspondence between the components of the informal specification and their formalized counterparts. The tool integrates, within a commercial environment, techniques for requirements management, for model-based design, and advanced techniques for formal validation. The tool allows the user to check consistency, entailment of required properties, and compatibility with desirable scenarios. The tool has been presented to and validated by about 30 potential users coming from 11 different railway organizations. The feedback was particularly positive by the people working on the ETCS specification and by those working for certifying bodies.

Although the tool has been validated in the railway domain, it has not been customized to this particular application domain. The language and the techniques are enough general to be applied to different domains. In particular, the language is suitable for the formalization of high-level requirements that constrain the interplay among objects, and their temporal evolution, even with real-time aspects.

In the future, we will try to improve the automation and the usability of the process. Currently, the first two phases of the methodology are the most demanding in terms of manual intervention. Works such as [5] and [1] aim at extracting automatically from a natural language description a formal model to be analyzed. Although, they are very limited in the expressiveness of the language and on the formal analysis of the requirements, our methodology can benefit from mature natural language processing techniques which are able to automatically dig out the ontology of the domain. On another line of activity, we will explore extensions to the expressiveness of the formalism, the relative scalability issues and optimization of the verification tools.

References

[1] V. Ambriola and V. Gervasi. On the Systematic Analysis of Natural Language Requirements with CIRCE. *Autom. Softw. Eng.*, 13(1):107–167, 2006.

[2] A. Cimatti, M. Roveri, A. Susi, and S. Tonetta. From Informal Requirements to Property-Driven Formal Validation. In *FMICS*, volume 5596 of *LNCS*, pages 166–181, 2008.

[3] A. Cimatti, M. Roveri, A. Susi, and S. Tonetta. Object models with temporal constraints. *Journal of Software and Systems Modeling (SoSyM)*, 8(4), 2009.

[4] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer-Verlag, 2006.

[5] A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. Assisting Requirement Formalization by Means of Natural Language Translation. *Formal Methods in System Design*, 4(3):243–263, 1994.

[6] R. Nelken and N. Francez. Automatic Translation of Natural Language System Specifications. In *CAV*, volume 1102 of *LNCS*, pages 360–361, 1996.