From Informal Requirements to Property-Driven Formal Validation

Alessandro Cimatti, Marco Roveri, Angelo Susi, and Stefano Tonetta

Fondazione Bruno Kessler
Istituto per la Ricerca Scientifica e Tecnologica
{cimatti,roveri,susi,tonettas}@fbk.eu

Abstract. Flaws in requirements may have severe impacts on the subsequent phases of the development flow. However, an effective validation of requirements can be considered a largely open problem.

In this paper, we propose a new methodology for requirements validation, based on the use of formal methods. The methodology consists of three main phases: first, an informal analysis is carried out, resulting in a structured version of the requirements, where each fragment is classified according to a fixed taxonomy. In the second phase, each fragment is then mapped onto a subset of UML, with a precise semantics, and enriched with static and temporal constraints. The third phase consists of the application of specialized formal analysis techniques, optimized to deal with properties (rather than with models).

1 Introduction

Most of the efforts in formal methods have historically been devoted to comparing a design against a set of requirements. The validation of the requirements themselves, however, has often been disregarded, and it can be considered a largely open problem, which poses several challenges. First, requirements are often written in natural language, and may thus contain a high degree of ambiguity. Despite the progresses in Natural Language Processing techniques, the task of understanding a set of requirements cannot be automatized, and must be carried out by domain experts, who are typically not familiar with formal languages. Second, the informal requirements often express global constraints on the system-to-be (e.g. mutual exclusion), and, in order to retain a direct connection with the informal requirements, the formalization cannot follow standard model-based approaches, but must be complemented with more suitable formalisms such as temporal logics. Third, the formal validation of requirements suffers from the lack of a clear correctness criterion (which in the case of design verification is basically given by the availability of high-level properties). Finally, the expressiveness of the language used in the formalization may go beyond the theoretical and/or practical capacity of state-of-the-art formal verification.

In this paper, we present a new methodology for the validation of requirements, that is based on formal methods. The main phases are the following. In the first phase, the informal requirements are split into basic fragments, which are classified into categories, and the dependency relationships among them are identified. During this informal inspection analysis, the natural language is disambiguated, and easy-to-detect flaws are

D. Cofer and A. Fantechi (Eds.): FMICS 2008, LNCS 5596, pp. 166–181, 2009.

discovered. In the second phase, each requirement fragment is formalized according to the categorization. The target formalism is a visual language such as UML, syntactically restricted in order to guarantee a formal semantics, and enriched with a highly-controlled natural language, to allow for expressing static and temporal constraints. In the third phase, an automatic formal analysis is carried out over the modeled requirements, using a number of advanced, complementary techniques. In particular, it is possible to carry out consistency checking, to verify whether some required properties are entailed, and whether the requirements are compatible with selected scenarios. Within this setting, diagnostic information is provided by means of traces, inconsistent cores, property vacuity, and scenario coverage.

The paper is structured as follows. In Section 2, we overview the methodology. In Section 3 and 4, we describe the informal analysis phase and the formalization phase. In Section 5, we present the procedures underlying the verification phase. In Section 6 we describe the support tools. In Section 7 and 8, we discuss the proposed methodology and the related work. Finally, in Section 9, we draw some conclusions and outline directions for future work.

2 Overview of the Methodology

We propose a novel methodology that addresses the issue of formalizing and validating a requirements specification written in an informal language.

Our approach builds on the use of the Unified Modeling Language (UML) to formalize the requirements, and on the use of a Controlled Natural Language (CNL) [30], based on a subset of the Property Specification Language (PSL) [21], to formalize the set of constraints on the requirements model. The set of UML concepts and artifacts we use in our approach represents a subset of the UML 2 concepts and diagrams described in the OMG UML 2 metamodel specification documents [1]. The subset of PSL our CNL builds on is the one that mixes Linear Time temporal Logic (LTL) [28] with Regular Expressions [4].

Our methodology consists of the the following three main steps:

M1 *Informal Analysis Phase.* It consists of the categorization and structuring of the informal requirement fragment described in the requirements document to produce categorized requirement fragments;

M2 *Formalization Phase.* The categorized requirement fragments are described trough the set of concepts and diagrams in UML, and additional constraints in the defined CNL to produce formalized requirement fragments;

M3 *Formal Validation Phase.* It consists of the identification of a subset of the formalized requirement fragments (together with the definition of a series of validation problems) for an automatic validation analysis.

Each step of the methodology is supported by a specific tool. In the following sections, we detail the methodology in terms of the associated sub-phases, artifacts, and modeling concepts. The categories of requirement fragments are detailed together with the steps for the analysis and structuring of the requirements. The methodology is presented using as running example a simple elevator control system specification.

Table 1. Requirement fragments categories

Requirement fragments conditions	Requirement fragment category
Does the requirement fragment define a particular concept in the domain?	Glossary
Does the requirement fragment introduce some system's modules and describe how they interact?	Architecture
Does the requirement fragment describe the steps a particular module performs or the states where the module can be?	Functional
Does the requirement fragment describe the messages some modules exchange?	Communication
Does the requirement fragment describe some constraints on the behaviors of system-to-be?	Behavioral
Does the requirement fragment describe some constraints on the environment?	Environmental
Does the requirement fragment describe a possible scenario of the domain?	Scenario
Does the requirement fragment describe an expected property of the domain?	Property
Is the requirement fragment a note in the specifications that does not add any information about the ontology or the behavior of the specified system?	Annotation

3 Informal Analysis Phase

The first activity in the methodology is the informal analysis of the set of requirements. In this phase the requirements are first categorized on the basis of their characteristics. Then, some dependencies among them are established to structure the categorized requirement fragments. The steps for this informal categorization and analysis are:

M1.1 Isolation of the fragments that identify a requirement unit of the domain requirements document.
M1.2 Categorization of the informal requirement fragments.
M1.3 Creation of the dependencies among the informal requirement fragments.
M1.4 Analysis of the informal requirement fragments based on standard inspection-based software engineering in order to identify flaws such as, e.g., recursive definitions.

The final result of the informal analysis phase is a database of categorized requirement fragments.

Requirement fragment categorization. We have identified nine possible categories: *Glossary*, *Architecture*, *Functional*, *Communication*, *Behavioral*, *Environmental*, *Scenario*, *Property*, *Annotation*. The conditions specified in Table 1 define the corresponding category to be assigned to each informal requirement fragment.

The categorization helps the analyst in understanding the domain and can also be used in the next steps of the methodology to guide the formalization by suggesting the use of particular UML or CNL constructs.

Dependencies definition. We have identified the following three kind of dependencies to describe the possible relationships among two requirement fragments A and B:

Table 2. Example of identified requirement fragments

R0	Elevator
R1.1	Call buttons to choose a floor
R1.2	Floor
R1.3	Key switches
R1.4	Call buttons could be key switches
R1.5	Certain floors are inaccessible unless using the key
R2.1	Elevator Door
R2.2	Door open buttons
R2.3	Door close buttons
R2.4	[Door close button] instructs the elevator [door] to close immediately
R2.5	[Door open button] instructs the elevator [door] to remain open longer

- *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 phase, to establish links among the formalized counterparts, and in the formal validation phase, to identify a well formed verification task.

3.1 Example of Informal Analysis

We start analyzing the specification of the elevator described at the URL `http://en.wikipedia.org/wiki/Elevator#Controlling_elevators`. We perform the step M1.1 of the methodology to get a set of requirement fragments. It is reported that: *A typical modern passenger elevator will have:*

1. *Call buttons to choose a floor. Some of these may be key switches: certain floors are inaccessible unless using the key.*
2. *Door open and door close buttons to instruct the elevator to close immediately or remain open longer.*

The first sentence introduces the concept *Elevator*. The first item introduces the concepts *Floor*, *Button*, *Call button* and *Key*. The functionality *choose a floor* is associated to concept *Call button*. Moreover, the concept *Key* is associated to the *Call button* by saying that *certain floors are inaccessible unless using the key*. The second item in the specification introduces three new concepts: the presence of an elevator *door*, the *door open and door close buttons* with their functionalities *to instruct the elevator to close immediately or remain open longer*. We can then define the requirement fragments as in Table 2.

In the step M1.2 we classify R1.{1-4} and R2.{1-3} as Glossary requirement fragments. The R1.5 and R2.{4-5} can be classified as a Behavioral requirement fragments (they describe constraints on the system-to-be). R1.5 can also be classified Scenario and Property.

In the step M1.3 we recognize a Strong Dependency of the requirement R1.5 to the requirements R1.2 and R1.3.

4 Formalization Phase

Our methodology requires proceeding with the formalization of the categorized requirement fragment version of the requirements produced as artifact of the informal analysis phase. The formalization phase consists of the following sub-activities:

M2.1 Formalize each requirement fragment identified in the informal analysis phase by specifying the corresponding UML concepts and diagrams, and/or the CNL constraints.

M2.2 Link the UML elements introduced in M2.1 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.

We have adopted: *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 Environmental, the Property requirements and the remaining Scenario requirements. The selection of UML diagrams and concepts has been performed on the basis of the expressive power of the UML concepts and on the need related to the formalization of the UML constructs in a formal language. (See [17] for details on the underlying object model extended with temporal constraints.)

Class diagrams. A *class* represents a concept in the domain. In our context, a class is associated with: a set of class attributes representing the set of characteristics of the concept; a set of class methods, representing actions/procedures the class can perform. A method accepts a set of parameters in input and has a return parameter. Each attribute and each method parameter has a type that can be primitive, e.g. $Integer$ or $Real$, or user defined, i.e., $enumerative$ or a class defined in the UML model.

Relationships among classes represent the relations existing between domain concepts. In our context we allow only for the following relationships:

- *Association*: it is the basic relationship that can be established among two classes.
- *Aggregation*: it specifies that the class belongs to a collection (another class).
- *Generalization*: it indicates that one class is a "superclass" of the other.

Association and Aggregation relationships are characterized by their multiplicity. This multiplicity represents the range of the number of instances of the involved classes that exist in the domain ($0..1$ zero or one instance, $0..*$ or $*$ no limit on the number of instances, 1 exactly one instance, 1..* at least one instance, and $n..m$ n to m instances).

State Machines. We use *state machines* to model the behavior of each method of a class in the domain. In our framework we restricted the syntax of the UML 2 state machines as follows. We only allow for the following kinds of UML 2 state machine states: *initial state*, that represents the entry point of the corresponding class method; *final state*, that represents the return point of the corresponding class method; "simple" *state*,

that represents a generic state of the corresponding class method; *conditional states*, that represent conditional branches in the execution of the corresponding method. A *transition* of a state machine represents the conditions that determine the change of state. Each transition is associated with a label of the form $event[guard]/activity$. The meaning of the label is that the transition is performed when the $event$ occurs and the $guard$ (that is a boolean predicate) is true. When the transition is fired the specified $activity$ is performed. In our framework, the events are restricted to be only class method calls, while the activity have to be a parallel combination of class method calls and class attribute assignments.

Sequence Diagrams. UML *sequence diagrams* model the evolution of the specified objects as a sequence of exchange of messages, focusing on the representation of their interactions. We use sequence diagrams to model Scenarios requirements that are requested/expected to happen in the domain in terms of exchange of messages. The UML 2 notation for a sequence diagram is restricted as follows: we use *objects* to represent the instances of the classes involved in a given interaction; *lifelines* to represent the lifetime of the objects involved in the interaction; *messages* are restricted to be only a method call performed by an object to a method of another object. We also allow for the specification of some *interaction operators* defined in UML 2 that specify particular configurations of messages, such as: the *negation*, the *alternative*, the *option*, the *parallel* and the *loop*.

Controlled Natural Language. In order to allow for the specification of constraints and temporal properties of the entities in the model, we extend the UML model with a constraint language. The constraint language is a Controlled Natural Language (CNL) [30], i.e., a well-defined subset of natural language whose grammar has been restricted in order to be automatically processable. The language includes temporal operators as in [25]. We proposed a CNL grammar, based on the subset of the PSL [21] logic that mixes Linear Temporal Logic (LTL) [28] operators with Regular Expressions [4]. This choice is motivated by the fact that this fragment, being linear time, is adapt to express constraints on the evolution of observable events of the system. The grammar has been defined to include enough syntactic sugar to be easily accepted and then used by non-experts in computer science or software engineering.

We have classified the CNL constraints we use to annotate the UML concepts and diagrams in the following five categories.

- *Initial*: it defines constraints that are valid initially.
- *Invariant*: it defines a constraint expected to be always valid over time.
- *Behavior*: it defines a constraint expressing admissible behaviors.
- *Scenario*: it describes behaviors that are expected to be admitted by the formalized requirement fragments.
- *Property*: it defines behaviors that every possible admissible behavior should satisfy (conversely, it defines a set of behaviors that are not admissible).

Example of formalization. Figure 1 represents the class diagram resulting from the formalization of the output of the informal analysis phase for the elevator. For requirements R1.{1-3} we defined the classes *Button*, *Call_Button*, *Floor*

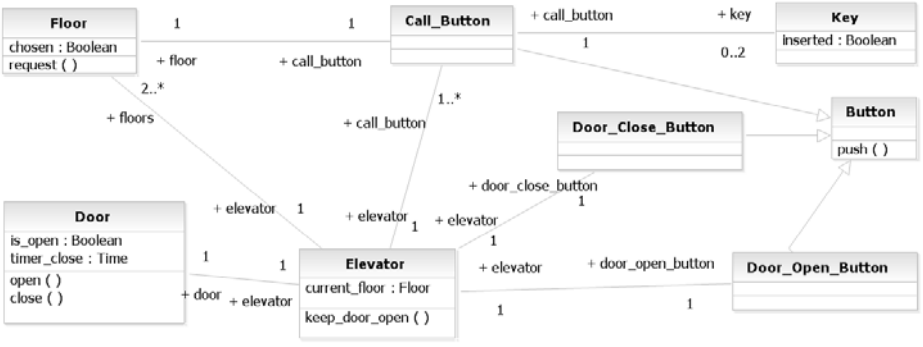

Fig. 1. The elevator class model

and *Key*, *Door_Open_Button*, *Door_Close_Button*, and *Elevator*. *Door_Open_Button*, *Door_Close_Button* and *Call_Button* are refinement of class *Button*. Requirement R1.4 is captured in the diagram by the association relationship between the two classes *Call_Button* and *Key* where is imposed, via the cardinalities at the two sides of the relationship, that for every *Call_Button* there could be 0 or at most 1 *Key* associated. Class *Floor* has an attribute *chosen* (from R1.1) of type Boolean to indicate whether the corresponding floor has been requested. Class *Key* has attribute *inserted*, of type Boolean, to indicate whether the key has been inserted. Class *Door* as attribute *is_open*, of type Boolean, to indicate whether the door is open or closed. Class *Door* has methods *close* and *open*, from requirement fragments R2.4 and R2.5 respectively, to model the activity of opening/closing the corresponding door. Class *Button* has method *push* to model the press of the button.

The analysis of the categorized requirement fragments for the elevator lead to the definition of some CNL constraints. Below we report an excerpt of the constraints classified Behavior (CB.#) necessary to formalize the elevator.

CB.1 *for all Call_Button b, whenever b.push(), b.floor.request()*
CB.2 *for all Call_Button b, if b has a key, whenever b.floor.request(), b.key.inserted*
CB.3 *for all Door_Close_Button b, whenever b.push(), b.elevator.door.close()*

Requirement R1.1 leads to the introduction of the constraint CB.1 to say that pushing the button related to a floor means to request that floor. Similarly, from R1.5 it is possible to extract the CNL constraint CB.2, that states that all call buttons that have a key are such that if the button has been pressed, then the key has been inserted. R2.4 states that when the door close button is pushed then the elevator door has to close. This is formalized with by the CNL constraint CB.3. From R1.5, it is possibile to obtain also the following constraints of type scenario (CS.#) and property (CP.#) respectively:

CS.1 *there exists a Call_Button b such that b has a key and in the future b.floor.request()*
CP.1 *for all Call_Button b, b has a key and never b.key.inserted implies never b.floor.request()*

Following the same process, other constraints have been imposed on the model on the basis of the specification. They have not been reported here for the sake of space. A more detailed description can be found at `http://es.fbk.eu/people/roveri/tests/fmics08`.

Phase M2.2 consists in linking all the elements in the model to the textual requirements fragments. For example, the requirement R1.1 will be directly related to the class *Call_Button* via a data structure that maintains this information (such as for instance the traceability structures provided by standard tools like IBM Rational RequisitPro and IBM Rational Software Architect).

5 Formal Validation Phase

The validation of the formalized requirement fragments aims at improving the quality of the requirements. This goal is achieved by performing several analysis steps, based on the use of formal techniques, that may help to pinpoint flaws that are not trivial to detect in an informal setting.

These steps include checks to identify inconsistencies, and to increase the confidence that the categorized requirement fragment and its corresponding formalized counterpart meet the design intent: for instance, a flaw may be in the fact that some desired behaviors have been ruled out by an over-constraining set of requirements; conversely, some undesired behavior may have not been ruled out by under-constraining requirements.

The formal validation phase of the methodology will be accomplished as follows:

M3.1 *Check the well-formedness of the formalized requirement fragments.* This initial activity aims at verifying that the formalized requirement fragments syntactically adhere to the formal language syntax, and that all the elements mentioned have been previously defined.

M3.2 *Narrowing of the formalized requirement fragments.* This phase aims at focusing the validation to a particular subset of interest of the formalized requirement fragments (e.g. to restrict the validation of the classes/functions of a specific module). In this phase the validation expert selects a set of objects per each class.

M3.3 *Formal validation of the identified formalized requirement fragments.* The subset of interest identified in M3.2 is formally analyzed to identify flaws if any.

Whenever a problem is identified in any of the above sub-phases, in order to try and solve the identified flaw, it may be required to go back to a previous phase. We remark that, in this phase, the domain expert responsible of the validation can specify additional desired and undesired behaviors w.r.t. the ones already formalized in previous phases, in order to guarantee that the design intents are captured, thus further enriching the formalized requirement fragment.

The phase M3.3 can be further decomposed depending on the scope and on the level of domain knowledge required to perform it. For this purpose we classify the validation checks in *Domain Independent* and *Domain Dependent* checks. There is a third kind of checks, aiming at further analyzing the *quality* of the results produced by the domain dependent checks e.g. by performing vacuity analysis, coverage analysis and safety analysis.

Domain Independent Checks. These checks aim at verifying properties of the formalized requirement fragment that do not require any domain knowledge, i.e. *logical consistency* and *realizability*.

Checking Logical Consistency. The formal notion of logical consistency can be intuitively explained as "freedom from contradictions". It is possible that two formalized requirement fragments mandate mutually incompatible behaviors. This check aims at formally verifying the absence of logical contradictions in the considered formalized requirement fragments. Consistency checking is carried out by dedicated formal verification algorithms [16].

Checking Realizability. Realizability [29,13] intuitively amounts to checking if there exists an open system implementing the considered formalized requirement fragments. The variables occurring in the considered formalized requirement fragments are classified as either controllable (by the specified system), or uncontrollable (depending on the environment). Moreover, the considered formalized requirement fragments are partitioned in two distinct sets, the formalized requirement fragments representing "assumptions" on the behavior of the uncontrollable variables, and the formalized requirement fragments representing the "guarantee", that must be enforced on the controlled variables. The check consists in verifying the existence of an open system whose controllable variables obey the guarantee for all possible behaviors of the uncontrollable signals obeying the assumptions. Realizability is substantially more informative than satisfiability, but also computationally more expensive [29]. Realizability checking is carried out by dedicated state of the art algorithms for checking and debugging realizability [15].

Providing Diagnostic Information. The checks for logical consistency and for realizability not only produce a yes/no answer, but they can also provide the validation expert with diagnostic information of different forms. For instance, when consistency 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, as outcome of the realizability check, it is possible to generate a witness of realizability, that in this case has the form of a Finite State Machine satisfying the considered formalized requirement fragments. We notice that, if the specification is inconsistent, no behavior can be associated to the considered formalized requirement fragments; similarly, when it is not realizable, then no Finite State Machine can be associated. In these cases, the verification algorithms can also generate diagnostic information. For consistency check, this has the form of a small un-satisfiable subset of the considered formalized requirement fragment [16], while for unrealizability check, an un-realizable [15] subset is identified. This information can be given to the domain expert, to support the identification and the fix of the flaw. The formalized requirement fragments can be traced back to the corresponding categorized requirement fragment, and up to the original requirements in order to remove the identified flaw.

The fact that a given formalized requirement fragments is not consistent can be traced back to a misinterpretation in the formalization of the corresponding categorized requirement fragments. In this case, the subset of the considered formalized requirement fragments produced as diagnostic information needs to be revised to remove the ambiguity

that led to the misinterpretation of the original requirements. A possible explanation for the un-realizability can also be traced back to a missing assumption on the environment. In this case, the fix consists in revising the whole set of requirements to add the missing assumptions.

Domain Dependent Checks. These checks aim at verifying that the considered set of formalized requirement fragments really captures the design intent. In this case, the formalized requirement fragments are validated against descriptions of desired and undesired behaviors identified by the domain expert. Desired behaviors are used to ensure that the considered formalized requirement fragments are not too strict, and that they have not been ruled out (*scenario compatibility*). Dually, undesired behaviours are used to ensure that the considered formalized requirement fragments are not too weak, and that they have indeed been ruled out (*property checking*).

Scenario compatibility. This check aims at verifying whether a set of conditions (also called a scenario) is possible, given the constraints imposed by the considered formalized requirement fragments. Intuitively, the check for scenario compatibility can be seen as a form of simulation guided by a set of constraints. The behaviors used in this phase can be partial, in order to describe a wide class of compatible behaviors.

The check for scenario compatibility can be reduced to the problem of checking the consistency of the set of considered formalized requirement fragments with the constraint describing the scenario. Thus, if the scenario is compatible, we obtain a behavior trace compatible with both the considered formalized requirement fragments and with the constraint describing the scenarios. Otherwise, we obtain a subset of the considered formalized requirement fragments that prevents the scenario to happen.

Property checking. This check aims at verifying whether an expected property is implied by the considered formalized requirement fragments. This check is similar in spirit to Model Checking [19], where a property is checked against a model. Here the considered set of formalized requirement fragment plays the role of the model against which the property must be verified. When the property is not implied by the specification, a counterexample is produced. A counterexample is a behavior witnessing the violation of the property, i.e. a trace that is compatible with the considered formalized requirement fragment, but does not satisfy the property being analyzed.

Property checking can be reduced to the problem of checking the consistency of the considered formalized requirement fragments with the negation of the property. If this set is consistent, then a witness behavior compatible with the considered formalized requirement fragment and satisfying the negation of the property is produced. This behavior is a counterexample for the property. If such witness does not exist then the property holds.

If the verification of the property fails, two causes are possible: the first one is that the property is not correctly formalized; the second possibility is in a wrong formalization of the informal sentences in the categorized requirement fragment that need to be disambiguated and/or corrected. An inspection of the counterexample can be carried out in order to discriminate among the two possibilities. If the property is wrong, then it is corrected and the check is repeated. Otherwise, the formalized requirement fragments

has to be corrected, either by modifying the formalization or by adding additional constraints, until the satisfaction of the given property is achieved.

Quality of the results of Formal Validation. The previous analyses can produce diagnostic information in several forms (witness/counterexample behaviors). It is worth noticing that, the fact that the formalized requirement fragment is consistent or that a property holds can be due to some under-specification in the considered formalized requirement fragment or in the property itself. Moreover, if a property fails, there can be several reasons that can cause the failure. Thus, before starting to fix the formalized requirement fragment it would be useful to identify all the causes of the flaw.

The first problem is tackled by performing what we called *vacuity checking* and *coverage checking*, while the second is tackled by *safety analysis*.

Vacuity checking. Corresponds to checking whether a given property holds vacuosly [7]. For instance, consider the property "whenever the signal A is received, a corresponding signal B must be issued". If the formalized requirement fragment is such that the signal A can never be received, then the property trivially holds (the pre-condition of the implication is not satisfiable), and is thus not informative. Vacuity is typically considered to be a flaw in a specification [7] due to missing or redundant constraints.

Coverage checking. Corresponds to checking which elements of the considered formalized requirement fragment have been stimulated (covered) by a generated trace. This check plays for scenario checking and consistency the role that vacuity plays for properties. Suppose the validation generates a trace such that a certain signal A is never issued, and that the considered formalized requirement fragments (possibly together with a property scenario) for which this trace has been generated is mandating that "whenever a signal A is received, a corresponding signal B must be issued". The trace "trivially" satisfies the considered set of requirements, but it does not stimulate the consequence of the mandating property (which is what the domain expert is interested to see), thus the trace is not informative. The fact that a generated trace is not informative, is not a flaw per se, but it can indicate that for instance the assumptions on the environment are under-specified or that the scenario is under-specified.

Formal safety analysis. It aims to identify all the causes leading to the violation of an expected property. The domain expert can identify the variables of interest that are to be considered causes of a specific violation, and advanced algorithms [6,8,9] can then be used to gather a description of the causes, and to organize them in form of a fault tree.

Validation Loop. 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. The narrowing phase M3.2 allows the domain experts to focus only on a subset of the formalized requirement fragments by selecting specific modules and consider only some of the functions of the selected module thus enabling for a *modular validation* approach. It also allows performing several kinds of *what-if* analysis, in particular, it allows checking which properties and scenarios remain valid after adding/removing new formalized requirement fragments. Moreover, in

the narrowing phase we can ignore the requirements with low-level details and consider the requirements at a higher level of abstraction, thus enabling for a hierarchical verification approach. This process results in a validation loop where every check increases the confidence of the domain expert in the correctness of the formalized requirement fragments.

Example of validation. We applied the proposed validation loop to the Elevator example. We selected the formalized requirement fragments described in Section 4. In the narrowing phase (M3.2) we identified the following set of objects: one *Elevator*, four *Floors*, four *Call_Buttons*, one *Door*, one *Door_Open_Button*, one *Door_Close_Button*, and one *Key*.

We first checked for the consistency of the formalized requirement fragments. We automatically translated the class diagram and the constraints into the input language of the model checker NuSMV [14]. The tool provided us with a witness of the example's consistency consisting of a loop over the initial state. This trace described the case where the elevator is initially at the fourth floor with the door open and nothing happens.

We then verified if the model is compatible with a scenario where the elevator is initially at the first floor, there is a request to go to the third floor, and the elevator goes to the third floor. The tool provided us a trace witnessing the compatibility with such scenario: the produced trace is such that it loops over requesting both first and third floor at the same time, going to the third floor and then going back to the first floor.

Finally, we verified the scenario CS.1 and the property CP.1. The formalized requirement fragments results compatible also with CS.1 and produces a trace where all buttons have a key, and only the first and the fourth become requested. This trace seems to contradict the assumption that we have only one key, but the point is that we did not force the buttons not to share their keys. After adding this new assumption, we get a new trace where only the second button has a key, and all floors become requested. Finally, the model checker proved that there are no counterexamples with length less than 40 time steps for property CP.1.

6 Overview of the Support Tools

Our methodology is supported by a tool chain we developed on top of standard-de-facto industrial tools.

We used IBM Rational RequisitePro (RRP), interfaced with Microsoft Word, and IBM Rational Software Architect (RSA), to support the informal analysis phase and the tracebility of the link between the informal requirement fragments and their formal counterparts. We used RSA interfaced with RRP and with the validation tool to support the formalization phase. The developed interface allows mapping the formal model into the input language of the validation tools. Moreover, it maps back the verification results as to use them within RSA back to RRP to correct the possible flaws identified during the validation phase.

The validation tool has been built on top of an extended version of the state-of-the-art NuSMV [14] verification tool. This extension provides advanced techniques to compile LTL and PSL properties into automata [18], and advanced abstraction based

verification techniques [12], exploiting the MathSAT [10] SMT solver, to efficiently deal with infinite-state components.

7 Discussion of the Approach

We discuss how the proposed methodology addresses the challenges of requirement validation we consider most relevant for a successful adoption of formal methods in the design flow of complex safety-critical systems.

Choice of a formalization language. The proposed methodology provides a fully formal language. Every statement is associated with a formalized counterpart, that is given unambiguous semantics. Nevertheless, the language can be used by the domain experts because it exploits the usability of graphical languages such as UML and the closeness of CNL to Natural Language. This way, we try to maximize the usability and expressiveness of the language. At the same time, we provide the automatic techniques for the formal analysis.

Ambiguity of natural language. The proposed methodology addresses the key problem that the informal requirement fragments are ambiguous and unstructured as follows: the informal requirement fragments are structured and categorized by means of an informal requirement analysis; every informal requirement fragment is linked with a precise set of elements in the formal model; this way, if the validation phase detects some bugs, the domain expert can easily distinguish if they are due to a wrong formalization or to the ambiguity of the informal requirement fragment.

Incompleteness of requirements validation. New verification techniques and tools have been developed to overcome the inadequacy of traditional tools for model checking and design verification to validate requirements. The analysis is no longer directed on a design; rather, the properties themselves become the object of the analysis. It is possible to check whether the specification is strict enough, by checking whether undesired behaviors have been indeed eliminated. Technically, this problem is reduced to checking whether the expected property is a logical consequence of the set of requirements. Conversely, it is possible to check if the specification is not too strict, by checking whether desirable behaviors have not been eliminated. This approach to requirements analysis is described in [27] and in [23]; the RAT (Requirements Analysis Tool) has been developed [15] to this end.

Quality of the validation feedback. The formal validation phase of our methodology tries to maximize the information the validation tools can produce in order to help the domain expert to correct the specification or the formalization: it produces traces animating the requirements; it can enable the diagnosis of inconsistencies by identifying inconsistent cores; it can identify vacuous properties and uncovered requirements; it can enable the formal safety analysis by performing Fault-Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA) using emerging techniques [6,9].

Complexity of the specification. The proposed methodology tackles the problem of the specification complexity by providing a mixed property/model-based approach. This allows handling and analyze complex system of specifications: each requirement is

encoded in a distinct piece of formalism; the methodology supports incremental and modular approaches to the validation; and, it enables for analysis at different level of abstraction. Most importantly, the property-based approach allows verifying the specification without the need of describing the model of the implementation. The approach is ideal for early validation when the requirements must be verified before the implementation of the system.

8 Related Work

The problem of formalizing and analyzing a requirement specification is one of the main challenges in Requirements Engineering.

Works such as [22] and [5] aim at extracting automatically from a natural language description a formal model to be analyzed. However, on one hand, their target formal languages cannot express temporal constraints over object models; on the other hand, they miss a methodology for an adequate formal analysis of the requirements. Nevertheless, our methodology can benefit from mature natural language processing techniques which are able to automatically dig out the ontology of the domain.

Several formal specification languages such as Z [31], Object-Z [11], VDM [2], B [3], and OCL [26] have been proposed for formal model-based specification. However, all of them are not adapt for the use by requirements analysts and domain experts. They are very expressive but require a deep background in order to write a correct formalization, they lack of completely automatic proof support tools, and the use of these tools requires deep knowledge of them in order to use them efficiently. Moreover, these languages have been designed for particular applications, and their usage for different purposes may become awkward and difficult. For instance, they are unable to express complex temporal constraints like, e.g., fairness.

Formal Tropos (FT) [32] and KAOS [20] are goal-oriented software development methodologies that provide a visual modelling language that can be used to define an informal specification. The visual modeling language is supported with annotations that characterize the valid behaviors of the model, expressed in a typed first-order linear time temporal logic (LTL). The main differences between the proposed approach and FT and KAOS are in the expressiveness of the formalization language: both FT and KAOS are limited to pure LTL and they are hardly committed to the goal representation of the requirements.

In [24], a framework is proposed for the automated checking of requirement specifications expressed in Software Cost Reduction tabular notation, which aims at detecting specification problems such as type errors, missing cases, circular definitions and nondeterminism. Although this work has many related points to our approach, the proposed language is not adapt to formalize requirements that contain functional descriptions of the system at high level of abstraction with temporal assumptions on the environment.

9 Conclusions

In this paper we have presented a methodology for the validation of a requirements specification. The methodology first envisages an informal analysis of the requirements

document to categorize each requirement. In the second phase, each requirement fragment is formalized according to the categorization by means of UML diagrams and the use of a Controlled Natural Language as to facilitate the use by non experts in formal methods. In the third phase, automatic formal analysis is carried out to identify possible flaws in the formalized requirements. The methodology is supported by a chain of tools built on top of standard-de-facto industrial tools (like e.g. Rational RequisitePro and Software Architect), and on an extended version of the NuSMV model checker.

The methodology and the related tools are currently under evaluation in a real-world project that aims at formalizing and validating the European Train Control System (ETCS) specification. The project is in response to the European Railway Agency tender ERA/2007/ERTMS/OP/01 ("Feasibility study for the formal specification of ETCS functions"), awarded to a consortium composed by RINA SpA, Fondazione Bruno Kessler, and Dr. Graband and Partner GmbH (see `http://es.fbk.eu/events/formal-etcs/` for further information on the project). The documents under consideration contain a huge set of requirements, that are intended to guarantee the interoperability between trackside railway systems and trains throughout Europe. This consortium is currenlty applying the methodology, and carrying out a training activity for domain experts. A detailed reporting of the results of the project is the object of future activities.

Acknowledgments. We are very grateful to the European Railway Agency for issuing the challenge of the ETCS formalization and validation. We thank A. Chiappini (ERA) for his continuous encouragement and support. We thank F. Caruso, L. Macchi, and B. Vittorini from RINA Spa, for their precious feedback after applying the methodology and using the tools. P. Zurek and A. Schulz-Klingner from Dr. Graband & Partner GmbH are also thanked for useful discussions. Finally, we thank the Provincia Autonoma di Trento for supporting S. Tonetta (project ANACONDA).

References

1. UML Version 2.1.2., `http://www.omg.org/spec/UML/2.1.2/`
2. Bjorner, D., Jones, C.B. (eds.): The Vienna Development Method: The Meta-Language, vol. 61. Springer, Heidelberg (1978)
3. Abrial, J.R.: The B-book: Assigning Programs to Meanings. C.U. Press, Cambridge (1996)
4. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers - Principles, techniques and tools. Addison-Wesley, Reading (1986)
5. Ambriola, V., Gervasi, V.: On the Systematic Analysis of Natural Language Requirements with CIRCE. Autom. Softw. Eng. 13(1), 107–167 (2006)
6. Banach, R., Bozzano, M.: Retrenchment, and the generation of fault trees for static, dynamic and cyclic systems. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 127–141. Springer, Heidelberg (2006)
7. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in temporal model checking. Formal Methods in System Design 18(2), 141–163 (2001)
8. Bertoli, P., Bozzano, M., Cimatti, A.: Symbolic model checking framework for safety analysis, diagnosis, and synthesis. In: Edelkamp, S., Lomuscio, A. (eds.) MoChArt IV. LNCS, vol. 4428, pp. 1–18. Springer, Heidelberg (2007)
9. Bozzano, M., Villafiorita, A.: The FSAP/NuSMV-SA Safety Analysis Platform. STTT 9(1), 5–24 (2007)

10. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4 SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
11. Carrington, D.A., Duke, D.J., Duke, R., King, P., Rose, G.A., Smith, G.: Object-Z: An Object-Oriented Extension to Z. In: FORTE 1989, Amsterdam (NL), pp. 281–296 (1990)
12. Cavada, R., Cimatti, A., Franzén, A., Kalyanasundaram, K., Roveri, M., Shyamasundar, R.K.: Computing Predicate Abstractions by Integrating BDDs and SMT Solvers. In: FM-CAD, pp. 69–76 (2007)
13. Church, A.: Logic, arithmetic and automata. In: Proc. 1962 Int. Congr. Math., Upsala, pp. 23–25 (1963)
14. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model checker. STTT 2(4), 410–425 (2000)
15. Cimatti, A., Roveri, M., Schuppan, V., Tchaltsev, A.: Diagnostic information for realizability. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 52–67. Springer, Heidelberg (2008)
16. Cimatti, A., Roveri, M., Schuppan, V., Tonetta, S.: Boolean Abstraction for Temporal Logic Satisfiability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 532–546. Springer, Heidelberg (2007)
17. Cimatti, A., Roveri, M., Susi, A., Tonetta, S.: Object models with temporal constraints. In: SEFM (2008) (to appear)
18. Cimatti, A., Roveri, M., Tonetta, S.: Syntactic Optimizations for PSL Verification. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 505–518. Springer, Heidelberg (2007)
19. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
20. Darimont, R., Delor, E., Massonet, P., van Lamsweerde, A.: GRAIL/KAOS: an environment for goal-driven requirements engineering. In: ICSE 1997, pp. 612–613. ACM, New York (1997)
21. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Springer, Heidelberg (2006)
22. Fantechi, A., Gnesi, S., Ristori, G., Carenini, M., Vanocchi, M., Moreschini, P.: Assisting Requirement Formalization by Means of Natural Language Translation. Formal Methods in System Design 4(3), 243–263 (1994)
23. Fuxman, A., Liu, L., Mylopoulos, J., Pistore, M., Roveri, M., Traverso, P.: Specifying and analyzing early requirements in Tropos. Requirements Engineering 9(2), 132–150 (2004)
24. Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated consistency checking of requirements specifications. ACM Trans. Softw. Eng. Methodol. 5(3), 231–261 (1996)
25. Nelken, R., Francez, N.: Automatic Translation of Natural Language System Specifications. In: CAV, pp. 360–371 (1996)
26. OMG. Object Constraint Language: OMG available specification Version 2.0 (2006)
27. Pill, I., Semprini, S., Cavada, R., Roveri, M., Bloem, R., Cimatti, A.: Formal analysis of hardware requirements. In: DAC 2006, pp. 821–826 (2006)
28. Pnueli, A.: The temporal logic of programs. In: Proceedings of 18th IEEE Symp. on Foundation of Computer Science, pp. 46–57 (1977)
29. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: 16th Annual ACM Symposium on Principles of Programming Languages, pp. 179–190 (1989)
30. Schwitter, R.: Dynamic Semantics for a Controlled Natural Language. In: DEXA Workshops, pp. 43–47 (2004)
31. Spivey, J.M.: The Z Notation: a reference manual, 2nd edn. Prentice Hall, Englewood Cliffs (1992)
32. Susi, A., Perini, A., Giorgini, P., Mylopoulos, J.: The Tropos Metamodel and its Use. Informatica 29(4), 401–408 (2005)