Validation of Requirements for Hybrid Systems: a Formal Approach

Article *in* ACM Transactions on Software Engineering and Methodology · November 2012

DOI: 10.1145/2377656.2377659

4 authors, including:

Alessandro Cimatti
Fondazione Bruno Kessler
402 PUBLICATIONS **18,951** CITATIONS

SEE PROFILE

Marco Roveri
University of Trento
195 PUBLICATIONS **9,453** CITATIONS

SEE PROFILE

Stefano Tonetta
Fondazione Bruno Kessler
132 PUBLICATIONS **2,943** CITATIONS

SEE PROFILE

Validation of Requirements for Hybrid Systems: a Formal Approach

ALESSANDRO CIMATTI, MARCO ROVERI, ANGELO SUSI, STEFANO TONETTA
FBK-irst, Center for Information Technology

Flaws in requirements may have unacceptable consequences in the development of safety-critical applications. Formal approaches may help with a deep analysis which takes care of the precise semantics of the requirements. However, the proposed solutions often disregard the problem of integrating the formalization with the analysis, and the underlying logical framework lacks either expressive power, or automation.

We propose a new, comprehensive approach for the validation of functional requirements of hybrid systems, where discrete components and continuous components are tightly intertwined. The proposed solution allows to tackle problems of conversion from informal to formal, traceability, automation, user acceptance, and scalability.

We build on a new language, OTHELLO, which is expressive enough to represent various domains of interest, yet allowing efficient procedures for checking the satisfiability. Around this, we propose a structured methodology where: informal requirements are fragmented and categorized according to their role; each fragment is formalized based on its category; specialized formal analysis techniques, optimized for requirements analysis, are finally applied.

The approach was the basis of an industrial project aiming at the validation of the European Train Control System (ETCS) requirements specification. During the project a realistic subset of the ETCS specification was formalized and analyzed. The approach was positively assessed by domain experts.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies*

General Terms: Languages, Verification

Additional Key Words and Phrases: Requirements validation, Formal languages, Methodology, Safety-critical applications, European Train Control System.

1. INTRODUCTION

Requirements engineering is a fundamental phase of the development process of software systems [Sommerville 2004; van Lamsweerde 2009]. Mature methodologies guide the engineers through the elicitation, analysis, and validation of the requirements.

Flaws in requirements may have unacceptable consequences in the development of safety-critical applications (e.g., avionics, railways, space). Missing and erroneous requirements are often associated with safety-related errors in the final system implementation. Missing checks on data inputs or wrong values in conditions and limits risk to trigger an erroneous response or to cause a failure in triggering the response in time. The cause of these safety-related functional errors in most cases can be traced to issues of the requirements specification [Lutz 1993].

Authors' address: Fondazione Bruno Kessler
Center for Information Technology
Via Sommarive, 18, 38123 Trento-Povo, Italy
S. Tonetta was partly supported by the Provincia Autonoma di Trento (project ANACONDA).

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

The *requirements validation* phase is devoted to pinpointing such issues. Unfortunately, the standard practice in requirements validation consists of a manual, syntactic effort, and often ignores the semantics of the requirements. Thus, it provides rather shallow analysis capabilities, and may miss subtle flaws resulting from complex interactions among requirements.

In safety critical domains, the systems under specification are often *hybrid systems*, where controlling components interact with the physical environment through sensors and actuators. In this setting, the requirements validation is even more difficult because the mathematical model underlying assumptions on the physical entities is very complex and may lead to subtle errors.

The need for more powerful techniques for requirements validation in safety-critical domains is being increasingly recognized. For instance, the European Railway Agency (ERA) issued a call for tender [EuRailCheck-cft 2007] for the development of techniques able to carry out a thorough validation of a body of the European Train Control System (ETCS) requirements specification, which defines the interoperability protocols of the forthcoming European railway network.

In order to provide deeper analysis capabilities, able to pinpoint flaws due to complex interactions over temporal behaviors, a possible way to go is to complement traditional techniques with a *formal approach*: the requirements are mapped into a logical formalism, with a clearly defined semantics, so that validation can be reduced to logical reasoning. Taking this direction to a practical level, however, presents a number of challenges.

First, the requirements are often written in natural language and they cannot be automatically formalized. Thus, the requirements engineers have to be both expert of the applicative domain, and able to use formal methods. For this reason, the adopted formal language must be as close as possible to the natural language used to write the requirements. This demands for a logical language very expressive both from the syntax and from the semantic point of view.

Second, the formalism must be highly expressive. To model requirements that are typical of safety-critical domains, the language must capture the possible configurations and their temporal evolutions, the physical constraints on the environment, such as the battery's charging pattern, or the assumptions on the data read from an odometer or an accelerometer.

Third, a suitable notion of quality for requirements is missing. In fact, the standard target of formal verification is the compliance of a design against a set of requirements which are considered as "golden", and says little on how to formalize the concept of "correct requirements". Apart from consistency, it is not obvious which formal checks can be performed to validate the requirements.

Finally, it is necessary to achieve a suitable degree of automation and scalability of the formal reasoning process. This is however very hard, due to the required expressiveness of the formalism, and also because the majority of the formal verification tools and techniques have been developed in a setting where the complexity lies in the system being verified (and not in the properties to verify on the design).

Contribution. In this paper, we propose a formal approach for the validation of functional requirements of safety-critical systems, based on three key ingredients. At the heart of the approach, we propose a new *formal language*, OTHELLO (Object Temporal with Hybrid Expressions Linear-time LOgic), whose expressiveness has been tailored to represent functional requirements in hybrid domains. OTHELLO combines a first-order temporal logic with objects and hybrid constructs to represent continuous real-time aspects of physical entities, and is made user friendly with the use of graphical notation and natural language expressions.

Efficient *reasoning techniques*, based on symbolic model checking [Clarke et al. 1999] and on Satisfiability Modulo Theories (SMT) [Sebastiani 2007], provide effective formal verification solutions that can be used to analyze the logic.

The approach is organized in a *structured methodology*, which guides the formalization of the informal requirements, and provides suitable formal checks. The methodology consists of three main phases. In the first phase, the informal requirements are decomposed into basic fragments, which are classified into categories, and the dependency relationships among them are identified. In the second phase, each requirement fragment is formalized according to its categorization. In the third phase, an automatic formal analysis is carried out over the modeled requirements, using a number of advanced, complementary techniques. In addition to consistency checking, the formal analysis allows to check whether the requirements are too strict (i.e., they are ruling out desired scenarios) or too weak (i.e., they admit undesirable scenarios), and provides suitable diagnostic information to explain the results.

Features of the approach. The approach explicitly aims at ensuring *usability* by domain experts, who have little background on formal methods. For this, the logical language and framework were designed balancing a mixture of techniques:

— *Graphical notation*. The notation is based on the class diagrams of the Unified Modelling Language [Rumbaugh et al. 2010], which allows the user to introduce concepts and their relationships.
— *Natural language*. Inspired by the work in [Nelken and Francez 1996] and by PSL [Eisner and Fisman 2006], we used English expressions instead of mathematical notation to easily bridge the gap between the textual specification and its formal counterpart.
— *Property-based approach.* As in [PROSYD 2007], we based the formalization on a logical formalism, which is more oriented to a natural formalization of requirements because of the resemblance between formal and informal requirements. The property-based approach allows to have a one-to-one mapping between an informal requirement and its formal counterpart, thus enabling the *traceability* of the formalization and of the validation.
— *First-order temporal logic.* We used a rich temporal logic, which is necessary to faithfully capture the semantics of the requirements, as already recognized by many previous works such as [Ghezzi et al. 1990; Bois et al. 1997].
— *Rich validation checks.* The proposed methodology provides a set of checks that are tailored to requirements validation (generalizing new formal techniques for requirements analysis that have emerged in the context of hardware design [PROSYD 2007; Pill et al. 2006]).
— *Automation and scalability of the analysis.* We adapted infinite-state model checking techniques in order to solve validation checks for the formalized requirements. We leverage recent trends in formal verification which build on the success of SMT solvers.
— *Diagnostic information.* In order to analyze the validation results, and understand better the semantics of the requirements, a yes/no answer is not sufficient. The approach provides the user with traces which animate the requirements and with unsatisfiable cores which identify a subset of requirements at the cause of some problem (e.g., inconsistency).

Clearly, some of the above features are inspired by previous works (see also Section 9 for a detailed comparison with related work). Here, we highlight the following points of novelty:

— *Expressiveness of the language.* The OTHELLO language is more expressive than similar formalisms proposed for the requirements formalization. Existing languages are too limited to represent requirements of safety-critical applications, because they do not capture both the continuous evolution of physical entities over time and the discrete changes of actions.
— *Automation of the analysis.* Despite its expressiveness, the OTHELLO language allows for an automatic analysis based on infinite-state model checking. Other approaches either use interactive theorem provers, or are limited to model checking in the finite-state case, thus trading in the expressiveness of the language.
— *Guided formalization.* The methodology includes a phase where informal requirements are fragmented and categorized according to a predefined schema. This step drives the requirements engineer in the formalization process, which is perhaps the most critical step for someone who is not an expert in formal methods. The categorization allows to break down the activity into sub-tasks, and to suggest, in a simple "cook book" style, how to formalize fragments according to their categorization. The effectiveness of this step has been confirmed by the users of the methodology in the experimental evaluation.

Practical Application. The methodology was applied in the EuRailCheck [Chiappini et al. 2010] project funded by ERA for the formalization and the validation of the ETCS requirements specification. Within the project, we developed a tool chain to support the proposed methodology. This was then deployed in an industrial setting, and used to formalize a realistic subset of the specification. The results of the project were then disseminated in a subsequent training phase, where 22 domain experts external to the consortium positively evaluated the proposed approach.

Outline. The paper is structured as follows. In Section 2, we describe the motivating domain of ETCS. In Section 3, we describe the language OTHELLO. In Section 4, we overview the methodology. In the subsequent three sections we describe in detail each of the phases of the methodology: categorization (Section 5), formalization (Section 6), and formal validation (Section 7). In Section 8, we present the achievements obtained during the project funded by ERA. In Section 9, we provide a detailed technical comparison of our approach with respect to related works. Finally, in Section 10, we draw some conclusions and outline directions for future work.

2. ETCS: A MOTIVATING APPLICATION DOMAIN

In this section, we introduce the ETCS case study, which is paradigmatic for the kind of safety critical systems. The ETCS is an initiative supported by the European Union, aiming at the implementation of a common train control system in all European countries. The ETCS requirements specification is a 300-pages document, written in natural language and related to the automatic supervision of the location and speed performed by the train on-board system. The system is intended to be progressively installed on all European trains in order to guarantee the interoperability with the trackside system across European borders.

In the rest of the paper, we will use the ETCS requirements specification as a running example to explain the methodology, the underlying logical formalism, and the formal validation checks. Specifically, we will refer to the requirements 3.8.1.1.{a-c} of the System Requirements Specification (SRS) [ETCS 2006] of the ETCS. These requirements define the concept of Movement Authority (MA), which is a set of information regarding the authorized location and speed of a train. In particular, an MA is given an End of Authority (EOA), which is the target location to which the train is authorized to move, a Target Speed, which is the speed at which the train is authorized to run passing over the EOA, and a Danger Point, which is a location beyond the

3.8.1 Characteristics of a MA

3.8.1.1 The following characteristics can be used in a Movement Authority (see Figure 17: Structure of an MA):

a) The End Of Authority (EOA) is the location to which the train is authorised to move.

b) The Target Speed at the EOA is the speed permitted at the EOA; when the target speed is not zero, the EOA is called the Limit of Authority (LOA). This target speed can be time limited.

c) If no overlap exists, the Danger Point is a location beyond the EOA that can be reached by the front end of the train without a risk for a hazardous situation.

Fig. 1. ETCS in Microsoft Word®.

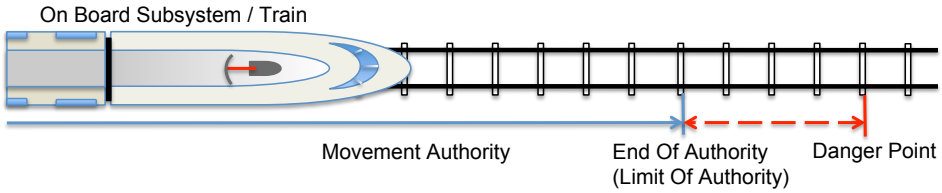

Fig. 2. The railway scenario described in the snapshot of the requirements document; a train authorized, via a Movement Authority, to run to the given End of Authority and the corresponding Danger Point.

EOA which should never be passed by the train. Figure 1 is a snapshot of Microsoft Word® document showing the considered requirements while Figure 2 is a sketch of real railway scenario described in the document.

3. THE SPECIFICATION LANGUAGE OTHELLO

In this section we describe OTHELLO, the language used to formalize the requirements. The name OTHELLO stands for "Object Temporal with Hybrid Expressions Linear-time LOgic", and it indicates the three main ingredients of the language, i.e., objects, temporal, and hybrid aspects. It integrates, in one language, the temporal logic with hybrid aspects defined in [Cimatti et al. 2009] and the object-oriented temporal logic introduced in [Cimatti et al. 2011]. We first present the intuitive underlying model, the syntax, the formal semantics, and discuss algorithms for reasoning with OTHELLO specifications.

3.1. Underlying hybrid model

The system specified by the requirements we are considering (hereafter, referred to as *system-to-be*) consists of a controller interacting with the physical environment. The underlying model is given by hybrid systems [Henzinger 1996], which combines a discrete component representing the control part (including for example concepts such as the current MA of a train or the current EOA of a MA) and a continuous component representing the physical part (consisting of the physical attributes such as the current location and speed of a train). An evolution of the system is represented by a sequence of discrete and continuous transitions. Discrete transitions are characterized by *instantaneous* changes of the systems involving control switches and events such as the reception of a new MA and changes to the continuous variables such as the reset of a timer. Continuous transitions are characterized by the *elapsing of time* which

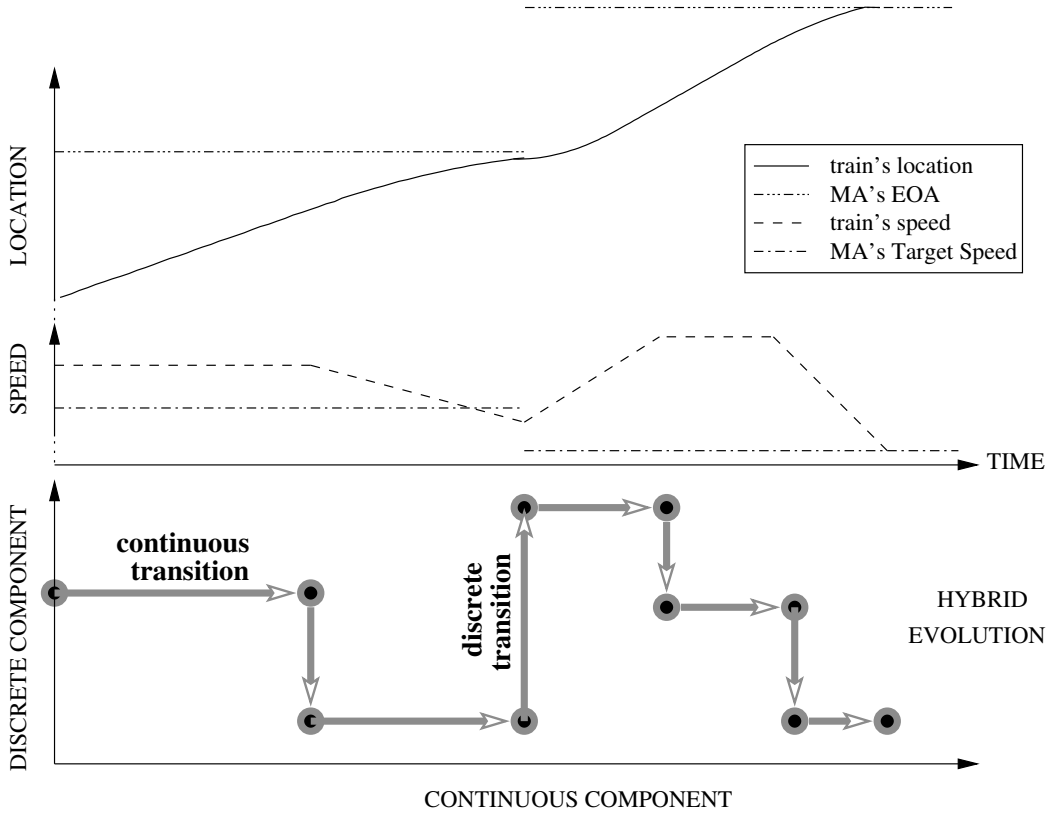

Fig. 3. Example of hybrid evolution. The location and the speed of the train evolve continuously along time. The EOA and the Target Speed of the MA instead change instantaneously.

makes the continuous component evolve according to its dynamics constraints, while the discrete component remains unchanged.

An example of such evolution is depicted in Figure 3: the location and the speed of the train are continuous variables and thus evolve with a continuous function, while the EOA and the Target Speed of the MA are discrete variables that change instantaneously, and thus evolve with a step function. In the figure, the train starts from an initial location running with constant speed and alternating continuous and discrete transitions (see lower part of the figure) it moves to a different location; it passes to deceleration mode; then, it decelerates while approaching the first EOA until the speed is lower than the corresponding target speed; then, the train receives a new MA with a farther EOA, and passes to acceleration mode; it moves on, passes to constant speed, moves on, passes to deceleration mode, and finally reaches the second EOA.

3.2. Syntax

An OTHELLO specification is a pair $\mathcal{O} = \langle \mathcal{C}, \mathcal{TC} \rangle$ where $\mathcal{C}$ is a class diagram, to define the classes of objects specified by the requirements, their relationships and their attributes, and $\mathcal{TC}$ is a set of constraints, to specify the temporal evolution of the possible configurations (see the example in Section 6.3). The core of the constraint language is a fragment of first-order temporal logic whose signature is defined by the class diagram.

The temporal structure of the logic encompasses the classical linear-time temporal operators [Pnueli 1977] combined with regular expressions [Aho et al. 1986]. This combination is well established in the context of digital circuits and forms the core of standard languages such as the Property Specification Language (PSL) [Eisner and Fisman 2006].

We enriched the logic with constructs able to specify hybrid aspects of the objects' attributes such as derivatives of the continuous variables and instantaneous changes of the discrete variables.

On the lines of PSL, we also provide a number of syntactic sugar which increases the usability of the language by the domain experts, and we use natural language expressions to substitute the Boolean and temporal operators, and the quantifiers.

3.2.1. Classes and Relationships. A *class* of $\mathcal{C}$ represents a concept in the domain. In our context, a class is associated with: a set of class *attributes* representing the characteristics of the concept; a set of class *methods*, representing actions/procedures associated to that concept. A method accepts a set of parameters in input and has a return parameter. Each attribute and each method parameter has a type. The types we allow in our framework are: *Boolean*, *Integer*, *Real* and *Continuous* as primitives types; an *enumerative* or a class, as user defined types. All variables apart from *Continuous* ones are referred to as *discrete variables* (where "discrete" does not refer to the domain of the variables, but to the domain of the time used to interpret their temporal evolution). Attributes have a *multiplicity* whose value is an interval of non-negative integers. When the multiplicity is different from $1..1$, the attribute is an array of elements of the attribute's type. The size of the array can vary, but must remain within the range defined by the multiplicity.

Attributes of class type represent a *relationship* between the two domain concepts represented by the type of the attribute and the class the attribute belongs to. A different relationship between two classes is the *generalization*, which can be used to indicate that one class is a "super-class" of the other.

3.2.2. Temporal constraints. The constraints in $\mathcal{TC}$ predicate over the attributes of the classes and the attributes of the associated classes via the classical dot notation. Associations and aggregations are treated as attributes. Generalizations have the effect that all attributes inherited from the super-classes are available as attributes in the sub-classes.

The grammar for the OTHELLO constraint language is summarized in Table I. A constraint consists of a set of atomic formulas connected with standard Boolean and temporal operators, which are expressed with English words.

The formulas can be quantified. Quantified formulas are in the form "**for all/there exists** scope formula" where the "scope" declares the bound variable and a restriction on its domain. This can be given by a class, or an identifier of type array, or a bounded integer interval.

The identifiers are the basic terms of the logic and are built on top of the variables by applying either an attribute or a method with the standard dot notation. The atomic formulas are arithmetic predicates over terms. The "**next**" operator can be used to refer to the value of a variable after a discrete change, while the "**der**" operator can be used to refer to the first derivative of continuous variables during a continuous evolution.

The constraints where neither temporal operators (**always**, **never**, **in the future**, **until**, **sequence matching**) nor methods are used are called *static constraints*.

3.3. Semantics

Table I. The OTHELLO constraints language grammar.

constraint	:=	*atom* |
		not *constraint* |
		constraint **and** *constraint* |
		constraint **or** *constraint* |
		constraint **implies** *constraint* |
		always *constraint* |
		never *constraint* |
		in the future *constraint* |
		constraint **until** *constraint* |
		a sequence matching *regexp* |
		a sequence matching *regexp* **followed by** *constraint* |
		any sequence matching *regexp* **triggers** *constraint* |
		for all *scope* **(** *constraint* **)** |
		there exists *scope* **(** *constraint* **)** |
		there exists *scope* **such that (** *constraint* **)** ;
scope	:=	variable **of type** *class* |
		variable **in** *identifier* |
		variable **in** *term* **..** *term* ;
atom	:=	**true** |
		false |
		term = *term* |
		term < *term* |
		term > *term* |
		term <= *term* |
		term >= *term* |
		term |
		term . method **()** |
		term . method **(** *parameters* **)** |
		term . method **()** **returns** *term* ;
term	:=	*identifier* |
		constant |
		term + *term* |
		term − *term* |
		term ∗ *term* |
		term / *term* |
		der(*identifier* **)** |
		next(*identifier* **)** ;
regexp	:=	{ *atom* }
		regexp ; *regexp*
		regexp | *regexp*
		regexp && *regexp*
		regexp [∗]
identifier	:=	variable |
		identifier **.** attribute |
		identifier **[** *term* **]** |
		identifier **.size** ;

3.3.1. Configurations and behaviors. A *configuration* represents formally the state of the system in a certain time instant. Given a class diagram, a *configuration* defines:

— for every class c a set U_c of *objects*, called the universe of that class;
— for every class c, for every object o in the universe U_c of c, for every attribute a of c, if the multiplicity of a is $1..1$, a value $[\![o.a]\!]$; otherwise an Integer $[\![o.a.size]\!]$ in the range defined by the multiplicity and, for every Integer i between 1 and $[\![o.a.size]\!]$, a value $[\![o.a[i]]\!]$; the values $[\![o.a]\!]$ and $[\![o.a[i]]\!]$ must belong to the domain defined by the attribute's type; thus, if the type is a class, the value is an object of the universe of that class; if the type is Boolean, Integer, or Real, the value is a Boolean, Integer, or Real value; if the type is Continuous, the value is a Real continuous and piecewise-differentiable function;

— for each method of every class, two Boolean values, which represent respectively the call and return events associated to that method. Thus, a method can be instantaneous or last the amount of time elapsed between the two events. The configuration also defines a value for the input parameter and for the returned value.

A *behavior* represents formally the evolution of the system-to-be. It consists of an infinite sequence of configurations and an infinite sequence of Real intervals that represent the time line: the intervals can be open or singular (time points), must be consecutive, and cover the whole Real non-negative time line. Two consecutive configurations of a behavior form a *step*. The succession of the same time point is allowed and is called a *discrete step*. If the i-th interval is open, then the $i-1$-th, the i-th, and the $i+1$-th configurations assign the same value to all attributes of all objects. In particular, in the case of a continuous attribute, the functions f_{i-1}, f_i, and f_{i+1} are equal so that if the open interval is (t, t'), $f_{i-1}(t) = f_i(t)$ and $f_i(t') = f_{i+i}(t')$.

3.3.2. OTHELLO *specifications models.* A behavior satisfies an OTHELLO specification if it satisfies every temporal constraint of the specification.

A constraint is interpreted over a behavior and over an assignment to the free variables that occur in the formula.

In the following, we will denote a behavior with ω, the i-th configuration of ω with ω^i, the interpretation of a term x in ω^i given the variables assignment μ with $[\![x]\!]_{\langle\omega^i,\mu\rangle}$.

The definition of the interpretation follows the recursive definition of the syntax and refers to the index of the step of the behavior, i.e., the position in the sequence of the behavior from which the constraint is interpreted. A behavior satisfies a constraint iff the constraint is true in the first step of the behavior. In the following, if not clear from the context, we refer explicitly or implicitly to the i-th step of the behavior.

As described in Section 3.3.1, a configuration gives an interpretation to all symbols of the class diagram. An identifier is interpreted at the step i according to the values defined by the i-th configuration of the behavior. In particular, if v is a variable, $[\![v]\!]_{\langle\omega^i,\mu\rangle} := \mu(v)$; if x is an identifier of a class C and a is an attribute of the class C, $[\![x.a]\!]_{\langle\omega^i,\mu\rangle}$ is the value assigned by ω^i to the attribute a of the object $[\![x]\!]_{\langle\omega^i,\mu\rangle}$; similarly for $x.a[j]$ and $x.a.size$.

Operations over Integer and Real terms are interpreted according to the standard semantics of the arithmetic operations: if x and y are Integer or Real terms and $\star \in \{+, -, *, /\}$, then $[\![x \star y]\!]_{\langle\omega^i,\mu\rangle} = [\![x]\!]_{\langle\omega^i,\mu\rangle} \star [\![y]\!]_{\langle\omega^i,\mu\rangle}$. This is extended to continuous terms, whose interpretation are Real functions: if x and y are continuous terms and $\star \in \{+, -, *, /\}$, then, for all Real t, $[\![x \star y]\!]_{\langle\omega^i,\mu\rangle}(t) = [\![x]\!]_{\langle\omega^i,\mu\rangle}(t) \star [\![y]\!]_{\langle\omega^i,\mu\rangle}(t)$.

The term "**next(***identifier***)**" is interpreted with the values defined by the $i+1$-th configuration: $[\![next(x)]\!]_{\langle\omega^i,\mu\rangle} = [\![x]\!]_{\langle\omega^{i+1},\mu\rangle}$.

The term "**der(***identifier***)**" is interpreted with the derivative of the function associated to the identifier by the i-th configuration: if x is a continuous term, for all Real t, $[\![der(x)]\!]_{\langle\omega^i,\mu\rangle}(t) = \frac{d}{dt}[\![x]\!]_{\langle\omega^i,\mu\rangle}(t)$.

Atoms (which are the basic elements of a constraint) have a Boolean value, i.e., *true* or *false*. Arithmetic predicates have standard semantics: if x and y are Integer or Real terms and $\bowtie \in \{=, <, >, <=, >=, != \}$, then $[\![x \bowtie y]\!]_{\langle\omega^i,\mu\rangle}$ holds iff $[\![x]\!]_{\langle\omega^i,\mu\rangle} \bowtie [\![y]\!]_{\langle\omega^i,\mu\rangle}$ holds. This is extended to continuous terms: if x and y are continuous terms and $\bowtie \in \{=, <, >, <=, >=, != \}$, then $[\![x \bowtie y]\!]_{\langle\omega^i,\mu\rangle}(t)$ holds iff, for all Real t, $[\![x]\!]_{\langle\omega^i,\mu\rangle}(t) \bowtie [\![y]\!]_{\langle\omega^i,\mu\rangle}(t)$ holds. An atomic formula is interpreted as the arithmetic predicate provided that, if the atom contains "**next**" then the i-th step must be discrete: if the atom does not contain "**next**", then the atom is true at the i-th step iff the predicate holds in the i-th configuration; otherwise, the atom is true at the i-th step iff the predicate holds in the

i-th configuration and the i-th configuration corresponds to a discrete step (i.e., the current and next configuration are associated to the same time point).

The atoms "*term*.method()" and "*term*.method(*parameters*)" are evaluated to true in discrete steps during which the method is being called. In the second case, the values of the variables representing the parameters are set to the value of the arguments passed to the method. The term "*term*.method() **returns** *term*" is evaluated to true in discrete steps during which the method is being returned, and the returned term is set to the value of the returned parameter.

The **not** of a constraint is true iff the constraint is false. The **and** of two constraints is true iff both constraints are true. The **or** of two constraints is true iff either of the two constraints is true. The **implies** of two constraints is true iff the first constraint is false or the second constraint is true.

"**always** *constraint*" is true iff the constraint is true in all future steps (i.e., it is true in the i-th step iff, for all $j \geq i$, it is true in the j-th step). "**never** *constraint*" is equivalent to "**always not** *constraint*". "**in the future** *constraint*" is true iff the constraint is true in a future step (i.e., it is true in the i-th step iff, for some $j \geq i$, it is true in the j-th step). "*constraint1* **until** *constraint2*" is true iff the second constraint is true in a future step, and from the current (the i-th step) to that step (excluded), the first constraint is true (i.e., it is true in the i-th step iff, for some $j \geq i$, the second constraint is true in the j-th step, and, for all h, $i \leq h < j$, the first constraint is true in the h-th step). "**a sequence matching** *regexp*" is true iff there exists a sequence of future steps which satisfies the regular expression defined by *regexp* (i.e., it is true in the i-th step iff, for some $j \geq i$, the regular expression is satisfied by the sequence from the i-th to the j-th step). "**a sequence matching** *regexp* **followed by** *constraint*" is true iff there exists a sequence of future steps which satisfies the regular expression defined by *regexp* followed by an infinite sequence satisfying *constraint* (i.e., it is true in the i-th step iff, for some $j \geq i$, the regular expression is satisfied by the sequence from the i-th to the j-th step, and the constraint is satisfied in the j-th step). "**any sequence matching** *regexp* **triggers** *constraint*" is true iff any sequence of future steps satisfying the regular expression defined by *regexp* is followed by an infinite sequence which satisfies *constraint* (i.e., it is true in the i-th step iff, for all $j \geq i$, if the regular expression is satisfied by the sequence from the i-th to the j-th step, then the constraint must be satisfied in the j-th step).

A regular expression is interpreted (as in PSL) by a finite sub-sequence of a behavior, provided that concatenation of sequences are separated by discrete steps: thus, "*regexp1* ; *regexp2*" is satisfied by a sequence if this is the concatenation of a sequence satisfying the first regular expression and ending with a discrete step, followed by another sequence satisfying the second expression; "*regexp1* | *regexp2*" is satisfied by a sequence if this satisfies either of two regular expressions; "*regexp1* && *regexp2*" is satisfied by a sequence if this satisfies both regular expressions; "*regexp* [*]" is satisfied by a sequence if either this is an empty sequence (a sequence of length 0) or this is the concatenation of a sequence satisfying "*regexp*" and ending with a discrete step, followed by another sequence satisfying "*regexp* [*]".

The quantifiers "**for all**" and "**there exists**" work with a set of objects or a range of indexes (the set of objects may be the universe of a class, or the objects in a multiple attribute). In the case of a set of objects, the constraint quantified with "**for all**" is true iff the constraint is true for all substitutions of the variable with the objects of the set; the constraint quantified with "**there exists**" is true iff the constraint is true for some substitution of the variable with an element of the set. In the case of a range of indexes, the constraint quantified with "**for all**" is true iff the constraint is true for all substitutions of the variable with the indexes in the set; the constraint quantified with

"**there exists**" is true iff the constraint is true for some substitution of the variable with an index in the range.

3.4. Reasoning as Satisfiability Checking

The OTHELLO language is the result of a careful trade-off between expressiveness and automation. Most of the reasoning tasks required for requirements validation can be reduced to *satisfiability* problems, i.e., finding, given a specification, a behavior that satisfies it. Unfortunately, the satisfiability problem for OTHELLO is undecidable even for the quantifier-free fragment [Cimatti et al. 2009]. Nevertheless, we can provide efficient automatic procedures that solve the problem in most cases although they are not complete.

The satisfiability of an OTHELLO specification can be automatically reduced to model checking problems. More specifically, the satisfiability problem is solved with the following automatic steps:

(1) Given a bound for the multiplicity of attributes and a bound on the number of objects per class, the quantifiers can be removed by enumerating the (finite) set of elements. An iterative procedure increments the bound until the formula is found satisfiable or a limit is reached.
(2) The specification is translated into an equi-satisfiable one over discrete time points, where the evolution of the continuous terms has been translated into discrete steps parametrized by a Real variable representing the elapsed time (see [Cimatti et al. 2009]).
(3) The specification is translated into an equivalent symbolically represented, infinite-state fair transition system [Manna and Pnueli 1992].
(4) The transition system is analyzed to look for a computation path [Manna and Pnueli 1992] by applying standard model checking techniques for infinite-state systems (in particular Bounded Model Checking [Biere et al. 1999] and Abstraction Refinement [Clarke et al. 2000] combined with SMT [Sebastiani 2007]); the path has a straightforward correspondence with a model for the original specification.

4. OVERVIEW OF THE METHODOLOGY

We propose a methodology for the validation of the requirements based on their formalization with OTHELLO and on the application of formal validation techniques. One of the crucial activity is the formalization of the requirements, which is a very difficult task, especially for people not expert in formal methods. To alleviate this problem, we propose a preliminary activity of informal analysis which tags the requirements with categories that help the subsequent formalization. Overall, the methodology consists of the following three main steps (see Figure 4):

M1 *Informal analysis phase:* the informal requirements described in the requirements document are categorized and structured in order to produce categorized requirement fragments;
M2 *Formalization phase:* the categorized requirement fragments are formalized in OTHELLO through a set of classes and a set of constraints to produce formalized requirement fragments;
M3 *Formal validation phase:* the formalized requirement fragments are validated with a set of queries to the verification engine and a validation report is automatically produced.

In the following sections, we detail the methodology in terms of the associated subphases, artifacts, and process.

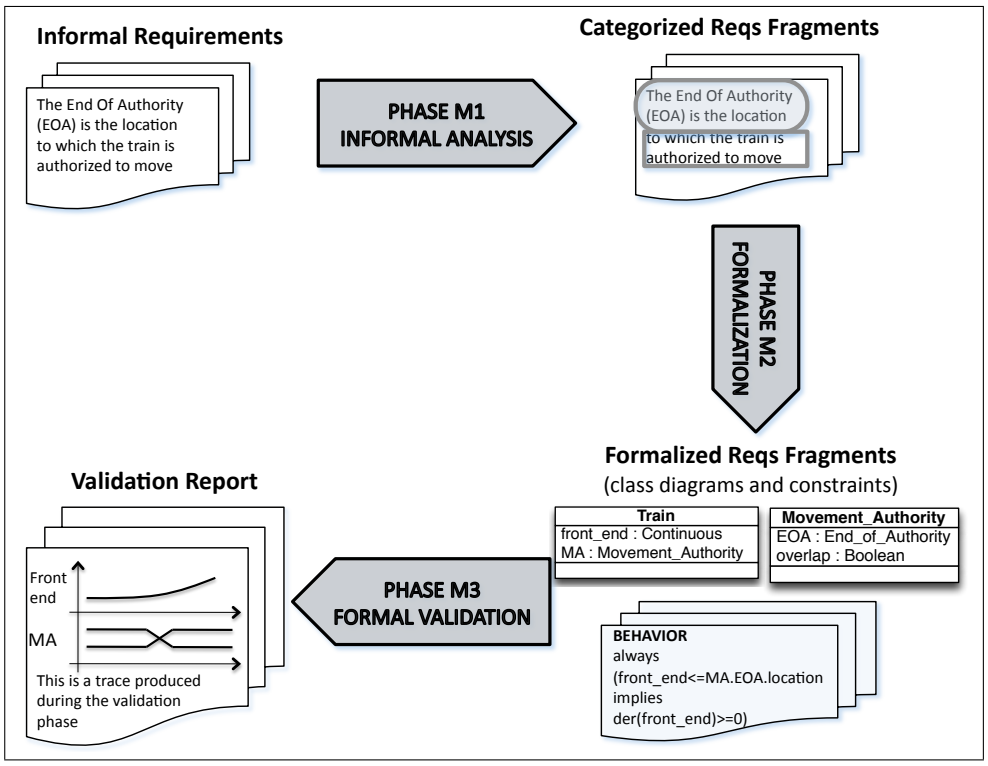

Fig. 4. Overview of the methodology.

5. INFORMAL ANALYSIS PHASE

The first activity in the methodology has the objective of decomposing the informal requirements specification to simplify its formalization. It consists of an informal analysis in which requirement fragments are identified and categorized on the basis of their characteristics. The set of categories have been chosen to have a direct mapping into the elements of an OTHELLO specification, and to reflect the respective role in the validation process. Thus, each category suggests which type of formal element should formalize the requirement fragment.

This phase can be combined with standard analysis techniques of requirements engineering in order to detect if the requirements are unclear, ambiguous or contradictory. These techniques aim at improving the quality of the requirements, although the product is still informal. The categorization helps the analyst in understanding the domain and is used in the next phases of the methodology to guide the formalization by suggesting the use of particular OTHELLO constructs.

5.1. Artifacts

Requirement fragments. We consider a requirement fragment as a part of the informal requirements 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 picture, a comment, and other textual and visual elements.

Categories. Requirement fragments are grouped according to their purpose. We identify eight possible categories: *Glossary*, *Relationship*, *Action*, *Configuration*, *Behavior*,

Table II. 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 relationships between two concepts?	*Relationship*
Does the requirement fragment introduce an action performed by some object of the system-to-be?	*Action*
Does the requirement fragment describe some constraints on the configurations of the system-to-be?	*Configuration*
Does the requirement fragment describe some constraints on the behaviors of the system-to-be?	*Behavior*
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 specification that does not add any information to be formalized and validated?	*Annotation*

Scenario, *Property*, *Annotation*. The conditions specified in Table II define the corresponding category to be assigned to each informal requirement fragment.

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

— *Strong Dependency*: A cannot exist without B.
— *Refinement*: 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 problem.

5.2. Process

5.2.1. Overview. The steps for this informal categorization and analysis are (see Figure 5):

M1.1 *Category driven requirements fragmentation*. Identification and categorization of the informal requirement fragments.
M1.2 *Fragments dependencies definition*. Creation of the dependencies among the informal requirement fragments.

The final result of the informal analysis phase is a set of categorized requirement fragments and their dependencies.

5.2.2. Description. During the category-driven requirements fragmentation the analyst chooses the subset of requirements under analysis from the specification documents, and starts analyzing it to identify the relevant parts of the requirements.

Following the process shown in Figure 5, and exploiting the questions in Table II as guidelines, initially, the analysis is devoted to the identification of the concepts in the requirements subset. The domain concepts are in general names or composed names describing a single concept (such as “Train” or “Movement Authority” in ETCS); they should be isolated and categorized as *Glossary* fragments. *Relationship* fragments represent descriptions of relationships between concepts. Examples of sentences that introduce the category are: “The following characteristics belong to” a given concept or a given concept is the “composition” or “aggregation” of other concepts, and more generally, all the sentences relating concepts to other concepts. *Action* fragments are generally associated to verbs introducing actions performed by objects in the domain.

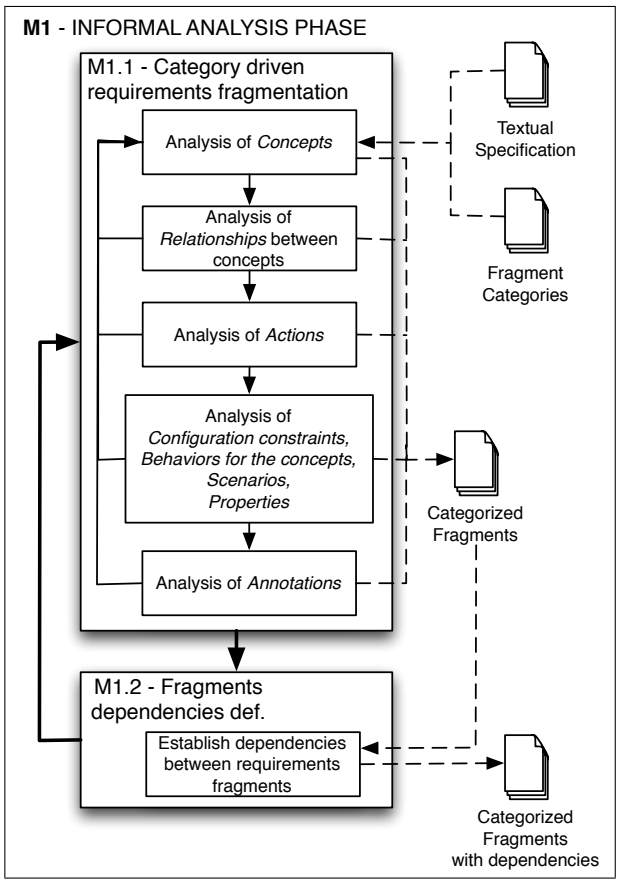

Fig. 5. Details of the informal analysis phase: the process is iterative and proposes the three activities of concept discovery, then the analysis of their relationships and of the actions associated to each concept; a fourth activity is devoted to the categorization of the different kinds of constraints. Finally, to have a complete categorization of the documents, annotations to the documents are categorized as *annotations*. The dependencies between informal requirement fragments are than added.

Examples of actions in the ETCS requirements specification are the verbs related to the movement of objects or communication between objects (such as “send”, “receive”, “start”, “move”, or “brake”). *Configuration* fragments describe constraints on the values for the elements in the domain such as: “the Danger Point is a location beyond the EOA”. *Behavior* fragments describe constraints on the behavior of the system-to-be in terms of actions to perform, ways these actions have to be performed, or conditions to respect, such as: “The train trip shall issue an emergency brake command, which shall not be revoked until the train has reached standstill and the driver has acknowledged the trip” or “The Target Speed at the EOA is the permitted speed at the EOA”. *Scenario* fragments are related to the sentences describing a possible situation, such as a series of actions or events, involving the objects of the domain. A typical scenario is the description of the exchange of messages among objects. Sometimes this kind of fragments is introduced as examples or pictures. A possible railway scenario is: “The ground control center, in some occasions, can repeat the stopping signal to a train three times before entering in alarm mode”. *Property* fragments are used for those fragments that define properties, such as qualities or characteristics that are expected to hold if

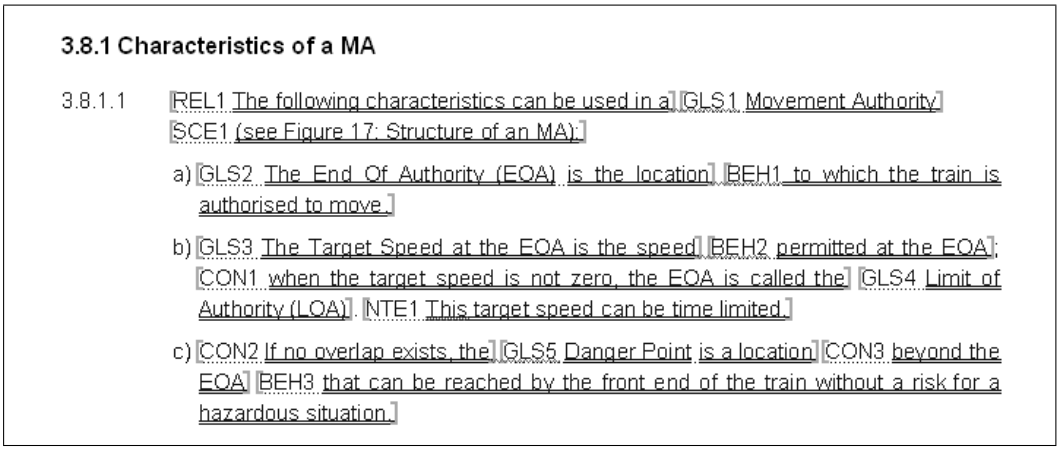

Fig. 6. Identification of ETCS requirement fragments in Rational Requisite Pro.

the requirements are satisfied. An example of such properties is: "two trains cannot be in the same position of the track at the same time". Finally, the text can contain annotations that have the only objective of better explaining requirements. In fact, they refine the concepts described in the previous set of requirements or recall other requirements of the specification without adding any detail to the requirements it is linked to. From the user point of view, it is important to maintain explicit knowledge about these kinds of comments even if they will not be formalized during the second phase; so, these fragments are candidates for *Annotation* category.

After the fragmentation/categorization phase, fragments retrieved during the analysis can be linked to each other exploiting the two kinds of dependency relationships (Strong dependency and Refinement) defined in Section 5.1. In particular, the Strong dependency allows, for example, to logically connect fragments related to a particular concept with fragments describing the properties of that concept in terms of its behaviors or scenarios and that are specified in different parts of the specification. The Refinement relationship allows to link the fragment describing the concept with the fragments refining that concept (such as, the dependency from a fragment introducing a generic "signalling device" to the fragment that defines the generic device as an "antenna"). This way the designer establishes a network of dependencies between the set of identified requirement fragments.

5.3. Example of informal analysis

Let us analyze the excerpt of ETCS introduced in Section 2 and shown in Figure 1. We perform the step M1.1 of the methodology to get a set of requirement fragments. The result is shown in Figure 6.

The text introduces the concept *Movement Authority* (MA) and connects the MA to a list of components: the *End Of Authority* (EOA), the *Target Speed* (TS) at the EOA, the *Limit of Authority* (LOA), the *Danger Point*. We split the sentence into:

— the requirement fragments GLS{1-5}, which introduce the above concepts and therefore are categorized as *Glossary* fragment;
— the requirement fragment REL1, which specifies that a MA is composed of the other concepts and is categorized as *Relationship* fragment;
— the requirement fragments CON{1-3}, which constrain the possible configurations of the above concepts and are categorized as *Configuration* fragment;

— the requirement fragments BEH{1-3}, which constrain the possible behaviors of the above concepts and are categorized as *Behavior* fragment;
— the requirement fragment SCE1, which refers to a figure to give an example of MA and is categorized as *Scenario* fragment;
— the requirement fragment NTE1, which comments the fact that the TS is time limited without giving any further detail and is categorized as *Annotation* fragment.

In the step M1.2, we recognize the following Strong Dependencies:

— BEH1 depends on GLS2;
— BEH2 depends on GLS{2,3};
— CON1 depends on GLS{2,3,4};
— CON2 depends on GLS{2,5};
— CON3 depends on GLS5;
— BEH3 depends on GLS5;
— REL1 depends on GLS{1,2,3,5}.

6. FORMALIZATION PHASE

In the second phase, the categorized requirement fragments are translated into OTHELLO. The language is made accessible by the use of a graphical notation. The UML2 (henceforth, simply UML) [Rumbaugh et al. 2010] graphical representation of class diagrams is adopted. UML classes are annotated with static and temporal constraints to support the definition of an OTHELLO specification.

6.1. Artifacts

Class Diagrams Annotated with Temporal Constraints. The methodology exploits the power of the visual representation of UML class diagrams to visually describe the ontology of the domain under analysis specified via classes, attributes, and generalizations.

We also allow to represent the relationships between two classes with the graphical notation of *associations* and *aggregations*. The former is used when two classes refer to each other with an attribute whose type is the other class. The latter is used to graphically represent an attribute that is an array of elements of another class.

Class diagrams are textually annotated with temporal constraints. The temporal constraints are partitioned into four sets according to their different roles both in the formalization and in the validation phases (see the corresponding process below):

— Configuration constraints, denoted with Ψ_c (restricted to static constraints).
— Behavior constraints, denoted with Ψ_b.
— Scenario constraints, denoted with Ψ_s.
— Property constraints, denoted with Ψ_p.

6.2. Process

6.2.1. Overview. Our methodology requires to proceed with the formalization of the categorized requirement fragment produced as artifact of the informal analysis phase. The formalization phase consists of the following sub-activities (see Figure 7):

M2.1 *Fragments formalization*. Formalize each requirement fragment identified in the informal analysis phase by specifying the corresponding OTHELLO concept.
M2.2 *Linking to requirements fragments*. Link the OTHELLO concept introduced in M2.1 to the requirement fragment. 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.

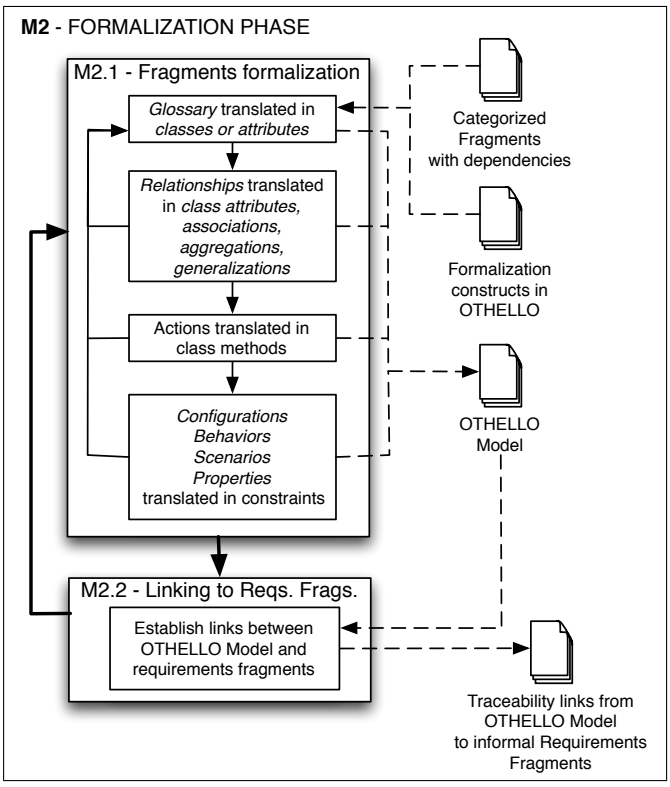

Fig. 7. Details of the formalization phase: after the formalization of the elements that can be translated into classes and related constructs (attributes, relationships and actions), the constraints are translated into OTHELLO. Finally, the traceability links between the OTHELLO model and the informal requirements are specified.

6.2.2. Description. The requirement fragments are translated into specific formal artifacts according to their categorization as described in the Table III. In particular, the *Glossary* fragments can be translated into either Classes, if they represent elements of the system-to-be, or Attributes of classes, if the concepts represent characteristics of other concepts. The *Relationship* category can be translated either into Attribute of a class having as type another class, if the relationship connects elements of the two classes, or into Generalization, in the case there is an *is-a* relationship between the two concepts represented by the classes. Associations and Aggregations can be used as graphical alternatives to the attributes. The *Action* requirement fragments are translated into class Methods. *Configuration*, *Behavior*, *Scenario*, and *Property* fragments are translated in OTHELLO constraints and they are kept separated because they have different roles in the validation. Thus, at the end of the formalization we have a set Ψ_c of OTHELLO constraints formalizing *Configuration* fragments, a set Ψ_b of OTHELLO constraints formalizing *Behavior* fragments, a set Ψ_s of OTHELLO constraints formalizing *Scenario* fragments, and a set Ψ_p of OTHELLO constraints formalizing *Property* fragments.

6.3. Example of formalization

Figure 8 represents the class diagram resulting from the formalization of the output of the informal analysis phase for the ETCS excerpt. For requirement GLS1 we defined

Table III. Mapping between categories and formal artifacts.

Requirement fragment category	Formal artifact
Glossary fragment	Class/Attribute
Relationship fragment	Attribute/Association/Aggregation/Generalization
Action fragment	Method
Configuration fragment	Configuration constraint (Ψ_c)
Behavior fragment	Behavior constraint (Ψ_b)
Scenario fragment	Scenario constraint (Ψ_s)
Property fragment	Property constraint (Ψ_p)
Annotation fragment	Not formalized

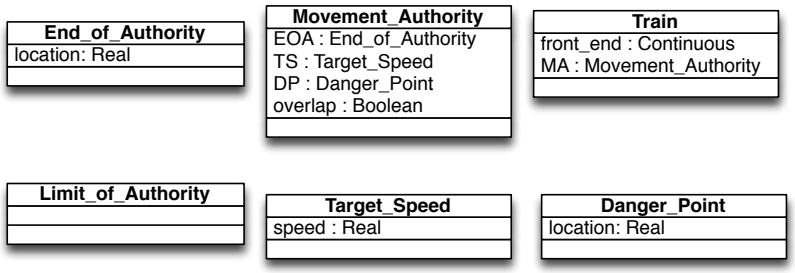

Fig. 8. The Movement Authority and related concepts class model.

class `Movement_Authority`; for requirement GLS2 we defined class `End_of_Authority` with attribute `location` of type Real; for requirement GLS3 we defined the class `Target_Speed` with attribute `speed` of type Real; for requirement GLS4 we defined class `Limit_of_Authority`, and for requirement GLS5 we defined class `Danger_Point` with attribute `location` of type Real.

Class `Train` has been added because the requirements refer to this concept, although it cannot be considered as the formalization of a particular requirement. Rather, either it is introduced elsewhere or it can be considered as a background concept. Similarly, attributes `MA` and `front_end` are added to class `Train`. Their type is `Movement_Authority` for `MA` and Continuous for `front_end`, since we want to refer to its derivative to predicate over the movement of a train.

Requirement REL1 is formalized with attributes `EOA`, `TS`, `DP`, and `overlap` of the class `Movement_Authority`, of type `End_of_Authority`, `Target_Speed`, `Danger_Point`, and `Boolean` respectively.

The formalization of BEH1 requires an OTHELLO constraint on class `Train` to relate the movement of a train to the EOA of its MA. BEH1 says that the train must move towards the EOA. We formalize it with the following constraint:

```
always (front_end<=MA.EOA.location implies der(front_end)>=0)
```

stating that if the train has not yet reached the EOA, its speed is greater than or equal to zero.

Similarly, for the requirement BEH2, we introduced the following Behavior constraint on the class `Train`:

```
always (front_end=MA.EOA.location implies
                                    der(front_end)<=MA.TS.speed).
```

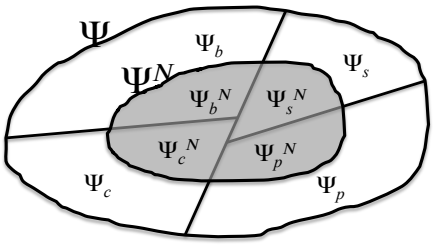

Fig. 9. The set of constraints in the formalized requirements Ψ and those contained in the narrowing Ψ^N, the area in gray ($\Psi_b^N \subseteq \Psi_b$, $\Psi_c^N \subseteq \Psi_c$, $\Psi_s^N \subseteq \Psi_s$ and $\Psi_p^N \subseteq \Psi_p$).

It means that when the train is at the EOA, the speed must be less than or equal to the target speed. Note that we do not have to specify what happens in the other locations, where the behavior of the train is not constrained by this requirement.

We can formalize the *Configuration* fragment CON2 with the following constraint on the `Movement_Authority` class:

```
DP.location>EOA.location
```

It constrains the Danger Point to be beyond the Movement Authority.

7. FORMAL VALIDATION PHASE

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

7.1. Artifacts

Validation problem and Narrowing. During the validation phase, the analyst may consider only a subset of the formalized requirements. We call such subset a *Narrowing*. The narrowing must be compatible with the dependencies defined during the phase M1.2. In particular, if A depends on B, then if A is in the set also B is in the set; on the contrary, if A refines B, then A and B cannot both stay in the

If Ψ is the set of all constraints defined in the formalization phase (thus, $\Psi = \Psi_c \cup \Psi_b \cup \Psi_s \cup \Psi_p$), a narrowing N defines a subset Ψ^N of Ψ which is given by the constraints that formalize *Configuration*, *Behavior*, *Scenario*, *Property* fragments included in the narrowing.

Like Ψ, the set Ψ^N is divided into four partitions depending on the type of the informal requirement fragments they represent (see also Figure 9):

— $\Psi_c^N = \Psi^N \cap \Psi_c$, the set of constraints formalizing the *Configuration* fragments that are considered in the narrowing N;
— $\Psi_b^N = \Psi^N \cap \Psi_b$, the set of constraints formalizing the *Behavior* fragments that are considered in the narrowing N;
— $\Psi_s^N = \Psi^N \cap \Psi_s$, the set of constraints formalizing the *Scenario* fragments that are considered in the narrowing N;
— $\Psi_p^N = \Psi^N \cap \Psi_p$, the set of constraints formalizing the *Property* fragments that are considered in the narrowing N.

Table IV summarizes the set of constraints from the requirements formalization (contained in the set Ψ^N) and from the validation problem.

Table IV. Summary of the constraints involved in the validation phase: part of them came from the formalized requirements, and in particular from the narrowing Ψ^N, other are defined in the validation problem.

	Constraints	Nature of the constraint
Constraints from *narrowing*	Ψ_b^N	Behavior
	Ψ_c^N	Configuration
	Ψ_s^N	Scenarios
	Ψ_p^N	Properties
Additional constraints for the *validation problem*	χ_s	Additional Scenarios
	χ_p	Additional Properties

A *validation problem* consists of a set of objects per each class and the following set of constraints:

— $\Phi_r = \Psi_b^N \cup \Psi_c^N$, where Ψ_b^N and Ψ_c^N are respectively the Behavior and Configuration constraints from the narrowing.
— $\Phi_s = \Psi_s^N \cup \chi_s$, where Ψ_s^N is the set of Scenario constraints from the narrowing, while χ_s are additional Scenario constraints used both for controlling the behavior that we expect to obtain and to represent assumptions on the environment and other parts of the system-to-be whose requirements are not considered.
— $\Phi_p = \Psi_p^N \cup \chi_p$, where Ψ_p^N is the set of Property constraints from the narrowing, while χ_p are additional Property constraints that should be implied by the requirements.

The validation problem is associated with the following set of checks (performed automatically by the verification engine), each consisting of the satisfiability problem for a conjunction of formulas (if the set of formulas is empty, the conjunction is defined to be the formula *true*):

— *Consistency check.* This check aims at formally verifying the absence of logical contradictions in the considered formalized requirement fragments. It is performed by checking the satisfiability of the formula $\bigwedge_{\phi_r \in \Phi_r} \phi_r$.
If the considered formalized requirement fragments is consistent we obtain a behavior trace compatible with it. Otherwise, we obtain a subset of the considered formalized requirement fragments that is responsible for the inconsistency.
— *Scenario compatibility check.* This check aims at verifying whether a set $S \subseteq \Phi_s$ of Scenario constraints is compatible with the constraints Φ_r. It is performed by checking the satisfiability of the formula $\bigwedge_{\phi_r \in \Phi_r} \phi_r \wedge \bigwedge_{\phi_s \in S} \phi_s$.
If the Scenarios are 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 Scenarios to happen.
— *Property check.* This check aims at verifying whether an expected Property $\phi_p \in \Phi_p$ is implied by Φ_r under the assumptions given by a set of Scenario constraints $S \subseteq \Phi_s$. It is performed by checking the validity of the formula $(\bigwedge_{\phi_r \in \Phi_r} \phi_r \wedge \bigwedge_{\phi_s \in S} \phi_s) \rightarrow \phi_p$.
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 fragments, but does not satisfy the Property being analyzed.
If the Property is violated, 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.

Validation results. The above checks will provide diagnostic information in the following forms:

— *Traces.* When consistency and scenario checking succeed, 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 requirements are inconsistent or incompatible with a Scenario, no behavior can be associated to the considered formalized requirement fragments; in these cases, we can generate diagnostic information in the form of a small unsatisfiable subset (unsatisfiable core) of the considered formalized requirement fragments. Similarly, when a Property is satisfied, the unsatisfiable core reveals the subset of requirements which guarantee that the Property holds.

This information can be given to the domain expert, to support the identification and fix of the flaw. The formalized requirement fragments can be traced back to the corresponding categorized requirement fragments, and up to the original requirements in order to remove the identified flaw, or to fix the formalization.

7.2. Process

7.2.1. Overview. The formal validation phase of the methodology is accomplished as follows (see Figure 10):

M3.1 *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 requirements specifying a particular functionality).
M3.2 *Problem creation.* The user defines a set of problems, each one consisting of a set of objects and a set of Scenarios and Properties.
M3.3 *Formal validation.* The user checks the defined problems using automatic model checking techniques and analyzes the results.

7.2.2. Description. In the first step, the user should identify a set of requirements of interest. This narrowing phase M3.1 allows the domain experts to focus only on a subset of the formalized requirement fragments. With this narrowing, we can select specific parts of the requirements specification, considering only some functionalities and enabling to take a *modular* approach to the validation. It also allows to perform several kinds of *what-if* analysis, in particular, it allows to check 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.

In the second step, the user should define a set of problems, each of which consists of a set of objects and a set of Scenario and Property constraints. The classes of the defined objects must be part of the formalization of the selected requirements. The Scenarios represent some evolutions of the objects that we expect to be consistent with the selected requirements. Finally, the Properties represent some facts that we expect to be guaranteed if the selected requirements are satisfied.

In the third step, the user can select a problem and activate the consistency check, the user can select a Scenario and activate the compatibility check, and finally can

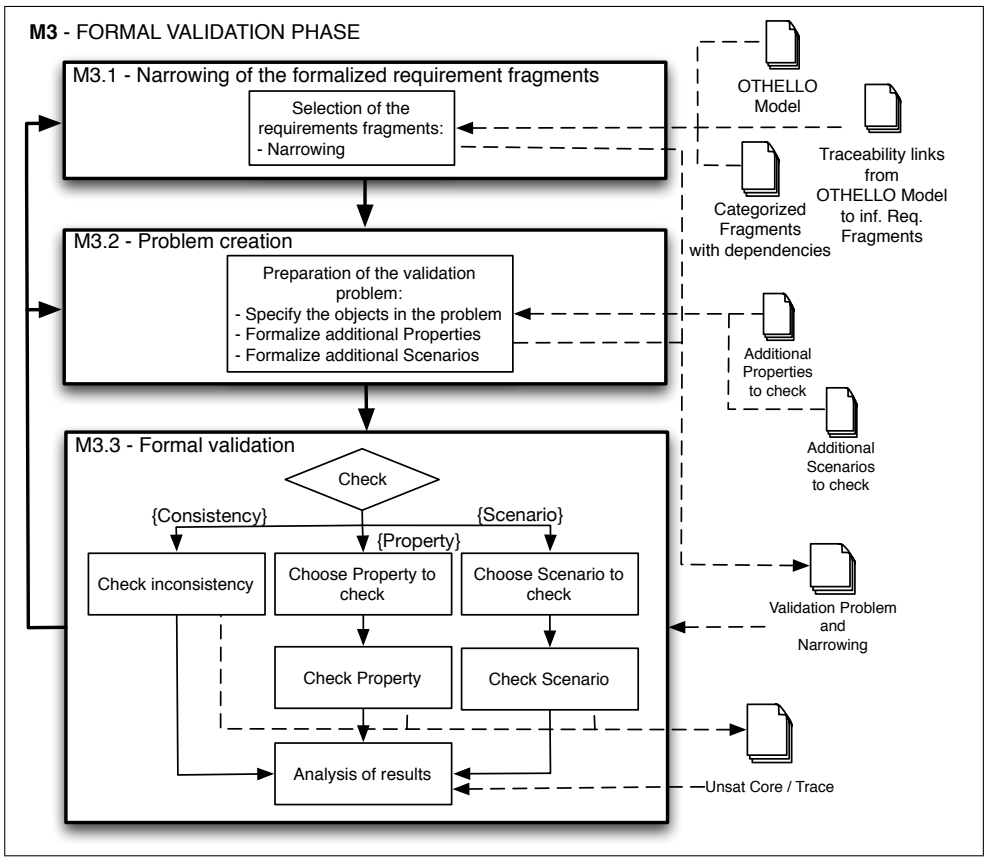

Fig. 10. Details of the formal validation phase. First, the set of requirements is narrowed; second, the validation problem is defined; third, the formal validation is performed by choosing among consistency, property, and scenario checks. The final product is an unsatisfiable core or a trace.

check a selected Property. For each check, the user should analyze the result, which is given in form of trace or unsatisfiable core.

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, new assumptions, and by analyzing different aspects of the requirements specification. 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.

Whenever an issue is identified in any of the three steps, 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 for 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.

7.3. Example of validation

Let us consider the requirements formalized in Section 6.3 and the problem defined by the following set of objects: three trains, ten MAs, ten EOAs, ten TSs, and ten DPs.

The requirements are consistent and this can be witnessed by a trace which describes the case where the trains are already on the location of their EOA and stay in the standstill in different positions.

Let us now consider the Scenario constraint:

```
for all t in trains
   ( in the future t.front_end!=t.MA.EOA.location and
      ( in the future t.front_end=t.MA.EOA.location ) )
```

which asks if it possible that each train at a certain point in time will be in a location different from its EOA, and afterwards will reach the EOA.

If we verify that the requirements are compatible with such Scenario, we will obtain a witness trace. We may, however, obtain an unexpected behavior: the trains may remain in a standstill because their EOA is set to their location. We can then add another Scenario constraint forcing the trains to receive a different EOA, and this time we will obtain a more interesting trace where the trains move and reach the EOA.

Let us then consider the Property constraint:

```
for all t in trains
   ( t.MA.TS.speed=0 implies not t.front_end>t.MA.EOA.location )
```

which claims that if the TS of a train is zero then it is not possible that the train is after the EOA. If we verify the Property, the Property will result correct, but we have to explicitly specify that the trains cannot jump during discrete transitions and that the train is not initially after the EOA.

8. THE EURAILCHECK PROJECT

The methodology has been evaluated within the project EuRailCheck[1] [Chiappini et al. 2010]. In this section, we describe in detail the experience starting from the tool support and how the domain experts used it, and concluding with the lessons we learned during the project. The validation experience allowed to verify the effectiveness of the methodology in the field and to understand the limits and refinements to be worked out. In fact, the language used in the EuRailCheck project was slightly different from OTHELLO, in that, besides the class diagrams, other UML diagrams, namely state machines and sequence diagrams were adopted. Nevertheless, we recognized that these were not necessary from the usability and expressiveness points of view.

8.1. Tool support

Within the project, we developed a tool supporting the methodology (also called EuRailCheck) [Cavada et al. 2009]. To make it more accepted by the end-user we built it on standard-de-facto industrial tools.

We used IBM Rational® RequisitePro®, interfaced with Microsoft Word® and IBM Rational® Software Architect®, to support the informal analysis phase and the traceability of the link between the informal requirement fragments and their formal counterparts. We then interfaced IBM Rational® Software Architect®, IBM Rational® RequisitePro® and the validation tool to support the formalization phase. The developed interface allows to map the formal model into the input language of the validation tool. The verification results can then be mapped back within IBM Rational® Software Architect® to IBM Rational® RequisitePro®, to support the identification and correction of the possible flaws discovered during the validation phase.

The validation tool consists of an extended version of the NUSMV [Cimatti et al. 2000] state-of-the-art symbolic model checker. This extensions provide advanced tech-

[1]The project web site is located at URL `http://es.fbk.eu/projects/eurailcheck`.

niques to compile temporal formulas into automata [Cimatti et al. 2008], and advanced abstraction based verification techniques [Cavada et al. 2007], exploiting the MATH-SAT [Bruttomesso et al. 2008] SMT solver, to efficiently deal with the first-order components.

A view of the software components of the EuRailCheck tool is given in Figure 11.

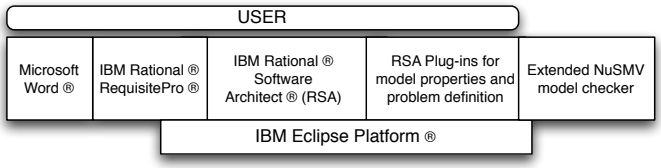

Fig. 11. The EuRailCheck tool architecture.

8.2. The approach at work

Within the project, we considered several parts of the ETCS System Requirements Specification (SRS) in order to exercise all the constructs defined in the methodology. We categorized and formalized a set of 90 requirements, and we produced around 150 classes and 350 different constraints.

The methodology was used by our partners, experts in the railways domain, to formalize and validate 79 requirements. During the informal analysis phase, 83 requirement fragments have been identified, of which 34 were categorized as *Glossary*, 30 as *Behavior*, 11 as *Relationship*, 5 as *Scenario*s and 3 were *Annotation*s. Totally, the formalization produced 113 classes, 3 state machines, 43 constraints, of which 24 of type *Configuration* and 19 of type *Behavior*.

In the validation, we focused on the requirements describing the Moving Authority, and we selected the corresponding formalized fragments. We successfully checked the consistency and different scenarios of increasing complexity. We obtained several traces, whose length had a maximum of 18 steps, and containing a maximum of 454 variable assignments. We also obtained traces describing not admissible behaviors in real settings: for example, a trace showed that some fixed devices were able to move (since no requirement was forcing the opposite); another trace showed a communication between train and track-side happening in unrealistic timings. This fostered the discussion among the domain experts on the necessity or not of new requirements which forbid such behaviors.

The evaluation of the domain experts in the consortium allowed to highlight some characteristics of the methodology. The application of the methodology yields a high degree of *traceability* between the various parts of the specification, the different parts of the formal model and verification artifacts. Thus, all the validation results can be directly related to the set of requirements under analysis. Moreover, the methodology allowed an *incremental approach* strategy. In fact, the formalization started up from the abstract signalling system definitions and, in a second stage, went deeper in the system details. This opens up the possibility to parallelize the modeling activity, and thus to enhance scalability. Finally, the formalization and analysis of the specifications allowed for an *early discovery of lacks* of definitions and minor inconsistencies that could lead to misunderstanding.

After the project, the results were presented for evaluation to domain experts, in form of a workshop and training sessions, attended by experts from different railway organizations and companies including International Union of Railways,

Table V. Questions proposed to the training experts.

	Questions
Q1	*Effectiveness of the informal analysis of the specification (and supporting tool)*
Q2	*Effectiveness of the formalization phase of the specification (and supporting tool)*
Q3	*Effectiveness and usability of formal validation phase of the previously defined model (and supporting tool)*
Q4	*Effectiveness of the overall proposed methodology*
Q5	*Effectiveness of the tool in the overall modeling process*
Q6	*Usability of the tools*
Q7	*Clearness of the output traces produced by the tool*

Table VI. Feedback of 18 experts about the seven questions related to the effectiveness of the methodology and supporting tool together with mean, median, mode, and standard deviation for each question. The scale used to rate the questions is: 1 for *very poor*; 2 for *poor*; 3 for *sufficient*; 4 for *good*; 5 *very good*, with the possibility to avoid giving a response ("*not sure*" value).

	very poor (1)	**poor** (2)	**sufficient** (3)	**good** (4)	**very good** (5)	**not sure**	**mean**	**median**	**mode**	**std. dev.**
Q1	0	2	5	8	3	0	3.67	4	4	0.90
Q2	0	0	5	10	3	0	3.89	4	4	0.67
Q3	0	2	5	6	4	1	3.70	4	4	0.98
Q4	0	4	4	5	5	0	3.61	4	4, 5	1.14
Q5	0	1	6	6	5	0	3.83	4	3, 4	0.92
Q6	0	1	5	9	2	0	3.78	4	4	0.80
Q7	0	1	7	9	1	0	3.56	4	4	0.70

Deutsche Bahn, Rete Ferroviaria Italiana, Gestionnaire des infrastructure Ferroviaires, Ansaldo, Alstom, CEDEX, German Aerospace Center. During the training sessions, 22 domain experts considered 18 ETCS requirements extracted from the ETCS SRS, extracted an average of 28 requirement fragments (mainly related to *Glossary* and *Behavior* requirements due to the nature of the selected fragments), we formalized the extracted requirement fragments with on average 16 classes and 12 constraints. The evaluation phase allowed the experts to reason about the traces produced by the tool, and to discuss several possible refinements and modifications to the piece of specifications analyzed.

At the end of the training we collected the feedback, both quantitative and qualitative from 18 experts. A questionnaire was filled in by the participants anonymously. The questions focused on the methodology and the supporting tool. In particular, the participants were asked to rate the dimensions in Table V.

Each question was rated in a standard scale, with 5 grades (1: *very poor*; 2: *poor*; 3: *sufficient*; 4: *good*; 5: *very good*) with the possibility to avoid giving a response (*not sure*). The results shown in Table VI (analyzed in details in [Chiappini et al. 2010]) indicate a good feeling of the experts in the use of the methodology and supporting tool. In fact, in all the questions, the *good* (4) answer is prevalent, and the standard deviation goes from 1.14 for Q4, for which the mean is 3.61, the median 4, and the modes are 4 and 5, to 0.67 for Q2, for which the mean is 3.89, the median 4, and the mode is 4.

The users also commented the quantitative judgment. From these comments both positive and negative important issues emerged. Some experts judged the methodology to be a good candidate, not only in supporting the validation of the requirements, but also in the phase of the requirements specifications. The discussion also showed that the possibility to trace the requirements analysis, formalization and validation phases, is considered an important characteristic of the methodology. Some experts pointed out that applying the methodology to real-world specifications may be complex and time

consuming. Finally, some experts expressed their preference for a validation of the design rather than validating the requirements. Perhaps, as claimed by other experts, both these validation phases are needed together with a clear traceability between them.

8.3. Lessons learned

On the basis of the different experiences during the project, we discuss some interesting positive and negative lessons emerged. These have been partially already considered to polish the methodology as it is presented in the current paper.

The domain experts identified two main advantages. The first one is the traceability from the informal requirements to the formalized requirements to the trace that exercised those requirements. The second one is the possibility to apply an incremental approach to the formalization. Finally, the third one is the early detection of missing requirements and contradictions in the requirements.

During the application of the methodology all the experts highlighted the need of more structured guidelines during all the phases, with particular emphasis on the categorization phase and its implications in the effectiveness of the methodology. This fact led to the refinement of the categories used during the training of the experts and to the definition of the current set of categories that are strictly connected to the constructs of OTHELLO, as discussed in Section 5.

One of the most interesting points is that the methodology has been accepted by experts in signalling systems, with little knowledge of formal methods. During the training courses, the experts have easily become proficient in formalizing requirements, and managed to specify constraints of significant complexity.

As for expressive power, the language presented here appeared to be sufficient for the portions of the ETCS domain that we considered. In general, other fragments of the ETCS requirements, e.g. the ones dealing with dependability, may also consider aspects that the proposed language cannot deal with (e.g. probabilistic aspects, or deontic modalities). These were considered to be out of scope, but may be the subject of future investigations. Another possible point of extension is the introduction of dedicated logical constructs that may increase the level of usability, in particular for the formalization of more complex ontological aspects.

The full automation of the validation checks has been appreciated by the people who used the tool and that had previous experience with formal methods. The generated traces have been found particularly useful to understand the informal and formal specifications. The time required by the verification engine to check the consistency problems was not an issue, even for large sets of requirements (taking no more than few seconds). As for scenario compatibility and property checking, the validation time was on average acceptable (see also [Cimatti et al. 2011] for some timing evaluation). However, the performance of the verification engine tends to degrade when the complexity of the scenarios increases. More advanced verification techniques, for example based on control patterns exploiting the structure of typical scenarios, are the goal of future investigations.

Two main aspects of the interaction with the underlying verification engine emerged to be possible issues for the usability of the tool. A first issue is the knowledge that the user needs in order to understand the traces generated by the underlying verification engine. In the nominal case, the traces are intuitive and easy to understand, but when unexpected traces are generated it may be difficult to understand why the underlying verification engine produces them. Second, the underlying verification engine showed to be particularly effective in picking the models that the user did not expect. This is particularly effective to disprove some properties, but it may result in a long process

of refinement of the assumptions and/or of the properties to check if desired behaviors were indeed compatible with the requirements.

9. RELATED WORK

The problem of formalizing and validating a requirement specification is one of the main challenges in requirements engineering. Many methodologies have been proposed to solve different aspects, which are related to the management, the elicitation, the representation, the analysis, and the validation of the requirements [Sommerville 2004; van Lamsweerde 2009]. On the other hand, in formal verification, many logics and automata-based languages have been conceived to formalize the property of the system under analysis. In the following sections, we list the related works in order to assess the novelty of our approach.

9.1. Temporal logics for hybrid systems

It is well recognized that hybrid automata [Henzinger 1996] are an appropriate mathematical model to represent safety-critical systems interacting with a physical environment, because they capture both the discrete changes of the control and the continuous evolution of the environment. The behaviors of hybrid automata are represented by hybrid traces where time is modeled with a sequence of intervals and physical quantities are described with continuous real functions. There are many logics used to describe properties of hybrid automata (see [Furia et al. 2010] for a survey). Typically, they are interpreted over discrete or dense sequences of time-points or over sequences of intervals. Most logics for real-time systems (as surveyed in [Alur and Henzinger 1991; 1993]) are interpreted over time-points and therefore cannot force the continuity of functions. The works closely related to the current paper are the extension of LTL with continuous and hybrid semantics (e.g., [de Alfaro and Manna 1995; Kapur 1998; Maler et al. 2008]. To the best of our knowledge, HRELTL [Cimatti et al. 2009] and OTHELLO are the first logics which extend LTL to be interpreted over hybrid traces and which contain both predicates with constraints on the derivatives and predicates to represent discrete changes.

On a different line of research, Duration Calculus [Chaochen et al. 1991] specifies requirements of real-time systems with predicates over the integrals of Boolean functions over finite intervals of time. Extensions such as Extended Duration Calculus [Chaochen et al. 1992], Hybrid Temporal Logic (HTL) [Henzinger et al. 1992], and Hybrid Projection Temporal Logic [Duan et al. 1994] can specify properties over continuous and differentiable functions.

Although there are tentatives to combine the use of the two types of logics (such as in [Liu et al. 2004]), we based our work on LTL, because it has been consolidated as specification language at the industrial level, and has the advantage to allow the reuse of efficient analysis techniques.

Other works such as [Ghezzi et al. 1990; Bois et al. 1997] provided expressive formal languages to represent the requirements. Although, these languages have some similarities with ours such as the adoption of first-order temporal logic, they do not allow the specification of hybrid aspects which are necessary for safety-critical applications. Moreover, these works do not provide a methodology for the analysis of formal requirements and the proposed verification is performed either using interactive theorem proving or with model checking restricted to the propositional fragment.

9.2. Requirements methodology with formal methods

The Software Cost Reduction (SCR) tabular notation is a formal language developed by the US Naval Research Laboratory as part of the SCR project. It is based on a tabular description of transition systems. It has been used in different projects to specify

and analyze the requirements of many practical systems, including control systems for nuclear power plants and avionics systems [Heitmeyer 2007]. The method aims at detecting specification problems such as type errors, missing cases, circular definitions and non-determinism. Although this work has many related points to our approach, the proposed language is not suitable to formalize functional requirements which describe relational constraints of the system at a high level of abstraction, with temporal assumptions on the environment. In fact, the tabular notation targets to formalize how the system-to-be reacts to events. Thus, it can represent configuration constraints and how they change upon some events, but cannot represent the continuous evolution of continuous quantities of the system-to-be and its environment.

Formal Tropos (FT) [Bresciani et al. 2004; Susi et al. 2005; Fuxman et al. 2004] and KAOS [Darimont et al. 1997] are goal-oriented software development methodologies that provide a visual modeling language that can be used to define an informal specification, allowing to model intentional and social concepts, such as those of actor, goal, and social relationships between actors, and annotate the diagrams with temporal constraints to characterize the valid behaviors of the model. Both in FT and in KAOS the specification consists of a sequence of class declarations such as actors, goals, and dependencies. Each declaration associates a set of attributes to the class. The temporal behavior of the instances is specified by means of temporal constraints expressed in a typed first-order LTL. Our proposed methodology is similar to the FT and the KAOS ones, but differs in the expressiveness of the language which can be automatically analyzed and in the guided formalization. In fact, FT and KAOS do not target the formalization of existing textual requirements but focus on the requirements elicitation and analysis. As for the expressiveness, the OTHELLO language allows for constraints on the physical entities that are not possible in the other languages.

The requirements validation proposed by the methodology in [Gervasi and Zowghi 2005] contains aspects that have not been considered in our work, such as the use of a Belief Revision approach to manage the evolution of the information related to the admissibility of a requirement during the elicitation and analysis phases. As for the results of the validation to be given to the user, [Gervasi and Zowghi 2005] proposes to present the results in the form of a set of predicates collections each one representing a possible non conflictual set of requirements. On the contrary, we propose to give back to the user traces and unsatisfiable cores (minimal sets of requirements that cannot be considered together in order to avoid conflicts) produced by the check.

9.3. Linguistic techniques

Works such as [Fantechi et al. 1994; Bucchiarone et al. 2008; Gervasi and Zowghi 2005; Ambriola and Gervasi 2006; Kof 2005; 2009] aim at extracting automatically from a natural language description a formal model using linguistic techniques. In [Kof 2005; 2009], linguistic techniques were used to extract domain ontologies or finite-state automata from requirements specification. In [Gervasi and Zowghi 2005], a complete framework was proposed for automating the requirements documents analysis, formalization, validation and evolution. They use linguistic techniques to automatically extract a formalization of the requirements, and they use model checking and SAT for the validation. The language proposed is based on default logic, that allow the explicit representation of requirements changes and evolution. The validation of the requirements is than performed via finite-state model checking techniques. However, the logic is a variant of propositional logic and therefore it cannot capture the semantics of requirements for safety-critical applications, for example complex real-time constraints (that are really pervasive in ETCS domain). This also simplifies the task of the automatic formalization. In our case, the automation of the translation of informal requirements into OTHELLO is more difficult and probably unfeasible.

In Controlled Natural Language (CNL), the idea is to use a precise subset of the natural language to write a textual specification which can be automatically parsed and analyzed. In particular, in [Nelken and Francez 1996], a CNL for propositional temporal logic was proposed. These linguistic techniques can be in principle integrated into OTHELLO to further increase the usability of the language.

9.4. Model-based specification languages

Several other formal specification languages such as Abstract State Machines [Gurevich 1995], Z [Spivey 1992], B [Abrial 1996], Alloy [Jackson 2002], and OCL [OMG 2006] have been proposed for formal model-based specification.

In particular, Alloy [Jackson 2002] is a formal language for describing structural properties of a system relying on the subset of Z [Spivey 1992] that allows object modeling. An Alloy specification consists of basic structures representing classes together with constraints and operations describing how the structures change dynamically. Alloy only allows to specify attributes belonging to finite domains (no Reals or Integers). Thus, it would have been impossible to model the Train position as requested by the ETCS requirements specifications. Although Alloy supports the "next" operator ("prime" operator), that allow to specify the temporal evolution of a given object, it does not allow to express properties using LTL and regular expressions. Thus, it is limited to state or transition invariants, and it does not allow to specify fairness conditions that are crucial in our context. Alloy supports two kinds of analysis: simulation and checking. In simulation the consistency of an invariant or operation is demonstrated by generating witness (a state or a transition). In checking a consequence of the specification is tested by attempting to generate a counterexample of a user specified length.

OCL [OMG 2006] is a constraint language which enriches UML models in order to express constraints on the object configurations and their changes . It is a pure specification language, so an OCL expression is guaranteed to be without side effects on the model. This means that the state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to specify a state change. In fact, in a post-condition, and only in this context, the expression can refer to values for each property of an object (via the operator $@pre$) at two different time instants: the value of a property at the start of the operation or method and the value of a property upon completion of the operation or method. This allows to describe and to predicate on the temporal evolution of a property. Similarly to Alloy, OCL cannot express some temporal properties, such as fairness. Only limited fragments of OCL can be analyzed automatically.

Similarly, the requirement engineering phase of the DENTUM project [Feilkas et al. 2009] and [Damas et al. 2009] respectively propose to formalize the requirements and analyze critical processes with some variants of state machine. These model-based approaches are not suitable to formalize functional requirements which constrain the temporal evolution of the system (e.g., fairness) and assumptions on the physical environment. Indeed, we claim that such requirements find a more natural and traceable formalization with a temporal logic.

A different approach is proposed in [Kühne et al. 2009], within the VeriSoft XT project, which validates the requirements against the design under verification. This approach may lead to a vacuous validation of the requirements, if the design is not correct. Our techniques aim at validating the requirements before the design is created.

9.5. Formal verification in the railways domain

We distinguish the problem of the formal validation of requirements, which is the focus of this paper, from the verification of systems against the requirements. Several works applied formal techniques to the latter in the railways domain. Many of them

concentrated on the interlocking system, such as [Cimatti et al. 1998; Eisner 2002; Peleska et al. 2004]. Other works tackled the formal verification of the ETCS models, i.e. they formalized a train control system and verify if it was compliant with some expected formal properties.

In [Platzer 2007], a hybrid dynamic logic is proposed for the verification of hybrid systems and it was used to prove safety properties for ETCS [Platzer and Quesel 2009]. In this approach, the description of the system implementation is embedded in the logical formula. The regular expression operations are used to define hybrid programs, which represent the hybrid systems. Properties of hybrid programs are expressed with the modalities of first-order dynamic logic.

Duration Calculus has been used in [Faber and Meyer 2006; Meyer et al. 2008] to specify constraints and properties for a model of the ETCS emergency procedure. The system consists of two trains running on the same track and communicating with the same Radio Block Center that coordinates the movements of the trains in that track.

In the area of railway safety-critical systems formal analysis, the work of [Ortmeier et al. 2005], focuses on the use of a novel failure mode and effect analysis (FMEA), called Deductive Cause-Consequence Analysis for (DCCA), to model safety properties of radio-based railroad crossing. In [Barney et al. 2001] authors face with the safety problem of the calculation of the train breaking distance, proposing a model for the calculation of such a distance in the case of manual or semi-automatic railway driving systems; this problem is not directly detailed in the ETCS requirements specification, being delegated to national implementations of the signalling systems and railway devices.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a formal approach to the validation of requirements tailored to hybrid domains. The underlying formal language, OTHELLO, is expressive enough to represent the functional requirements of hybrid systems in safety-critical applications, yet simple enough to allow for the use by non experts in formal methods. Formal reasoning based on SMT supports a number of automated analysis techniques. The approach is organized in a methodology that drives the domain expert through the formalization process. The methodology and the related tools have been evaluated in a real-world project, where a realistic subset of the ETCS requirements specification has been considered. The results were positively evaluated by domain experts and by potential end users external to the consortium.

In the future, we will pursue the following lines of activity. First, we will investigate the application of automated techniques for linguistic analysis (on the lines of [Gervasi and Zowghi 2005; Kof 2009; Sinha et al. 2009]) to automatically analyze the natural language and extract a partial formalization. Second, we will explore optimizations of the verification engine tailored to the problems coming from the validation of requirements (on the lines of [Cimatti et al. 2007]). Third, we will exploit the availability of formal support for the requirements by integrating the methodology with the subsequent phases of the development flow. In particular, we plan to analyze the impact on the automated generation of test cases and the extraction of coverage measurements with respect to the requirements, and to support the verification of compliance of systems with respect to the requirements they are supposed to implement. Finally, we will extend the methodology to deal with change management and limit the re-validation effort for systems with long time cycles.

ACKNOWLEDGMENTS

We are very grateful to the European Railway Agency for issuing the challenge of the ETCS formalization and validation. We thank Angelo Chiappini and Oscar Rebollo (ERA) for their continuous encouragement and support. We thank Francesco Caruso, Luca Macchi, and Berardino Vittorini from RINA Spa, for their precious feedback after applying the methodology and using the tools. Peter Zurek and Axel Schulz-Klingner from Dr. Graband & Partner GmbH are also thanked for useful discussions. Finally, we thank the Provincia Autonoma di Trento for supporting Stefano Tonetta (project ANACONDA).

REFERENCES

ABRIAL, J.-R. 1996. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers - Principles, techniques and tools*. Addison-Wesley, Reading, MA.

ALUR, R. AND HENZINGER, T. A. 1991. Logics and models of real time: A survey. In *REX Workshop*, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, Eds. Lecture Notes in Computer Science Series, vol. 600. Springer, Berlin, Heidelberg, New York, 74–106.

ALUR, R. AND HENZINGER, T. A. 1993. Real-Time Logics: Complexity and Expressiveness. *Inf. Comput. 104,* 1, 35–77.

AMBRIOLA, V. AND GERVASI, V. 2006. On the Systematic Analysis of Natural Language Requirements with CIRCE. *Autom. Softw. Eng. 13,* 1, 107–167.

BARNEY, D., HALEY, D., AND NIKANDROS, G. 2001. Calculating train braking distance. In *Proceedings of the Sixth Australian workshop on Safety critical systems and software-Volume 3*. Australian Computer Society, Inc., Darlinghurst, Australia, 23–29.

BIERE, A., CIMATTI, A., CLARKE, E. M., AND ZHU, Y. 1999. Symbolic model checking without bdds. In *TACAS*, R. Cleaveland, Ed. Lecture Notes in Computer Science Series, vol. 1579. Springer, Berlin, Heidelberg, New York, 193–207.

BOIS, P. D., DUBOIS, E., AND ZEIPPEN, J.-M. 1997. On the use of a formal r. e. language - the generalized railroad crossing problem. In *RE*. IEEE Computer Society, Washington, DC, USA, 128.

BRESCIANI, P., GIORGINI, P., GIUNCHIGLIA, F., MYLOPOULOS, J., AND PERINI, A. 2004. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems 8,* 3, 203–236.

BRUTTOMESSO, R., CIMATTI, A., FRANZÉN, A., GRIGGIO, A., AND SEBASTIANI, R. 2008. The mathsat 4smt solver. In *CAV*, A. Gupta and S. Malik, Eds. Lecture Notes in Computer Science Series, vol. 5123. Springer, Berlin, Heidelberg, New York, 299–303.

BUCCHIARONE, A., GNESI, S., TRENTANNI, G., AND FANTECHI, A. 2008. Evaluation of natural language requirements in the MODCONTROL project. *ERCIM NEWS* 75, 52–53.

CAVADA, R., CIMATTI, A., FRANZÉN, A., KALYANASUNDARAM, K., ROVERI, M., AND SHYAMASUNDAR, R. K. 2007. Computing predicate abstractions by integrating bdds and smt solvers. In *FMCAD*. IEEE Computer Society, Washington, DC, USA, 69–76.

CAVADA, R., CIMATTI, A., MARIOTTI, A., MATTAREI, C., MICHELI, A., MOVER, S., PENSALLORTO, M., ROVERI, M., SUSI, A., AND TONETTA, S. 2009. Supporting Requirements Validation: The EuRailCheck Tool. In *ASE*. IEEE Computer Society, Washington, DC, USA, 665–667.

CHAOCHEN, Z., HOARE, C. A. R., AND RAVN, A. P. 1991. A calculus of durations. *Inf. Process. Lett. 40,* 5, 269–276.

CHAOCHEN, Z., RAVN, A. P., AND HANSEN, M. R. 1992. An extended duration calculus for hybrid real-time systems. See Grossman et al. [1993], 36–59.

CHIAPPINI, A., CIMATTI, A., MACCHI, L., REBOLLO, O., ROVERI, M., SUSI, A., TONETTA, S., AND VITTORINI, B. 2010. Formalization and validation of a subset of the european train control system. In *ICSE (2)*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, New York, NY, USA, 109–118.

CIMATTI, A., CLARKE, E. M., GIUNCHIGLIA, F., AND ROVERI, M. 2000. NuSMV: A new symbolic model checker. *STTT 2,* 4, 410–425.

CIMATTI, A., GIUNCHIGLIA, F., MONGARDI, G., ROMANO, D., TORIELLI, F., AND TRAVERSO, P. 1998. Formal Verification of a Railway Interlocking System using Model Checking. *Formal Asp. Comput. 10,* 4, 361–380.

CIMATTI, A., ROVERI, M., SCHUPPAN, V., AND TONETTA, S. 2007. Boolean abstraction for temporal logic satisfiability. In *CAV*, W. Damm and H. Hermanns, Eds. Lecture Notes in Computer Science Series, vol. 4590. Springer, Berlin, Heidelberg, New York, 532–546.

CIMATTI, A., ROVERI, M., SUSI, A., AND TONETTA, S. 2011. Formalizing requirements with object models and temporal constraints. *Journal of Software and Systems Modeling (SoSyM) 10,* 2, 147. DOI 10.1007/s10270-009-0130-7.

CIMATTI, A., ROVERI, M., AND TONETTA, S. 2008. PSL Symbolic Compilation. *IEEE Trans. on CAD of Integrated Circuits and Systems 27,* 10, 1737–1750.

CIMATTI, A., ROVERI, M., AND TONETTA, S. 2009. Requirements validation for hybrid systems. In *CAV*, A. Bouajjani and O. Maler, Eds. Lecture Notes in Computer Science Series, vol. 5643. Springer, Berlin, Heidelberg, New York, 188–203.

CLARKE, E. M., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. 2000. Counterexample-guided abstraction refinement. In *CAV*, E. A. Emerson and A. P. Sistla, Eds. Lecture Notes in Computer Science Series, vol. 1855. Springer, Berlin, Heidelberg, New York, 154–169.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. 1999. *Model Checking*. The MIT Press, Cambridge, Massachusetts.

DAMAS, C., LAMBEAU, B., ROUCOUX, F., AND VAN LAMSWEERDE, A. 2009. Analyzing critical process models through behavior model synthesis. In *ICSE*. IEEE, Washington, DC, USA, 441–451.

DARIMONT, R., DELOR, E., MASSONET, P., AND VAN LAMSWEERDE, A. 1997. GRAIL/KAOS: an environment for goal-driven requirements engineering. In *ICSE*. ACM, New York, NY, USA, 612–613.

DE ALFARO, L. AND MANNA, Z. 1995. Verification in continuous time by discrete reasoning. In *AMAST*, V. S. Alagar and M. Nivat, Eds. Lecture Notes in Computer Science Series, vol. 936. Springer, Berlin, Heidelberg, New York, 292–306.

DUAN, Z., KOUTNY, M., AND HOLT, C. 1994. Projection in temporal logic programming. In *LPAR*, F. Pfenning, Ed. Lecture Notes in Computer Science Series, vol. 822. Springer, Berlin, Heidelberg, New York, 333–344.

EISNER, C. 2002. Using symbolic CTL model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhugowaard. *STTT 4,* 1, 107–124.

EISNER, C. AND FISMAN, D. 2006. *A Practical Introduction to PSL*. Springer-Verlag, Berlin, Heidelberg.

ETCS 2006. ERTMS/ETCS — Baseline 3: System Requirements Specifications. SUBSET-026. `http://www.era.europa.eu/Document-Register/Pages/UNISIGSUBSET-026.aspx`.

EuRailCheck-cft 2007. Feasibility study for the formal specification of ETCS functions. Call for tender, `http://www.era.europa.eu/The-Agency/Procurement/Pages/ERA-2007-ERTMS-OP-01.aspx`.

FABER, J. AND MEYER, R. 2006. Model checking data-dependent real-time properties of the european train control system. In *FMCAD*. IEEE Computer Society, Washington, DC, USA, 76–77.

FANTECHI, A., GNESI, S., RISTORI, G., CARENINI, M., VANOCCHI, M., AND MORESCHINI, P. 1994. Assisting Requirement Formalization by Means of Natural Language Translation. *Formal Methods in System Design 4,* 3, 243–263.

FEILKAS, M., FLEISCHMANN, A., HÖLZL, F., PFALLER, C., SCHEIDEMANN, K., SPICHKOVA, M., AND TRACHTENHERZ, D. 2009. A Top-Down Methodology for the Development of Automotive Software. Tech. rep., Technische Universität München.

FURIA, C. A., MANDRIOLI, D., MORZENTI, A., AND ROSSI, M. 2010. Modeling time in computing: A taxonomy and a comparative survey. *ACM Comput. Surv. 42,* 2.

FUXMAN, A., LIU, L., MYLOPOULOS, J., PISTORE, M., ROVERI, M., AND TRAVERSO, P. 2004. Specifying and analyzing early requirements in Tropos. *Requirements Engineering 9,* 2, 132–150.

GERVASI, V. AND ZOWGHI, D. 2005. Reasoning about inconsistencies in natural language requirements. *ACM Trans. Softw. Eng. Methodol. 14,* 3, 277–330.

GHEZZI, C., MANDRIOLI, D., AND MORZENTI, A. 1990. TRIO: A logic language for executable specifications of real-time systems. *Journal of Systems and Software 12,* 2, 107–123.

GROSSMAN, R. L., NERODE, A., RAVN, A. P., AND RISCHEL, H., Eds. 1993. *Hybrid Systems*. Lecture Notes in Computer Science Series, vol. 736. Springer, Berlin, Heidelberg, New York.

GUREVICH, Y. 1995. Evolving Algebras 1993: Lipari Guide.

HEITMEYER, C. 2007. Formal Methods for Specifying, Validating, and Verifying Requirements. *J. UCS 13,* 5, 607–618.

HENZINGER, T. A. 1996. The Theory of Hybrid Automata. In *LICS*. 278–292.

HENZINGER, T. A., MANNA, Z., AND PNUELI, A. 1992. Towards refining temporal specifications into hybrid systems. See Grossman et al. [1993], 60–76.

JACKSON, D. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol. 11,* 2, 256–290.

KAPUR, A. 1998. Interval and point-based approaches to hybrid system verification. Ph.D. thesis, Stanford University, Stanford, CA, USA.

KOF, L. 2005. Natural language processing: Mature enough for requirements documents analysis? In *NLDB*, A. Montoyo, R. Muñoz, and E. Métais, Eds. Lecture Notes in Computer Science Series, vol. 3513. Springer, Berlin, Heidelberg, New York, 91–102.

KOF, L. 2009. Requirements analysis: Concept extraction and translation of textual specifications to executable models. In *NLDB*, H. Horacek, E. Métais, R. Muñoz, and M. Wolska, Eds. Lecture Notes in Computer Science Series, vol. 5723. Springer, Berlin, Heidelberg, New York, 79–90.

KÜHNE, U., GROSSE, D., AND DRECHSLER, R. 2009. Property analysis and design understanding. In *DATE*. IEEE, Washington, DC, USA, 1246–1249.

LIU, Z., RAVN, A. P., AND LI, X. 2004. Unifying proof methodologies of duration calculus and timed linear temporal logic. *Formal Asp. Comput. 16,* 2, 140–154.

LUTZ, R. 1993. Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems. In *RE*. 126–133.

MALER, O., NICKOVIC, D., AND PNUELI, A. 2008. Checking temporal properties of discrete, timed and continuous behaviors. In *Pillars of Computer Science*, A. Avron, N. Dershowitz, and A. Rabinovich, Eds. Lecture Notes in Computer Science Series, vol. 4800. Springer, Berlin, Heidelberg, New York, 475–505.

MANNA, Z. AND PNUELI, A. 1992. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA.

MEYER, R., FABER, J., HOENICKE, J., AND RYBALCHENKO, A. 2008. Model checking duration calculus: a practical approach. *Formal Asp. Comput. 20,* 4-5, 481–505.

NELKEN, R. AND FRANCEZ, N. 1996. Automatic translation of natural language system specifications. In *CAV*, R. Alur and T. A. Henzinger, Eds. Lecture Notes in Computer Science Series, vol. 1102. Springer, Berlin, Heidelberg, New York, 360–371.

OMG 2006. *Object Constraint Language: OMG available specification Version 2.0*. OMG.

ORTMEIER, F., REIF, W., AND SCHELLHORN, G. 2005. Formal safety analysis of a radio-based railroad crossing using deductive cause-consequence analysis (dcca). In *EDCC*, M. D. Cin, M. Kaâniche, and A. Pataricza, Eds. Lecture Notes in Computer Science Series, vol. 3463. Springer, Berlin, Heidelberg, New York, 210–224.

PELESKA, J., GROSSE, D., HAXTHAUSEN, A. E., AND DRECHSLER, R. 2004. Automated Verification for Train Control Systems. In *Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2004), Braunschweig*.

PILL, I., SEMPRINI, S., CAVADA, R., ROVERI, M., BLOEM, R., AND CIMATTI, A. 2006. Formal analysis of hardware requirements. In *DAC*, E. Sentovich, Ed. ACM, New York, NY, USA, 821–826.

PLATZER, A. 2007. Differential dynamic logic for verifying parametric hybrid systems. In *TABLEAUX*, N. Olivetti, Ed. Lecture Notes in Computer Science Series, vol. 4548. Springer, Berlin, Heidelberg, New York, 216–232.

PLATZER, A. AND QUESEL, J.-D. 2009. European train control system: A case study in formal verification. In *ICFEM*, K. Breitman and A. Cavalcanti, Eds. Lecture Notes in Computer Science Series, vol. 5885. Springer, Berlin, Heidelberg, New York, 246–265.

PNUELI, A. 1977. The temporal logic of programs. In *FOCS*. IEEE, Washington, DC, USA, 46–57.

PROSYD 2007. The PROSYD project on property-based system design. `http://www.prosyd.org`.

RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. 2010. *Unified Modeling Language Reference Manual*. Addison-Wesley.

SEBASTIANI, R. 2007. Lazy Satisability Modulo Theories. *JSAT 3,* 3-4, 141–224.

SINHA, A., PARADKAR, A. M., KUMANAN, P., AND BOGURAEV, B. 2009. A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases. In *DSN*. IEEE, Washington, DC, USA, 327–336.

SOMMERVILLE, I. 2004. *Software Engineering*. Addison Wesley.

SPIVEY, J. M. 1992. *The Z Notation: a reference manual, 2nd edition*. Prentice Hall, Inc., Upper Saddle River, NJ, USA.

SUSI, A., PERINI, A., GIORGINI, P., AND MYLOPOULOS, J. 2005. The Tropos Metamodel and its Use. *Informatica 29,* 4, 401–408.

VAN LAMSWEERDE, A. 2009. *Requirements Engineering*. Wiley.

Received Month XX; revised Month YY; accepted Month ZZ