# Automated SMT-based Consistency Checking of Industrial Critical Requirements

Predrag Filipovikj
Mälardalen University
Västerås, Sweden
predrag.filipovikj@mdh.se

Guillermo Rodriguez-Navas
Mälardalen University
Västerås, Sweden
guillermo.rodriguez-navas@mdh.se

Mattias Nyberg
Scania AB CV
Södertälje, Sweden
mattias.nyberg@scania.com

Cristina Seceleanu
Mälardalen University
Västerås, Sweden
cristina.seceleanu@mdh.se

## ABSTRACT

With the ever-increasing size, complexity and intricacy of system requirements specifications, it becomes difficult to ensure their correctness with respect to certain criteria such as consistency. Automated formal techniques for consistency checking of requirements, mostly by means of model checking, have been proposed in academia. Sometimes such techniques incur a high modeling cost or analysis time, or are not applicable. To address such problems, in this paper we propose an automated consistency analysis technique of requirements that are formalized based on patterns, and checked using state-of-the-art Satisfiability Modulo Theories solvers. Our method assumes several transformation steps, from textual requirements to formal logic, and next into the format suited for the SMT tool. To automate such steps, we propose a tool, called `PROPAS`, that does not require any user intervention during the transformation and analysis phases, thus making the consistency analysis usable by non-expert practitioners. For validation, we apply our method on a set of timed computation tree logic requirements of an industrial automotive system called the Fuel Level Display.

## CCS Concepts

•**Computing methodologies** → **Model verification and validation;** *Modeling and simulation; Model development and analysis;*

## Keywords

Requirements Consistency Analysis, Formal Methods, SMT, Z3

## 1. INTRODUCTION

Late detection of errors in the requirements specifications of industrial systems often results in a redesign or reimplementation of certain parts of the system, which leads to a considerable increase of costs. For these reasons, industry has high demands for techniques that enable early debugging of system specifications. This paper addresses the problem of detecting *inconsistencies* within system specifications, which occurs whenever the set of requirements is not realizable as such, due to internal contradictions. For illustration, let us look at the following example: assume a system `S` that at any time operates in either of the two mutually exclusive operational modes `M1` and `M2`. The mode change in the system is event triggered and is described by the following requirements: *R1: "If the event P is observed, the system enters the M1 operational mode and remains in the same for the next 5 time units"; R2: "Whenever event R occurs, it is immediately followed by an event Q";* and *R3: "If event Q occurs, the system must switch to M2 operational mode within 2 time units".* The system specification does not impose any restriction on the occurrence of the events P, Q and R, meaning that there are two ways in which the system specification can be satisfied: trivially and non-trivially. Trivial satisfaction means that one constructs a system `S` where events P, Q and R never occur. For such system it is impossible to violate the above system specification. However, we are interested in the following case: can the system specification be satisfied for a system in which events P, Q, R do occur, and moreover they occur simultaneously? Intuitively, the answer to this question is negative, because the satisfaction of the specification would require the system to be simultaneously in modes `M1` and `M2` somewhere within 2 time units after the occurrence events of P and Q.

For simple scenarios such as the one shown above, expert human-based debugging is usually enough to detect possible inconsistencies. However, the problem arises when one needs to deal with large system specifications composed of several tens or even hundreds of requirements. For such cases, a tool-supported approach, for instance based on formal methods, is needed. Most of the existing consistency analysis approaches [12, 3, 24] are based on model checking. Despite the methods being systematic and exhaustive, some of their characteristics, such as the complexity of constructing the analysis model or the analysis time, which in some cases has been reported to take days, limit the potential of such approaches to be adopted by industry. A less exhaustive, yet systematic and lightweight, approach could

be more suitable for debugging the system specifications at early stages of system development, prior to the existence of any behavioral or structural model of the system.

As a potential solution to this need, in this paper, we propose a completely automated methodology based on Satisfiability Modulo Theories (SMT) [8] for the consistency check of requirements specifications, starting from their description in natural language. We apply specification patterns [10, 21] to formalize the textual requirements into temporal logic, which are later transformed into a set of assertions encoded as a Satisfiability Modulo Theories Library (SMT-LIB) script [5] that can then be fed to an SMT solver of interest. For performing the consistency analysis in this paper we use the Z3 SMT solver [7]. Our idea of SMT-based methodology has already been discussed in our previous work [16], which in this paper is extended in two ways. First, the extension addresses the problem of automation. For that purpose, we propose a tool, called `PROPAS`, which automates the transformation of temporal logic formulas into SMT-LIB assertions suitable for analysis. Rather than encoding the SMT version of the system specification directly in a tool's particular language, such as Z3 Python script, `PROPAS` generates a tool-independent encoding in the SMT-LIB language, which can be used as input to most modern SMT solvers that would fit the purpose. This constitutes the second extension of our work, as compared to our previous paper [16].

The paper continues as follows. In Section 2 we introduce the needed preliminaries, such as timed computational tree logic (TCTL), specification patterns, the formal definition of consistency and the satisfiability modulo theories together with the Z3 tool. Next, in Section 3 we describe the Fuel Level Display (FLD) system, which is an operational industrial system used as a working example for validating the proposed methodology, described in Section 4. In Section 5 we present our tool, `PROPAS`. Section 6 shows the application of `PROPAS` and our method on checking the consistency of the FLD example, using Z3, followed by discussion on the strengths and limitations in Section 7. We compare to related work in Section 8, and conclude the paper in Section 9, where we also outline future research directions.

# 2. PRELIMINARIES

In this section we introduce the concepts that are used in the rest of the paper. First, in Section 2.1 we introduce the computational tree logic (CTL) and its timed extension (TCTL), suitable for the specification of real-time systems, followed by an overview of the specification patterns that represent a user-friendly way to formally specify system requirements for non-experts in formal methods in Section 2.2. Next, in Section 2.3 we introduce a formal definition for consistency of a set of requirements encoded as temporal formulas, and finally in Section 2.4 we give an overview of the Satisfiability Modulo Theories (SMT) method and the Z3 tool used in our work.

## 2.1 (Timed) Computational Tree Logic

Computation tree logic (CTL) is a temporal logic [20] used for the formal specification of finite-state systems. The interpretation of a CTL formula is defined over a branching model $M$ that consists of a non-empty set of states $S$, a successor relation $R$ that assigns a set of successor states to each state and a labeling function *Label* that assigns a set of atomic propositions to each state. Timed CTL (TCTL) is a timed extension of CTL suitable for specifying timed system properties [1]. The concept of time is modeled through a set of non-negative real-valued variables called `clocks`, manipulated by clock formulas expressing constraints over the clocks. The clocks are incorporated into the notion of *state*, which includes the model's location and clock valuation that determines the validity of clock constraints.

The syntax of CTL consists of path quantifiers (*All, Exists*), and path-specific temporal operators. The universal path quantifier *"A"* stands for *"all paths"*, while the existential quantifier *"E"* denotes that *"there exists a path"* from the set of all future paths $P_M(s)$ starting from a given state $s$. A valid CTL formula is of type $\varphi\ U\ \psi$, where $U$ (*"until"*) is the basic path operator, which is combined with either of the two path quantifiers. The rest of the path-specific temporal operators are defined based on the $U$ operator, as follows: the $F$ (*Future*) operator, which denotes that a formula eventually becomes true ($F\varphi \Leftrightarrow true\ U\ \varphi$) and the $G$ (*Globally*) operator which denotes that a given formula is valid in all states along a given path ($G\varphi \Leftrightarrow \neg F \neg \varphi$) [1]. There exists also a weaker version of the $U$ operator called *"weak-until"* denoted as $W$, with the following semantics: $\varphi\ W\ \psi \equiv (\varphi U \psi) \vee G\varphi$. It basically denotes a formula where the $\psi$ might hold, thus $\varphi$ must hold in all future states.

Timed CTL defines a timed version for each of the path-specific temporal operators based on clock constraints. In this paper, we use the following notation for timed path-specific operators: $Oper_{\sim T}$, where: Oper $\in$ {U, W, F, G}; $\sim \in \{=, <, \leq\}$ and T being a positive integer bound on clock variables. For instance, the formula $AF_{\leq T}\varphi$ denotes that on all execution paths starting from some initial state $s_0$, $\varphi$ eventually becomes true within T time units. For more details we refer the reader to previous work [1] [20].

## 2.2 Specification Patterns

The Specification Pattern System (SPS) was introduced by Dwyer et al. [9] to aid practitioners not skilled in formal methods to formally specify system properties. The proposed approach is based on the assumption that systems' specifications are framed within reoccurring solutions, from which a set of patterns can be extracted and saved for future reuse.

The initial SPS catalog [9, 10] is compiled by analyzing more than 500 requirements from various domains. It contains 13 qualitative patterns, divided into two categories: *order* and *occurrence*, expressed in various temporal logics. Each pattern is characterized by two main parts: a *behavior* that it captures, and a *scope* that denotes the extent of program execution in which the behavior must hold. According to Dwyer et al. [10], there are five scopes defined as follows: *Globally* - the entire program execution, *Before Q* - before

the first occurrence of R, *After Q* - after the first occurrence of Q, *Between Q and R* - any part of program execution between events Q and R, *After Q until R* - similar to the previous, except that the occurrence of R is not mandatory.

Each pattern is expressed as a combination of literal and non-literal terminals. The non-literal terminals can be either boolean expressions that describe system properties or integer values that capture timing aspects. The rest of the pattern consists of literal terminals, which are fixed and cannot be changed.

In subsequent research endeavors, the initial SPS catalog has been extended in different ways. In one of the early extensions proposed by Konrad and Cheng [21], the catalog is enriched with real-time specification patterns intended to support the specification of real-time systems. The same extension introduces the *constrained natural language* (CNL) view of the patterns in addition to the various formalisms, such that properties become accessible to a broader set of users. There are other extensions of the specification patterns, such as the one by Grunske [17] which introduces probabilistic patterns. In this paper, we use only the patterns provided in the initial SPS catalog [9, 10] and the real-time extension by Konrad and Cheng [21].

## 2.3 Formal Definition of Consistency

Assuming that the system requirements specification has been encoded as a set of logical formulas, we can consider the following definition to check its consistency:

**Definition 1** (Inconsistent specification)**.** *Let* $\Phi = \{\varphi_1, \varphi_2, ..., \varphi_n\}$ *denote the system requirements specification, where each of the formulas* $\varphi_1, \varphi_2, ..., \varphi_n$ *encodes a requirement, respectively. We say that the set* $\Phi$ *is inconsistent if the following implication is satisfied:* $\varphi_1 \wedge \varphi_2 \wedge ... \wedge \varphi_n \implies False$.

From the definition above, it follows that a system requirements specification is *inconsistent* if there does not exist a truth valuation of the conjunction of all the formulas in the specification. To disprove the inconsistency, it is enough to provide a witness set of valuations of variables that satisfies the conjunction of all the formulas $\varphi_1 \wedge \varphi_2 \wedge ... \wedge \varphi_n$.

## 2.4 Satisfiability Modulo Theories, SMT-LIB and Z3

The problem of determining whether a Boolean formula can be made true by assigning true/false values to the constituent Boolean variables is called the Boolean satisfiability problem (SAT). If a given Boolean formula is satisfiable, the decision procedure generates a model that contains the valuation of the variables such that the formula is true. In the opposite case, that is, when the formula is not satisfiable, there exists no valuation for the constituent variables that will make the formula true. Satisfiability Modulo Theories (SMT) are an extension of SAT, in which some of the symbols are interpreted by a background theory [8]. One such example is the theory of arithmetic that restricts the interpretation of symbols to: {+, -, ≤, 0, 1}.

For SMT-based consistency analysis, in this paper we use the Z3 tool [7] from Microsoft Research, which is a state-of-the-art SMT solver and theorem prover. The input to the tool is a script composed of *assertions* that can be either *declarations* or *formulas*. The assertions are specified using the SMT-LIB language [5], which represents a standard input supported by most of the modern SMT solvers. Declarations can be either constants or functions. Constants are represented as uninterpreted functions with no inputs, whereas the functions are represented as uninterpreted functions that have one or more inputs. The data types in Z3 are called *sorts*, and the set of predefined ones consists of: `Int, Real, Bool` and `Function`. The set of sorts can be additionally extended by user-defined data types. The assertions in SMT-LIB language are specified in a Prolog-like manner. For example, an uninterpreted function $\texttt{fun}_1$ that accepts one parameter $\texttt{param}_1$ is specified as follows: $(\texttt{fun}_1\ \texttt{param}_1)$. The formulas express constraints over the declared variables, which are added to the internal stack using the `assert` command. There are two types of quantifiers: a universal (denoted as `forall`) and existential (denoted `exists`) one. For optimizing the decision procedure, there is a number of tactics provided by Z3.

The command `check-sat` determines whether the current formulas on the Z3 stack are satisfiable or not. If the set of formulas is satisfiable Z3 returns *SAT*, that in our case proves the analyzed consistency. If the set of formulas is not satisfiable, Z3 returns *UNSAT*, thus proving that the set of requirements is inconsistent. In cases when the Z3 cannot determine if the set of formulas is satisfiable or not, it returns *UNKNOWN*. When the command `check-sat` returns *SAT*, an additional command `get-model` can be used to retrieve an interpretation that makes all formulas on the Z3 internal stack true. In case of an *UNSAT*, the minimal inconsistent set of formulas is retrieved by calling the `unsat-core` command.

# 3. MOTIVATING EXAMPLE

In this section we describe the Fuel Level Display (FLD) system, whose requirements we want to analyze for consistency.

FLD is an operational system installed in all heavy-load vehicles produced by Scania, Sweden. The main functionality of the system is to estimate the remaining fuel in the fuel tank and display the correct value to the driver. It is realized through a cooperation between a number of computational components, sensors and actuators.

The estimation of the remaining fuel in the tank is implemented as a software function installed inside the Coordinator (`COO`) Electronic Control Unit (`ECU`). The value is calculated based on the following inputs: remaining fuel in the fuel tank (`FT`), provided by the fuel sensor (`fuelSensor`), and the current fuel consumption provided by the Engine Management (EMS) ECU. The system is classified as *safety critical*, meaning that its correct operation must be assured at all times. The system's failure may lead to hazardous situations that potentially can cause severe material damage to the environment or even endanger human lives.

The simplified architectural breakdown of the FLD functionality is given in Figure 1. The design is based on the concept of *element*, which is an extension over Heterogeneous Rich

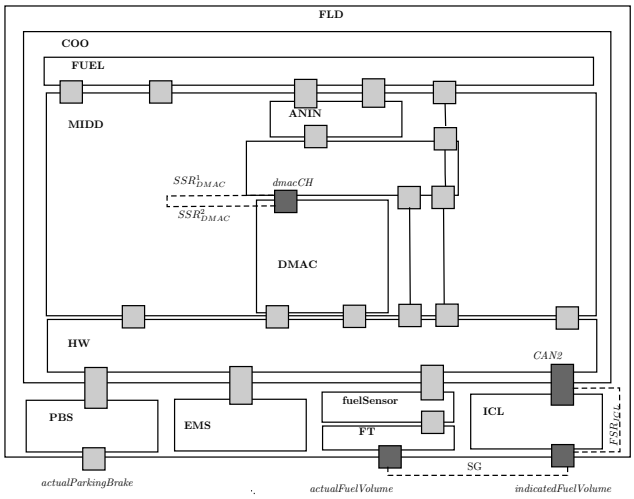



**Figure 1: Simplified version of the high-level architecture of the Fuel Level Display system**

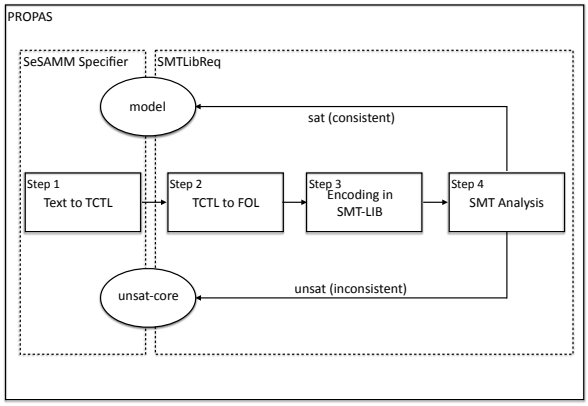



**Figure 2: PROPAS: Automated SMT-based consistency checking of requirements specifications.**

Components [26]. All the entities in the system design, be they physical or logical, are represented via elements. In the architectural design given in Figure 1 the elements are denoted using rectangles (ex: Fuel, PBS, ICL, ect.). The communication between an element and its environment occurs via its interface, represented as a collection of `ports`, denoted with gray rectangles. The behavior of the components is expressed via a set of constraints over their ports, specified according to the contract-based approach by using the *assumption-guarantee* type assertions [26]. For illustration, we list a subset of FLD requirements that we will model and analyze for consistency.

**SG** If actualParkingBrake (aPB) is false, then indicatedFuelVolume (iFV), shown by the fuel gauge, is less than or equal to actualFuelVolume (aFV).

**$\mathbf{FSR_{ICL}}$** If it has not passed more than 1s since the last time CAN message DashDisplay (DD) appeared on CAN2 CAN bus, and the DD message is valid, then the iFV, shown by the fuel gauge, corresponds to FuelLevel (FL) signal value from the DD message.

**$\mathbf{SSR^1_{DMAC}}$** The Direct Memory Access (DMA) channel that corresponds to the input value of dmacCH when Dmac_enableCh() function is called, is enabled when Dmac_enableCh() function finishes its execution.

**$\mathbf{SSR^2_{DMAC}}$** The DMA channel that corresponds to the input value of dmacCH when Dmac_disableCh() function is called, is disabled when Dmac_disableCh() function finishes its execution.

# 4. SMT-BASED METHODOLOGY FOR CONSISTENCY ANALYSIS OF REQUIREMENTS

Our methodology for consistency checking of requirements is illustrated in Figure 2. It consists of four steps given as follows: in *Step 1*, the system requirements are specified in constrained natural language (CNL) via the Specification Pattern System (SPS) [10] [21]. The formalized behavior encoded in the specification patterns and the user input are automatically combined to produce the temporal formulas, expressed in TCTL [1]. Since the SMT solvers operate over first-order logic (FOL) formulas, in *Step 2*, the TCTL patterns are transformed into FOL formulas by instantiating the semantics of path-specific temporal operators and path quantifiers. The FOL formulas are then encoded into SMT-LIB assertions in *Step 3*. In order to facilitate the analysis, the SMT-LIB assertions are additionally optimized by using a number of abstraction rules. Finally, in *Step 4*, we perform the consistency analysis using a state-of-the-art SMT solver (e.g. Z3), which returns the consistency verdict for the considered set of requirements.

In case the conjunction of the requirements is consistent (*SAT* verdict), the tool returns a model that contains a valuation of the system variables satisfying the analyzed requirements specification; in the opposite case (*UNSAT*), the tool generates the minimal inconsistent set (`unsat-core` command) containing the conflicting requirements. The traceability of the requirements starting from their natural language form until the SMT-LIB assertions used for analysis is assured by assigning a unique identifier which is preserved during all steps. To make the methodology potentially useful in industrial settings, we propose a tool called `PROPAS` that automates all the steps from the proposed method. The automation allows one to perform consistency analysis of their system specification with no intervention during the transformation and analysis steps, making `PROPAS` a suitable candidate for industrial adoption.

## 4.1 Step 1: Text to TCTL

The set of FLD requirements used in this study is originally specified in unrestricted natural language using a general-purpose text editor. Such free-text specifications in natural language are readable and expressive, yet sometimes ambiguous and definitely not amenable to automated analysis. In this section, we describe *Step 1* of the method proposed in Figure 2, during which the free-text specification is converted into a more disciplined format with fixed structure and precisely defined semantics.

The formal system specification often represents a bottleneck as the industrial practitioners lack expertise in formal methods required for properly formalizing system specifications. To alleviate the problem, we adopt the specification patterns approach, which is considered a user-friendly formal specification approach that is expressive enough to capture requirements in the automotive domain [15] [25]. In addition to this, we use our in-house tool called `SeSAMM Specifier` [14] that provides a user-friendly interface for requirements specification based on the specification patterns. The tool also offers mechanisms for validation of formalized behaviors and automatic generation of the formal counterpart based on the semantics attached to the specification patterns and the user input. In this paper, we use TCTL to formally encode the system requirements. The decision is justified by the fact that TCTL is suitable for capturing real-time requirements, which are part of the FLD system specification. Moreover, a simplified subset of TCTL can be used for model checking eventual system realizations, for instance by using UPPAAL [22]. Below, we show how the set of FLD requirements given in Section 3 is encoded in CNL using the specification patterns and `SeSAMM Specifier`, followed by their TCTL encoding, as follows:

FLD requirements expressed in CNL via the SPS:

**SG** Globally, it is always the case that when the aPB = False holds, then the iFV $\leq$ aFV holds as well.

**$\mathbf{FSR_{ICL}}$** After CAN2 = DD & DD $\neq$ ERR holds until CAN2 $\neq$ DD, it is always the case that iFV = DD holds for 1s.

**$\mathbf{SSR^1_{DMAC}}$** Globally, it is always the case that when Dmac_EnableCh(chID) = True holds, then the dmacCH(chID) = True holds as well.

**$\mathbf{SSR^2_{DMAC}}$** Globally, it is always the case that when Dmac_DisableCh(chID) = False holds, then the dmacCH(chID) = False holds as well.

FLD requirements in TCTL:

**SG** $AG(\neg\, aPB \Rightarrow iFV \leq aFV)$

**$\mathbf{FSR_{ICL}}$** $AG(CAN2 = DD \wedge DD \neq ERR \Rightarrow A(iFV = DD\, W_{\leq 100}\, CAN2 \neq DD))$

**$\mathbf{SSR^1_{DMAC}}$** $AG(Dmac\_enableCh(chU32) \Rightarrow dmacCH(chID))$

**$\mathbf{SSR^2_{DMAC}}$** $AG(Dmac\_disableCh(chU32) \Rightarrow \neg dmacCH(chID))$

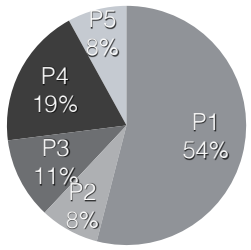



**Figure 3: Pattern distribution for the formalized FLD requirements**

We formalize the complete set of FLD requirements using only five specification patterns, as shown in the list below. The frequency with which the patterns occur, that is the percent of the total FLD requirements specified using a specific pattern is given in Figure 3. The results are aligned with the earlier formalization attempts [15] [25], revealing that, in principle, a small subset of SPS patterns suffices to express the majority of automotive systems' requirements.

**P1:** Globally, Universally: $AG(\varphi)$

**P2:** Timed Globally, Universally: $AG(AG_{\leq T}(\varphi) \Rightarrow \psi)$

**P3:** Globally, Response: $AG(\varphi \Rightarrow AF_{\leq T}\psi)$

**P4:** After $\varphi$ Weak-until $\theta$ Universally $\psi$: $AG(\varphi \Rightarrow A(\psi\, W_{\leq T}\, \theta))$

**P5:** Timed After $\varphi$ Weak-until $\theta$ Universally $\psi$: $AG(AG_{\leq T}(\varphi) \Rightarrow A(\psi\, W_{\leq T}\, \theta))$

## 4.2 Step 2: TCTL to FOL

In this section, we present *Step 2* of the methodology, during which the TCTL patterns are transformed into FOL formulas. The importance of this transformation is twofold: i) it bridges the semantic gap between TCTL formulas and SMT-LIB assertions, and ii) ensures the conservation of information between the two. This step is performed on a pattern level, meaning that once a pattern is transformed, all the requirements instances from that pattern recall the result from the structured derivation of the pattern. Due to page limitation and similarity of proofs, in this section we present only one lemma that shows the structured derivation of two of the TCTL patterns into equivalent FOL formulas. The rest of the patterns are transformed in a similar fashion.

The TCTL to FOL transformation is carried out by instantiating the semantics of the TCTL operators according to the definitions given by Katoen [20], assuming a timed transition system as the underlying semantic model of our system. The semantics of a TCTL formula is based on the following concepts: $\sigma$ denotes a single path from the set of all paths $P_M(s)$ starting from a given position $s$. Each path $\sigma$ is represented as a set of positions, denoted as $Pos(\sigma)$. A position in the path is a pair $(i, d_i)$, where $i$ is the location number, whereas $d_i$ is the time distance. The location determines the set of atomic propositions that are valid for that position, whereas the time distance is a real number that corresponds to the time elapsed during the delay transitions as compared

from the initial state in the path; a set of such points characterizes the states traversed along $\sigma$ while going from state $s_i$ to the successor $s_{i+1}$ for any $i \in \mathbb{N}$. The time elapsed on a path relative to the initial state $s_0$ to any state $s_i$ is defined as:

$$\Delta(\sigma, 0) = 0,$$

$$\Delta(\sigma, i+1) = \Delta(\sigma, i) + \begin{cases} 0, \text{ for discrete transition,} \\ d_i, \text{ for delay transition.} \end{cases}$$

The elapsed time on a path is measured using real-valued variables called *clocks*, which increase with rate one. We denote a clock valuation by $v$. The value of the clocks can be manipulated only through a reset action $(reset\ z\ in\ v)$, which sets a set of clocks to zero. The definition of the reset function is given as follows:

$$(reset\ z\ in\ v)(y) = \begin{cases} v(y), \text{ if } y \neq z, \\ 0, \text{ if } y = z. \end{cases}$$

Lemma 1 below proves the conjectured equivalent FOL form of pattern P4 in TCTL, as a structured derivation that uses the FOL counterpart of a P1, which is also stated by Lemma 1. The proof for (1) has been omitted due to space limitation and the fact that similar proof exists already [20].

**Lemma 1** (P1, P4 into FOL)**.** *Given a transition system M, predicates $\varphi$, $\psi$, $\theta$, a state s of M, and $\omega$ a clock valuation formula, the following two equivalences hold:*

(1)
$s, \omega \models AG_{\geq 0}(\varphi)$
$\Leftrightarrow$
$\forall \sigma \in P_M(s).(\forall (i,d) \in Pos(\sigma).(\sigma(i,d), (z = \Delta(\sigma, i)) \models \varphi))$

(2)
$s, \omega \models AG_{\geq 0}(\varphi \Rightarrow A(\psi\ W_{\leq T}\ \theta))$
$\Leftrightarrow$
$\forall \sigma \in P_M(s).(\forall (i,d) \in Pos(\sigma).\sigma(i,d), (z = \Delta(\sigma, i)) \models (\neg\varphi \vee$
$(\forall \sigma' \in P_M(\sigma(i,d)).(\exists (j,d').(i < j \vee (j = i \wedge d \leq d')) \in Pos(\sigma')$
$.\sigma'(j,d'),\ (z = \Delta(\sigma', j)) \models (\theta \wedge z \leq T)) \wedge (\forall (k,d'').(k\ <\ j \vee$
$(k = j \wedge d'' \leq d')) \in Pos(\sigma').\sigma'(k,d''), (z = \Delta(\sigma', k)) \models (\psi \wedge$
$z < \Delta(\sigma', j)))) \vee (\forall \sigma' \in P_M(\sigma(i,d)).(\forall (j,d')(i\ <\ j \vee (j\ =\ i \wedge$
$d \leq d' \leq d+T) \in Pos(\sigma').\sigma'(j,d'), (z = \Delta(\sigma', j)) \models$
$(\psi \wedge z \leq T))))))$

*Proof.*

(2)
$s, \omega \models AG(\varphi \Rightarrow A(\psi\ W_{\leq T}\ \theta))$
$\Leftrightarrow$ {$AG \Leftrightarrow AG_{\geq 0}$; Rule: $\varphi \Rightarrow \psi \Leftrightarrow \neg\varphi \vee \psi$, definition of $W_{\leq T}$}
$s, \omega \models AG_{\geq 0}(\neg\varphi \vee A(\psi\ U_{\leq T}\ \theta) \vee AG_{\leq T}(\psi))$
$\Leftrightarrow$ {Rule: $AG_{\leq T}\varphi \Leftrightarrow \neg EF_{\leq T}\neg\varphi$, definition of $F_{\leq T}$}
$s, \omega \models AG_{\geq 0}(\neg\varphi \vee A(\psi\ U_{\leq T}\ \theta) \vee \neg E(True\ U_{\leq T}\ \neg\psi))$
$\Leftrightarrow$ {Definition of $U_{\leq T}$; let z be a 'fresh' clock}
$s, \omega \models z\ in\ AG_{\geq 0}(\neg\varphi \vee A((\psi \wedge z \leq T)\ U\ \theta) \vee \neg E(True$
$U\ (\neg\psi \wedge z \leq T)))$
$\Leftrightarrow$ {Semantics of z in $\varphi$}
$s, reset\ z\ in\ \omega \models AG_{\geq 0}(\neg\varphi \vee A((\psi\ \wedge z \leq T)\ U\ \theta) \vee$
$\neg E(True\ U\ (\neg\psi \wedge z \leq T)))$
$\Leftrightarrow$ {Definition of $AG_{\geq T}, A(\varphi\ U\ \psi), E(\varphi\ U\ \psi)$}
$\forall \sigma \in P_M(s).(\forall (i,d) \in Pos(\sigma).\sigma(i,d), (reset\ z\ in\ \omega) +$
$\Delta(\sigma, i) \models (\neg\varphi \vee (\forall \sigma' \in P_M(\sigma(i,d)).(\exists (j,d') \gg (i,d)$
$\in Pos(\sigma').\sigma'(j,d'), (reset\ z\ in\ \omega) + \Delta(\sigma', j) \models (\theta \wedge z \leq T)$
$\wedge (\forall (k,d'') \ll (j,d') \in Pos(\sigma').\sigma'(k,d''), (reset\ z\ in\ \omega)$
$+ \Delta(\sigma', k) \models (\psi \wedge z \leq \Delta(\sigma', j)))) \vee \neg(\exists \sigma' \in P_M(\sigma(i,d)).$
$(\exists (j,d') \gg (i,d) \in Pos(\sigma').\sigma'(j,d'), (reset\ z\ in\ \omega) +$
$\Delta(\sigma', j) \models (\neg\psi \wedge z \leq T) \wedge (\forall (k,d'') \ll (j,d') \in Pos(\sigma').$
$\sigma'(k,d''), (reset\ z\ in\ \omega) + \Delta(\sigma', k) \models True))))))$
$\Leftrightarrow$ {Logic, definition of total order, semantics of reset z in $\omega$}
$\forall \sigma \in P_M(s).(\forall (i,d) \in Pos(\sigma).\sigma(i,d), (z = \Delta(\sigma, i)) \models (\neg\varphi \vee$
$(\forall \sigma' \in P_M(\sigma(i,d)).(\exists (j,d').(i < j \vee (j = i \wedge d \leq d'))$
$\in Pos(\sigma').\sigma'(j,d'), (z = \Delta(\sigma', j)) \models (\theta \wedge z \leq T)) \wedge$
$(\forall (k,d'').(k\ <\ j \vee (k = j \wedge d'' \leq d'))\ \in Pos(\sigma').\sigma'(k,d''),$
$(z = \Delta(\sigma', k)) \models (\psi \wedge z < \Delta(\sigma', j)))) \vee (\forall \sigma' \in P_M(\sigma(i,d)).$
$(\forall (j,d')(i\ <\ j \vee (j\ =\ i \wedge d\ \leq\ d'\ \leq d+T)\ \in Pos(\sigma')$
$.\sigma'(j,d'),\ (z = \Delta(\sigma', j)) \models (\psi \wedge z \leq T))))))$

*Q.E.D.*

The FOL formulas obtained by similar derivations, which correspond to the rest of the patterns are given below.

P2 $\forall \sigma \in P_M(s).(\forall (i,d) \in Pos(\sigma).\sigma(i,d), (z = \Delta(\sigma, i)) \models ((\exists \sigma'$
$\in P_M(\sigma(i,d)).(\exists (j,d').((i < j \vee (i = j \wedge d' \leq d+T)) \in$
$Pos(\sigma').\sigma'(j,d'), (z = \Delta(\sigma', j)) \models (z \leq T \wedge \neg\varphi)) \vee \psi))))$

P3 $\forall \sigma \in P_M(s).(\forall (i,d) \in Pos(\sigma).\sigma(i,d), (z = \Delta(\sigma, i)) \models (\neg\varphi \vee$
$(\forall \sigma' \in P_M(\sigma(i,d)).(\exists (j,d').(i < j \vee (i = j \wedge d \leq d'))$
$\in Pos(\sigma').\sigma'(j,d'), (z = \Delta(\sigma', j)) \models (z \leq T \wedge \psi)))))$

P5 $\forall \sigma \in P_M(s).(\forall (i,d) \in Pos(\sigma).\sigma(i,d), (z = \Delta(\sigma, i) \models$
$(\neg(\forall \sigma \in P_M(\sigma(i,d)).(\forall (j,d).(j > i \vee (j = i \wedge d \leq d \leq T)$
$\in Pos(\sigma).\sigma(j,d), (z = \Delta(\sigma, j)) \models (z \leq T \wedge \varphi)))) \vee$
$(\forall \sigma \in P_M(\sigma(i,d)).(\exists (j,d).(j > i \vee (j = i \wedge d \leq d) \in Pos(\sigma).$
$\sigma(j,d), (z = \Delta(\sigma, j)) \models (z \leq T \wedge \theta))) \wedge (\forall (k,d) \in Pos(\sigma).$
$((k < j \vee (k = j\ \wedge\ d \leq d)) \in Pos(\sigma).(\sigma(k,d), (z = \Delta(\sigma, k)) \models$
$(z \leq \Delta(\sigma, j) \wedge \psi)) \vee (\forall \sigma \in P_M(\sigma(i,d)).(\forall (j,d).(j > i \vee (j = i$
$\wedge\ d \leq d \leq d+T)) \in\ Pos(\sigma).\sigma(j,d), (z = \Delta(\sigma, j)) \models$
$(z \leq T \wedge \psi)))))))))$

Returning to Definition 1, the conjunction of all requirements is obtained by instantiating the above patterns in FOL. Next, the conjunction is encoded as SMT-LIB assertions, which can then be used as an input into an SMT solver of choice, Z3 in our case, for checking the consistency.

## 4.3 Step 3: Encoding in SMT-LIB Language

In this section, we present the third step of our methodology, in which FOL formulas are encoded as SMT-LIB assertions,

which can be analyzed by state-of-the-art SMT solvers. In this step, we apply the following three encoding rules:

**R1:** Directly map the FOL constructs into SMT-LIB syntax elements. For instance, mapping the quantifiers ($\forall$ into `forall`, $\exists$ into `exists`, etc.), modeling port values as functions of time, etc.

**R2:** Reduce complexity by abstraction: (a) eliminate path ($\sigma$) universal quantifiers, and (b) collect location ($i$) and time in location ($d$) into a tuple position ($pos$).

**R3:** Abstract the universally quantified $pos = (i, d)$ to the universally quantified $pos.d$.

The process of applying rules **R1**, **R2** and **R3** on the set of patterns from Section 4.2 can be illustrated as follows:

$$P_i \xrightarrow{R1,R2} P_i' \xrightarrow{R3} P_{i\,SMT},\ i \in [1,5]$$

The application of rule **R1** results in an SMT-LIB script where each assertion corresponds to an individual requirement, with quantifiers and Boolean expressions encoded using SMT-LIB-specific constructs. Due to the underlying branching model over which the TCTL formulas are interpreted (Section 2.1), the assertions are quantified over three variables: execution path (s-paths), locations and clock valuations. However, only the clock quantifiers are bounded due to the timed-constrained nature of the system specification.

The number of quantified variables has a negative impact on the decidability of the SMT procedure [23]. To remedy this, we propose an abstraction technique (rules **R2** and **R3**) that reduces the number of quantified variables in the assertions, abstracting only the information related to variables that cannot be sources of requirements inconsistency (e.g., $\sigma$, and $i$). This is possible because all formulas are universally quantified over the branches, and there is no fixed labeling function as the model of the system is not available. Further, we collect location and time variables into a tuple position, denoted by *pos* [20]. To access the location component of the position we write *pos.i*. Similarly, time valuation in that position is obtained by *pos.d*.

For the FLD requirements, the path component of all FLD properties is always universally quantified because all the requirements are safety requirements, meaning that no inconsistency can occur due to path quantifiers, as existentially quantified path properties do not exist in our case study. Therefore, proving consistency on an arbitrary path of infinite length (chosen via the "select" operator) suffices. Consequently, the quantified path variable disappears in our SMT-LIB encoding.

All our patterns rely on semantic models in which progress is ensured by instantaneous discrete transitions in which location index *pos.i* increases, or via delay transitions, which model the passage of time while the system remains in the same location, causing an increase of the time distance compared to the initial position on the path, that is *pos.d* increases. The progress along the path is modeled by the binary operator "$\ll$" that compares positions, defined as: `pos` $\ll$ `pos'` $\iff$ `(pos.i < pos'.j)` $\vee$ `((pos.i = pos'.j)` $\wedge$ `(pos.d < pos'.d'))`. Possible inconsistencies can arise from contradicting formulas that should hold in each position, e.g. $\varphi$, $\psi$ etc., at or from a certain time point on.

By applying the rules **R1** and **R2** explained above we obtain the following valid abstracted versions of patterns P1-P5:

$P1'$ : $select\ \sigma \in P_M(s).(\forall pos \in Pos(\sigma).pos.i, (z = \Delta(\sigma, pos.i) \models \varphi))$

$P2'$ : $select\ \sigma \in P_M(s).(\forall pos \in Pos(\sigma).pos.i, (z = \Delta(\sigma, pos.i)) \models (\neg(select\ \sigma' \in P_M(pos).(\forall pos'.(pos \ll pos' \ll pos + T) \in Pos(\sigma').pos'.i, (z = \Delta(\sigma', pos'.i)) \models (z \leq T \wedge \varphi)) \vee \psi)))$

$P3'$ : $select\ \sigma \in P_M(s).(\forall pos \in Pos(\sigma).pos.i, (z = \Delta(\sigma, pos.i) \models (\neg\varphi \vee (select\ \sigma' \in P_M(pos).(\exists pos'.(pos \ll pos' << pos + T).pos'.i, (z = \Delta(\sigma', pos'.i)) \models (z \leq T \wedge \psi))))))$

$P4'$ : $select\ \sigma \in P_M(s).(\forall pos \in Pos(\sigma).pos.i, (z = \Delta(\sigma, pos.i)) \models (\neg\varphi \vee (select\ \sigma' \in P_M(pos).(\exists pos'.(pos \ll pos \ll pos' + T \in Pos(\sigma').pos'.i, (z = \Delta(\sigma', pos'.i)) \models (\theta \wedge z \leq T) \wedge (\forall pos'' \ll pos' \in Pos(\sigma').pos''.i, (z = \Delta(\sigma', pos''.i)) \models (z < \Delta(\sigma', pos'.i) \wedge \psi))) \vee (select\ \sigma' \in P_M(pos).(\exists pos'.(pos' \ll pos) \in Pos(\sigma').pos'.i, (z = \Delta(\sigma', pos'.i)) \models (\psi \wedge z \leq T)))))))$

$P5'$ : $select\ \sigma \in P_M(s).(\forall pos \in Pos(\sigma).pos.i, (z = \Delta(\sigma, pos.i)) \models (\neg(select\ \sigma' \in P_M(pos).(\forall pos'.(pos' \gg pos \in Pos(\sigma')).(pos'.i, (z = \Delta(\sigma', pos'.i)) \models (z \leq T \wedge \varphi)))) \vee (select\ \sigma' \in P_M(pos).(\exists pos'.(pos' \gg pos) \in Pos(\sigma').pos'.i, (z = \Delta(\sigma', pos'.i)) \models (z \leq T \wedge \theta))) \wedge (\forall pos''.(pos \ll pos'' \ll pos') \in \sigma'.(pos''.i, (z = \Delta(\sigma', pos''.i)) \models (z \leq \Delta(\sigma', pos'.i) - \wedge\psi)))) \vee (select\ \sigma' \in P_M(pos).(\forall pos'.(pos \ll pos' \ll pos + T) \in Pos(\sigma').(pos'.i, (z = \Delta(\sigma', pos'.i)) \models (z \leq T \wedge \neg\psi)))))$

Finally, we apply rule **R3** on *P1'*, ..., *P5'* to abstract $\forall pos$ and $\exists pos$ into $\forall pos.d$ and $\exists pos.d$ as it is the only component of the tuple that counts for the inconsistency checking. The abstracted patterns can be encoded into SMT-LIB assertions that contain only one quantified component, *pos.d* modeled via the real-valued variable "`time`". In addition, the predicates ($\varphi$, $\psi$, $\theta$) in the FOL patterns are substituted with Boolean expressions over the system variables represented as functions of time denoted as ($\varphi$ *time*), ($\psi$ *time*) and ($\theta$ *time*), respectively. The complete set of patterns encoded in SMT-LIB is presented in the list below:

$P1_{SMT}$ $(forall((time\ Real))(=\ (var_1\ time)\ val_1))$

$P2_{SMT}$ $(forall((time\ Real))(=>\ (and(=\ (var_1\ time)\ val_1)\ (not(exists\ ((time_1\ Real))(and\ (not\ (=\ (var_1\ time)\ val_1))\ (>=\ time_1\ time)(<=\ time_1\ (+\ time\ T))))))\ (=\ (var_2\ time)\ val_2)))$

$P3_{SMT}$ $(forall((time\ Real))(=>\ (=\ (var_1\ time)\ val_1)(exists\ ((time_1\ Real))(and\ (>\ time_1\ time)(<\ time_1\ (+\ time\ T))\ (=\ (var_2\ time_1)\ val_2)))))$

$P4_{SMT}$ $(forall\ ((timeReal))(=>\ (=\ (var_1 time)\ val_1)\ (or\ (exists\ ((time_1 Real))(and\ (>=\ time_1\ time)(<=\ time_1\ (+\ time\ T))(=\ (var_2\ time_1)\ val_2)(not\ (exists\ ((time_2\ Real))\ (and\ (>=\ time_2\ time)(<=\ time_2\ time_1)(=\ (var_2\ time_2)\ val_2)(not\ (=\ (var_3\ time)\ val_3)))))))(not(exists\ ((time_1$

$Real))\ (and\ (>=\ time_1\ time)(<\ time_1\ (+\ time\ T))$
$(not\ (=\ (var_2\ time_1)\ val_2))))))))$

$P5_{SMT}$ $(forall\ ((time\ Real))(=>\ (=>\ (=\ (var_1\ time)\ val_1)$
$(not\ (exists\ ((time_1\ Real))(and\ (>=\ time_1 time)$
$(<\ time_1(+\ time\ T))(not\ (=\ (var_1\ time_1)\ val_1))))))$
$(or\ (=>\ (=\ (var_2\ time)\ val_2)(not\ (exists\ ((time_1\ Real))$
$(and\ (>=\ time_1 time)(<\ time_1(+time K))(not\ (=\ (var_2$
$time_1)\ val_2))))))\ (exists\ ((time_1\ Real))(and\ (>=\ time_1$
$time)(<\ time_1\ (+\ time\ K))(=\ (var_3\ time)\ val_3)(not\ (exists$
$((time_2\ Real))(and\ (>=\ time_2\ time)(<\ time_2\ time_1)$
$(not\ (=\ ((var_2\ time_2)\ val_2))))))))))$

The original set of requirements expressed via patterns *P1*, ..., *P5* are stronger than their counterparts encoded in SMT-LIB using the abstraction rules. Consequently, proving the inconsistency of the encoded versions means proving the inconsistency of the original ones, however the similar inference does not hold for the positive case in which the encoded versions are proven consistent. However, practically, we can still infer the consistency of the original requirements if their abstract encoding is satisfiable, based on our argumentation around the only possible sources of inconsistency.

In the next section, we show how the complete procedure of formalization of natural language requirements and their transformation into a format suitable for SMT-based consistency analysis is automated via our `PROPAS` tool. The automation is a very important feature of the approach that makes it accessible for users not skilled in formal methods who want to check the consistency of their specifications and contributes to its potential adoption in industrial settings.

# 5. TOOL SUPPORT: PROPAS

In this section, we present our tool called `PROPAS` (**PRO**perty **PA**tten **S**pecification and analysis), illustrated in Figure 2, which automates the procedure of transforming the TCTL formulas into an SMT-LIB script suitable for analysis with a state-of-the-art SMT solvers. The core of the tool is a library called `SMTLibReq`, which provides the underlying functionalities for transforming the temporal formulas into SMT-LIB, by automating Steps 2 and 3 (described in Sections 4.2 and 4.3, respectively) of our methodology. It is an open source project, with the source code available freely [13].

## 5.1 The SMTLibReq Library

The high-level design of the `SMTLibReq` library is given in Figure 4. The overall functionality of the library is realized by two components, depicted as squares: i) a parsing engine, called `ExpressionParser`, that transforms logical expressions given as strings into binary tree structures encoding both the semantics and the syntax of each part of the original expression (the `Formula`, represented with a circle in Figure 4); and ii) a transformation engine, called `FormulaTransformer`, that transforms formulas into a formalism of interest, in this case, an SMT-LIB standard script.

The `SMTLibReq` library is designed modularly. There are numerous advantages of such design, including reduced complexity, separation of concerns, higher testability and maintainability. An additional benefit is the fact that it can be used independently of the `PROPAS` tool, meaning that it can be used as an external library in any other project or tool developed using the `C#` programing language within the `.NET` framework.

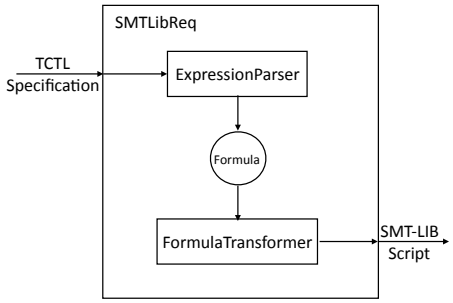


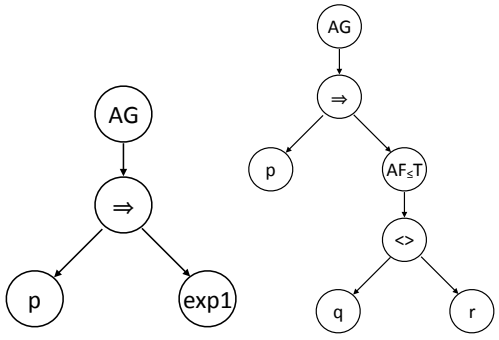

**Figure 4: High-level Architecture of the SMTLibReq Library**

### 5.1.1 ExpressionParser

The main functionality of the `ExpressionParser` is to parse logical formulas encoded in TCTL provided as arrays of characters (strings) into a tree-structure that encodes both the syntax and the semantics of the input formula. In the following, the input formula provided as string will be referred to as *expression*, whereas the tree structure will be referred to as *formula*. Concretely, the produced formula is of type binary tree, which is suitable for encoding unary and binary operators. Considering the fact that the primary focus of our work are system properties expressed in TCTL, the expressiveness of binary tree suffices. In the binary-tree structure there are two types of nodes: *internal* and *leaf* nodes. Each internal node represents an operator, be it unary or binary, whereas the leaf nodes represent the atomic propositions of the expression.

The parsing of the expressions into formulas is performed in two steps, called *flattening* and *parsing*, respectively. The *flattening* procedure represents a preprocessing routine, during which complex formulas composed of a number of subformulas are transformed into flat ones. Such flat structures are then used as an input for the parsing procedure that generates the binary tree that encodes the original formula. Both the *flattening* and *parsing* procedures are generic and applicable to most of the formalisms, whereas the operators and their ordering must be defined for a specific one. Currently, the `ExpressionParser` supports the transformation of TCTL and FOL properties composed of binary operators only into SMT-LIB assertions. For the next releases, we plan to extend the library to support more formalisms. In the following, we illustrate the transformation process of a logical expression into a formula.

Let us consider the following TCTL formula, encoded as string: $AG(p \Rightarrow AF_{\leq T}(q <> r))$, which captures the following behavior: *it is always the case that whenever the proposition `p` becomes true, then eventually within T time units `q <> r` has to become true as well.* One should note that in our work the "<>" operator denotes boolean inequality and not the future path-specific temporal operator, which is denoted




(a) (b)

**Figure 5: Binary-tree representation of a TCTL formula produced by the ExpressionParser: a) an intermediate flat formula and b) final formula.**

by *"F"*. During the parsing procedure, the original formula is analyzed for existence of sub-formulas. In this case, the expression contains one sub-formula: $AF_{\leq T}(q <> r)$. In order to create a flat expression, the sub-formula is replaced by a temporary atomic proposition, with the sub-formula placed in hashtable. After the substitution is applied, we obtain the following flat formula: $AG(p \Rightarrow exp1)$ (Figure 5.a). A hashtable entry is created, with $exp1$ being the key and the sub-formula being the value. This procedure is recursive and starts from the atomic sub-formulas and is build upwards. For TCTL, an atomic sub-formula is a TCTL formula that does not contain nested path-specific operators or quantifiers over paths. The flattened expression is then passed to the parser engine together with the hashtable that contains the mappings.

For parsing the TCTL expressions into formulas, the parsing engine assumes the following order of operators (AG, AF, A, U, W, =>, ||, &&, !, ==, <>, +, -, *, /), from the weakest to the strongest binding. This means that arithmetic operators have the strongest binding, followed by logical operators, with path-specific temporal operators and the path quantifiers having the weakest binding.

A flattened TCTL formula has the following properties: it contains at most one path-specific temporal operator, at most one path quantifier and one or more logical operators. In some cases, path-specific temporal operators and the path quantifiers are considered as one operator in the binary tree (ex: AG, AF), whereas in the case of U and W operators they are always treated separately. Once the temporal operators have been identified and parsed, the logical formula is then transformed according to the predefined ordered list of logical operators and arithmetic ones. Everything else is considered to be an atomic proposition.

When the parser encounters an atomic proposition, it looks in the hashtable to check whether it is an original one, or it corresponds to a mapping introduced by the flattening procedure. In the given example, `p` is an atomic proposition from the original formula, whereas the proposition `exp1` corresponds to a mapping. Once the mapping is identified, the original sub-formula is retrieved from the hash-table and the procedure continues with the sub-formula being treated as a new property. The sub-tree that the sub-formula generates is then simply appended to the original tree by replacing the mapping leaf node (Figure 5.b). The procedure terminates once all the mappings from the hashtable have been transformed.

### 5.1.2 FormulaTransformer

The second component in the `SMTLibReq` library is the `FormulaTransformer` that parses the binary tree formula into a format that can be used as a direct input into an SMT solver. Similar to the `ExpressionParser`, the `FormulaTransformer` component has its basic functionality customizable for particular formalisms. The customization depends on two factors: first, the semantics and the syntax of the input formula (the parsing engine operates on a predefined set of operators and their semantics), and second, the format of the output, which can for instance be the general SMT-LIB, a Z3 Python script, or something else. For the sake of simplicity, in this section, we describe a transformation procedure of a TCTL binary-tree formula into SMT-LIB, which is the same as any other transformation procedure except that it "understands" the semantics of the TCTL and FOL operators only, thus is capable of generating an SMT-LIB script that can be used as an input to state-of-the-art SMT solvers.

The main procedure of transforming the binary-tree formulas into an SMT-LIB script involves two main activities: i) creating assertions that describe the constraints encoded in the formula, and ii) creating the declarations that define the variables that are used within the model.

The parsing of the binary-tree formula is performed top-to-bottom starting from the root node of the tree. The transformation for both the operators and the atomic propositions is quite similar. For processing each node, be it internal or leaf, the procedure analyzes the expression and loads an appropriate template used to instantiate the expression into an SMT-LIB construct. Each template is composed of literal and non-literal parts, with the literal parts being fixed, and the non-literal parts being replaced by the formula-specific expressions.

Once the template has been loaded, the non-literal parts of the template are instantiated by the template-instantiation function. In order to illustrate the parsing procedure, we recall the same example as previously, i.e. we apply the transformation on the binary-tree given in Figure 5.b.

The transformation starts from the root node, which in this case contains the *AG* operator. The template selection engine analyzes the operator expression and determines that it should be transformed using the quantifier template, given as:

`(#quantifier# (#quantifiedVariable#) (#expression#))`.

The expressions surrounded by hash symbols represent the non-literal terms of the template. The loaded template then represents the basis for the template instantiation functionality that determines the non-literal terms and substitutes them in the template. For the *AG* expression, the template instantiation engine determines that the `#quantifier#` will be replaced with a *universal quantifier* over a real-valued

variable that models the notion of time. To be able to determine the formula specific value of the `#expression#` part, the parsing calls the parsing procedure for the child nodes of the current one. All the subsequent operator nodes are transformed in a similar fashion. The recursive transformation stops at the leaf nodes. Since the leaf nodes contain the atomic propositions, when such a node is parsed, a declaration is additionally created for the variables defining that particular atomic expression. For that purpose, the engine determines the type of the variable (which in SMT-LIB can be either a constant or a function) and its inputs and output and creates an appropriate declaration for it. Once the complete formula structure has been traversed, the assertion is added to the set of assertions and the set of declarations produced by the formula is added to the set of declarations of the script.

After the complete formula from Figure 5.b has been completed, we obtain the following assertion:

```
(forall ((time Real)) (implies (= (p time) 1.0)
(exists ((t1 Real)) (and (>= t1 time) (< t1
(+ time T)) (not (= (q t1) (r t1)))))))
```

accompanied by the following declarations:

```
(declare-const T Real)
(declare-fun p (Real) Real)
(declare-fun q (Real) Real)
(declare-fun r (Real) Real)
```

# 6. CONSISTENCY ANALYSIS OF FLD REQUIREMENTS USING Z3

In this section, we describe the process of consistency analysis of the SMT-LIB assertions generated during the previous step.

In order to be able to analyze the FLD requirements, we integrate the `SMTLibReq` library into the `SeSAMM Specifier`, which is possible due to the modular design of the `PROPAS` (see Section 5). Then, one can automatically generate an SMT-LIB script that corresponds to the complete set of requirements for the FLD system. An excerpt of the script that corresponds to the requirements presented in Section 4.1 is given as follows:

```
(assert (!  (forall (time Real) (=> (= (aPB time)
0.0) (<= (iFV time) (aFV time)))) :named SG))

(assert (!  (forall ((time Real)) (=> (and (=
(CAN2 time) (DD time)) (not (= (DD time) ERR)))
(or (exists ((t1 Real)) (and (> t1 time) (<= t1
(+ time 100)) (not (= (CAN2 t1) (DD t1))) (not
(exists ((t2 Real)) (and (> t2 time) (<= t2 t1)
(not (= (iFV t2) (DD t2))))))))(not (exists ((t1
Real)) (and (> t1 time) (<= t1 (+ time 100)) (not
(= (iFV t1) (DD t1)))))))))) :named FSRICL))

(assert (!  (forall ((time Real)) (=> (= (Dma-
cEnableCh time chU32) 1.0) (= (dmacCH time chID)
1.0))) :named SSR1DMAC))

(assert (!  (forall ((time Real)) (=> (= (DmacDis-
ableCh time chU32) 1.0) (= (dmacCH time chID)
0.0)))) :named SSR2DMAC)
```

By analyzing the assertions generated based on the requirements, we notice that the analysis process can be additionally optimized by encoding the domain knowledge not explicitly captured by the requirements. An example of such information is the fact that the fuel level cannot be less than zero or greater than the tank size. Below, we show one such assertion, `actualFuelBound`, which bounds the value of the `actualFuelVolume` parameter to a range of allowed values used in the *FSR_ICL* requirement of Section 3.

```
(assert (!  (forall ((time Real)) (and (>= (aFV
time) 0) (<= (aFV time) TANK_SIZE)) :named actual-
FuelBound)))
```

For performing the SMT analysis, we use an instance of the Z3 SMT solver configured as follows:

```
(set-option :mbqi True)
(set-option :mbqi-max-iterations 1000)
(set-option :pull-nested-quantifiers True)
(set-option :unsat-core True)
```

Four out of five TCTL patterns include implication, so they can be trivially satisfied if the antecedent evaluates to false. For example, *FSR_ICL* is trivially satisfied if `CAN2(time) = DD(time)` never evaluates to true. To prevent the trivial satisfaction of the requirements, we explicitly instruct the solver to check for satisfiability when all of the antecedents hold. The hidden problem of using this technique may arise from requirements that model complementary behavior, so enabling of the antecedents must be performed carefully in order to avoid false positive inconsistencies.

The SMT-LIB script containing 36 assertions that correspond to the FLD requirements has been analyzed using the Z3 tool on a Linux machine with 2.4 GHz Dual Core processor and 4GB RAM. Using the unbounded model-based quantifier instantiation (mbqi) the procedure does not terminate within 48 hours, whereas bounding the procedure to a maximum of 1000 runs for generating the model yields the verdict *UNKNOWN*. To determine the cause of non-termination of the SMT analysis, we incrementally insert the requirements one by one, into the solver, and perform the consistency analysis on every step. In this way, we are able to isolate the requirements for which the SMT procedure cannot terminate. By applying this strategy, we discover two classes of requirements: the ones for which the SMT procedure terminates (*solvable*) and the ones for which it does not, called *non-solvable*. In the following, we discuss the characteristics of both classes and the mitigation strategy used for the non-solvable ones.

**Solvable Requirements.** The requirements formalized by instantiating the patterns *P1*, *P2* and *P3*, which represent 73% of the total requirements (see Figure 3) do not hinder termination of the SMT analysis process; an input script constructed exclusively from such requirements is analyzed within seconds. This shows that the tool can handle pattern instances with a maximum of two nested quantifiers without difficulty. Pattern P1 contains only one universal quantifier

(encoded as `ForAll(time)` in Z3), while P2 and P3 have two levels of nested quantifiers of the following types
`(forall ((time Real)) (forall ((t1 Real))))` or
`(forall ((time Real)) (exists ((t1 Real))))` .
For optimization reasons, the nested universal quantifier is converted into an existential one by using the conversion rule: $\forall x : p(x) \iff \neg\exists x : \neg p(x)$.

**Non-solvable Requirements.** The patterns *P4* and *P5* covering 27% of total requirements (see Figure 3), prevented termination of the SMT procedure. Compared to the patterns in the *solvable* category, *P4* and *P5* have a more complex structure, arising from the nested TCTL formula of the $W$ operator, which is translated into two levels of nested quantifiers. When the $W$ operator is used within an invariant property (e.g. P4 in Section 4.1), an additional universal quantifier is created, thus yielding three levels of nested operators.

**Mitigating Non-solvable Requirements.** In order to tackle the requirements formalized using patterns *P4* and *P5*, one of the nested quantifiers must be eliminated. By analyzing the semantics of the patterns, we find that an additional existential quantifier is added to model the sporadic events. These quantifiers can be eliminated by converting the sporadic events into periodic, that is, by providing a witness valuation from the set of allowed values. For illustration, we apply this technique on the $\text{FSR}_{ICL}$ requirement (see Section 4.1). The original requirement captures the sporadic occurrence of the event CAN2 = DD. By applying our mitigation strategy, we modify the requirement such that the given event occurs every 100 time units after the antecedent is satisfied. In the TCTL form, we replace the $\text{W}_{\leq 100}$ with $\text{U}_{=100}$, which results in the following formula:

$AG(CAN2 = DD \wedge DD \neq ERR \Rightarrow iFV = DD\, U_{=100}\, CAN2 \neq DD)$.

This model is pessimistic but still valid, since once a witness is found, the satisfaction of the original formula follows.

After applying the mitigation technique, the SMT analysis over the complete set of FLD system requirements returns *SAT* accompanied by a valid model within. To validate that our approach can detect temporal inconsistencies, we perform controlled faulty assertions injection. Examples of such assertions include: enabling requirements expressing mutually exclusive behaviors ($\text{SSR}^1_{DMAC}$ and $\text{SSR}^2_{DMAC}$) at the same time, or assertions that violate existing ones. All of the injected faults have been detected by Z3, and the conflicting assertions (requirements) contained in the minimal inconsistent set have been generated by the solver using the `unsat-core` command.

# 7. DISCUSSION

In this section, we reflect on the advantages and limitations of the tool, based on the results obtained after the consistency analysis of the case-study requirements.

In its current state, the `SMTLibReq` library provides the necessary means for transforming the temporal formulas encoded in TCTL into SMT-LIB constraints suitable for consistency analysis using any state-of-the-art SMT solver. Despite the fact that the TCTL specifications are generated using patterns, the implemented transformation procedure is more general and can be applied on any valid TCTL formula, be it part of the original set of patterns or not. This feature enables us to expand the set of patterns used for formalizing the requirements, while not jeopardizing the correctness of the final result. However, one should note that the current version was tested over the patterns from the FLD specification only, so evaluation with other patterns or arbitrary TCTL formulas may lead to situations that the tool cannot handle.

For determining the type of the variables and the return type of the functions in the system specification, `SMTLibReq` implements a simple *type-inferring* mechanism, which assigns type `Real` to the functions and variables in the system specification. Despite the fact that such an approach represents an over approximation (the boolean and integer values are subsets of the real-valued variables) it is less efficient. For example, if there is a boolean variable in the system and it is declared as real-valued one, the SMT solver in the background must call the arithmetics engine to solve this constraint, whereas in reality it could have been done on a propositional level. We are already working on improving the type-inferring mechanism and the improved version is expected to be released in the subsequent updates of the library.

Last but not least, for the formal specification of free-text requirements (Step 1), we relied on our existing tool, called `SeSAMM Specifier`. For confidentiality reasons, the `SeSAMM Specifier` cannot be shared with the community for further evaluation and usage. To circumvent this limitation, as a direction for future work we aim at developing a `PROPAS` user interface that can be distributed freely as an open source project. As for now, it seems that a tool, which would be a simplified version of the existing `SeSAMM Specifier` without the privacy-protected features, is the way to go. With the introduction of this feature, the `PROPAS` tool will become more accessible to a broader audience, but it will also open a door of possibilities for integrating other features that will make it evolve into a more comprehensive framework for specification, analysis and verification of industrial embedded systems.

# 8. RELATED WORK

Various approaches for checking requirements consistency, based on different definitions of consistency and different analysis techniques, have been proposed in the literature.

A consistency checking procedure similar to ours has been proposed by Barnat et al. [3]. The authors define a *model-free* sanity-checking procedure including consistency for system requirements specifications in Linear Temporal Logic (LTL) by means of model checking. The notion of consistency is reduced to checking whether an automaton obtained as a conjunction of all the formulas in the specification has a non-empty accepting language. The same has later been extended [2] to be able to generate a minimal inconsistent set of requirements. The approach relies on tool support similar to our `PROPAS`, which uses specification patterns for formal system specification and a parallel LTL model-checker called DiVinE [4]. Despite the exhaustiveness, the approach suffers

from the inherent complexity of transforming the LTL formulas into automata, especially for large systems, thus potentially making its application in industrial contexts challenging. A similar approach for consistency checking of requirements specified in LTL is proposed by Ellen et al. [12]. The paper presents a so-called existential definition, that is, the existence of at least one run of the system that satisfies the complete set of requirements - which is an approach close to ours. For performing the consistency checking, the authors use bounded model checking using the iSAT SMT solver [11]. The proposed technique is capable of generating a maximal set of consistent requirements, as well as a minimal inconsistent subset of requirements. Similar to our approach, the tool has been integrated into the existing tool called BTC Embedded Specifier [6], which is a proprietary software that relies on their tool-specific specification patterns.

The work by Post et al. [24] defines the notion of rt-(in)consistency of real-time requirements. The notion covers cases where the requirements in the system's requirements specification can be inconsistent due to timing constraints. The checking for rt-inconsistency is reduced to model checking. Compared to our approach, the generation of the formalism that is subjected for consistency analysis is performed manually.

The notion of consistency is also checked for requirements specified in domain-specific notations. Heimdhal and Leveson [18] provide an approach for consistency analysis for requirements specified in RSML (Requirements State Machine Language). The proposed definition for consistency is suitable only for requirements specified in RSML and is not applicable for requirements expressed in any other notation. Real-time embedded systems can also be specified using the Software Cost Reduction (SCR) method. The SCR method is suitable for specifying both functional and extra-functional system requirements. A complete suite for analyzing system specifications in SCR has been developed by Heitmeyer et al. [19]. The suite provides tools for requirements specification, symbolic execution and formal analysis.

Despite the fact that the approaches above [2] [3] [24] can exhaustively check for the consistency of requirements specifications, all of them suffer from one major limitation, which is the verification time that grows exponentially with the number of requirements. In the early phases of system requirements specification, a more lightweight and considerably faster procedure as proposed in this paper might be more suited. Hence, our method can be used as a complementary approach to the above listed methods for consistency checking. Another important aspect of our approach is the fact that the complete procedure is completely automated and the formal system specification is performed in a user-friendly manner using the specification patterns.

# 9. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an automated solution for SMT-based consistency analysis of formal requirements specifications encoded as Timed CTL formulas. The proposed solution is supported by our tool called `PROPAS`, which completely automates the requirements transformation and consistency analysis procedures.

The implementation of the `PROPAS` tool provides a *"push-button-analysis"* approach, meaning that the complete process of transformation and ultimately the analysis of the requirements is completely hidden from the users. The generation of the analyzable format is performed according to well-justified abstraction rules that simplify the original formulas to ensure their analyzability while assuring the preservation of information that could contribute to potential inconsistencies. Our initial validation of the tool on an industrial case, the Fuel Level Display from Scania, shows its potential for consistency checking of running industrial systems. However, the full potential of the tool needs to be tested on more complex and larger system specifications.

Given the current status of the underlying SMT-based consistency analysis methodology and its automation though the `PROPAS` tool, there are several directions for future research. The SMT-based methodology can be improved in several ways, including: proposing a more general definition of consistency, improving the encoding in order to speed up the analysis procedure, which could possibly lead to the tool being able to analyze other classes of formal properties also. Moreover, we have to address the missing features and the limitation of the current version of the `PROPAS` tool, discussed in Section 7. The fact that the transformation process is completely automated opens up the possibility to validate the tool at a more extensive scale, on larger industrial systems. To meet such a goal, we have already started preparing future case studies, based on running industrial examples to further investigate the boundaries of applicability of our tool and method.

## Acknowledgments

This work has been funded by the Swedish Governmental Agency for Innovation Systems (VINNOVA) under the VeriSpec project 2013-01299.

# 10. REFERENCES

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, pages 2–34, 1993.

[2] J. Barnat, P. Bauch, N. Beneš, L. Brim, J. Beran, and T. Kratochvíla. Analyzing Sanity of Requirements for Avionics Systems. *Form. Asp. Comput.*, 28(1):45–63, Mar. 2016.

[3] J. Barnat, P. Bauch, and L. Brim. Checking Sanity of Software Requirements. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 48–62, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] J. Barnat, L. Brim, M. Ceska, and P. Rockai. Divine: Parallel distributed model checker. In *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, pages 4–7. IEEE, 2010.

[5] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department

of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[6] BTC Embedded Systems AG. BTC Embedded Validator Pattern Library, Release 3.6, 2012.

[7] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM*, 54(9):69–77, Sept. 2011.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-state Verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, FMSP '98, pages 7–15, New York, NY, USA, 1998. ACM.

[10] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[11] A. Eggers, N. Kalinnik, S. Kupferschmid, and T. Teige. Challenges in constraint-based analysis of hybrid systems. *CSCLP*, 5655:51–65, 2008.

[12] C. Ellen, S. Sieverding, and H. Hungar. *Detecting Consistencies and Inconsistencies of Pattern-Based Functional Requirements*, pages 155–169. Springer International Publishing, Cham, 2014.

[13] P. Filipovikj. PROPAS. `https://github.com/predragf/propas`, 2017.

[14] P. Filipovikj, T. Jagerfield, M. Nyberg, G. Rodríguez-Navas, and C. Seceleanu. Integrating Pattern-Based Formal Requirements Specification in an Industrial Tool-Chain. In *40th IEEE Annual Computer Software and Applications Conference, COMPSAC Workshops 2016, Atlanta, GA, USA, June 10-14, 2016*, pages 167–173. IEEE Computer Society, 2016.

[15] P. Filipovikj, M. Nyberg, and G. Rodriguez-Navas. Reassessing the pattern-based approach for formalizing requirements in the automotive domain. In *Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE)*, volume 00, pages 444–450, Los Alamitos, CA, USA, 2014. IEEE Computer Society.

[16] P. Filipovikj, G. Rodriguez-Navas, M. Nyberg, and C. Seceleanu. SMT-based Consistency Analysis of Industrial Systems Requirements. In *The proceedings of the 32nd ACM Symposium on Applied Computing (SAC). Marrakech, Morocco.* ACM, April 2017.

[17] L. Grunske. Specification Patterns for Probabilistic Quality Properties. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 31–40, New York, NY, USA, 2008. ACM.

[18] M. P. E. Heimdahl and N. G. Leveson. Completeness and Consistency in Hierarchical State-Based Requirements. *IEEE Trans. Softw. Eng.*, 22(6):363–377, June 1996.

[19] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Transactions Software Engineering Methodology*, 5(3):231–261, July 1996.

[20] J.-P. Katoen. *Concepts, Algorithms and Tools for Model Checking.* 1999.

[21] S. Konrad and B. H. C. Cheng. Real-time Specification Patterns. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 372–381, New York, NY, USA, 2005. ACM.

[22] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[23] C. M. Papadimitriou. *Computational complexity.* Addison-Wesley, Reading, Massachusetts, 1994.

[24] A. Post, J. Hoenicke, and A. Podelski. Rt-inconsistency: A New Property for Real-time Requirements. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*, FASE'11/ETAPS'11, pages 34–49, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] A. Post, I. Menzel, J. Hoenicke, and A. Podelski. Automotive Behavioral Requirements Expressed in a Specification Pattern System: A Case Study at BOSCH. *Requir. Eng.*, pages 19–33, 2012.

[26] J. Westman and M. Nyberg. Contracts for Specifying and Structuring Requirements on Cyber-Physical Systems. In D. B. Rawat, J. Rodriques, and I. Stojmenovic, editors, *Cyber Physical Systems: From Theory to Practice.* CRC Press, 2015.

## ABOUT THE AUTHORS:

Predrag Filipovikj is a PhD student at Mälardalen University, Sweden since January 2014. He holds MSc degrees in Software Engineering from Mälardalen University, Sweden, and Computer Networks and e-Technologies from Ss. Cyril and Methodius University, Macedonia, respectively, and a Licentiate degree in Technology from Mälardalen University, Sweden. His research is focused on the application of formal methods for increasing the quality of cyber-physical systems, with particular focus on the automotive domain. His active topics of research include: engineer-friendly formal system specification and analysis, and formal verification of design-time models mostly by model checking. He performs research in cooperation with Scania AB CV and Volvo Groups Truck Technology, both from Sweden. He is a student member of ACM and IEEE, and a member of the Swedish Requirements Engineering Research Network.

Guillermo Rodriguez-Navas received the telecommunication engineer degree from the University of Vigo, Spain, in 2001, and the doctorate in informatics from the University of the Balearic Islands (UIB), Spain, in 2010. At present, he is senior lecturer at the School of Innovation, Design and Engineering of the Mälardalen University, Sweden. He has authored or co-authored +60 peer-reviewed scientific papers and is currently co-supervising three doctoral dissertations. His research interests include dependability and safety aspects of complex embedded systems, fault tolerance for distributed embedded systems, clock synchronization, scheduling algorithms for real-time time-triggered networks, and application of formal verification techniques to requirements engineering and system specification, particularly in the domains of automotive and cyber-physical systems.

Mattias Nyberg is an adjunct (part-time) professor at Royal Institute of Technology (KTH) in the department of Mechatronics. His main affiliation is Scania CV AB, a leading global heavy-truck manufacturer. He received a PhD in Electrical Engineering from Linköping University in 1999 specializing in vehicular systems. After dissertation he has worked mainly in industry; first for Daimler in Stuttgart, Germany, with diesel engine diagnosis, and later at Scania with diagnosis and functional safety. In parallel with his industrial career, he is very active in academic research. He has supervised six PhD students in the area of diagnosis and functional safety. He is also an author of more than 100 scientific publications, and has received the SAE Vincent Bendix award for the best paper of year 2015 in the area of automotive electronics engineering.

Docent Cristina Seceleanu is Associate Professor at Mälardalen University, Sweden, Embedded Systems Division, and leader of the Formal Modeling and Analysis of Embedded Systems research group. She has a MSc. in Electronics (Polytechnic University of Bucharest, Romania) and a Ph.D. in Computer Science (Åbo Akademi, Finland). Her research focuses on developing formal models and verification techniques for predictable real-time and adaptive embedded systems. She has been involved in several national and European research projects (FP7, ARTEMISIA, AAL), out of which she is currently leading three. She has served in the editorial boards of several journals such as Frontiers in ICT: Formal Methods, and International Journal of Embedded and Real-Time Communication Systems (IJERTCS), as well as in the PC of about 100 conferences in the field.