SMT-based Consistency Analysis of Industrial Systems Requirements

Predrag Filipovikj[1], Guillermo Rodriguez-Navas[1], Mattias Nyberg[2], Cristina Seceleanu[1]
[1]Mälardalen University, Västerås, Sweden
{predrag.filipovikj, guillermo.rodriguez-navas, cristina.seceleanu}@mdh.se
[2]Scania, Södertälje, Sweden
mattias.nyberg@scania.com

ABSTRACT

As the complexity of industrial systems increases, it becomes difficult to ensure the correctness of system requirements specifications with respect to certain criteria such as consistency. Automated techniques for consistency checking of requirements, mostly by means of model checking, have been proposed in academia. However, such approaches can sometimes be costly in terms of modeling and analysis time or not applicable for certain types of properties. In this paper, we present a complementary method that relies on pattern-based formalization of requirements and automated consistency checking using the state-of-the-art SMT tool Z3. For validation, we apply our method on a set of timed computation tree logic requirements of an industrial automotive subsystem called the Fuel Level Display.

CCS Concepts

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

Keywords

System Requirements; Specification Patterns; TCTL; Consistency Analysis; SMT; Z3

1. INTRODUCTION

The costs associated to the late detection of errors in the requirements specifications of industrial systems are typically high, as their mitigation might call for the redesign or reimplementation of certain parts of the system. For this reason, industry has high demands for techniques that provide early debugging of system specifications. This paper tackles the problem of detecting *inconsistencies* within system specifications, which occurs whenever the set of requirements is not realizable as such, due to internal contradictions. We propose a method for consistency checking of requirements specifications, starting from their description in natural language. We apply patterns to formalize the textual requirements, after which we employ Satisfiability Modulo Theories (SMT) [6] to check the formalized requirements by using the Z3 SMT solver [5]. The proposed method consists of four steps, is shown in Figure 1.

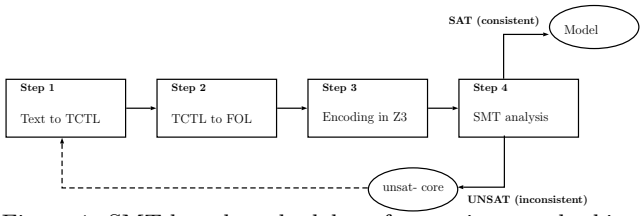

Figure 1: SMT-based methodology for consistency checking of requirements specifications.

In *Step 1*, the system requirements are specified in Constrained Natural Language (CNL) via the Specification Pattern System (SPS) [7] [15]. Then, the CNL requirements are automatically encoded into temporal formulas, expressed in (Timed) Computational Tree Logic (TCTL) [1]. Next, in *Step 2*, the (T)CTL patterns are transformed into First-Order Logic (FOL) formulas by instantiating the semantics of temporal and path operators. In *Step 3*, the FOL formulas are encoded into Z3 assertions, which are later optimized for analyzability by using a number of abstraction rules. The traceability between the steps is ensured by assigning unique identifiers to the requirements during Step 1, which remain the same throughout the process. Our Z3 encoding preserves the relevant information of the natural language requirements, and complies with the technical limitations of the analysis tool. Finally, in *Step 4*, SMT analysis is conducted, resulting in a verdict that states whether the system requirements specification is realizable as such. 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.

Our method can be seen as a complementary approach to the already existing consistency analysis techniques based on model checking [8] [13] [18]. Moreover, in this paper we aim

SAC 2017, April 03-07, 2017, Marrakech, Morocco

http://dx.doi.org/10.1145/3019612.3019787

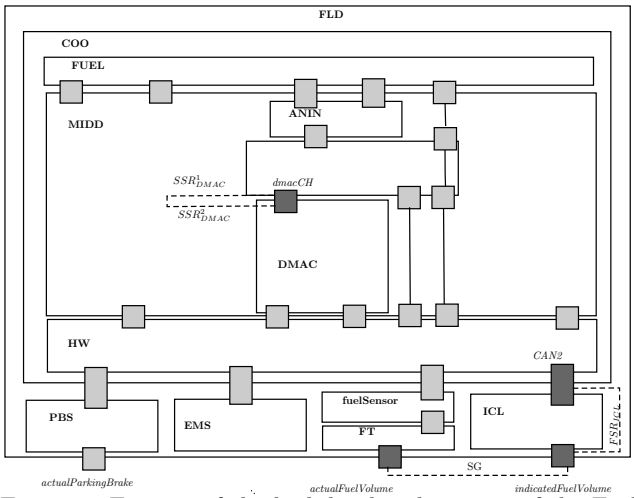

Figure 2: Excerpt of the high-level architecture of the Fuel Level Display system

to check the feasibility of the method for industrial systems requirements. Therefore, we illustrate the applicability of our approach on a real case study from the heavy-load vehicle domain, namely the Fuel Level Display system of a vehicle.

The paper continues as follows: in Section 2 we describe the Fuel Level Display system, whereas in Section 3 we introduce the concepts used in the paper. Steps 1, 2, 3 and 4 are discussed in Sections 4, 5, 6 and 7, respectively. We compare to related work in Section 8, and conclude the paper in Section 9 as well as outline future research directions.

2. INDUSTRIAL CASE-STUDY: FUEL LEVEL DISPLAY

In this section, we introduce the industrial system called Fuel Level Display (FLD) on which we apply our method.

The FLD system is a function installed in all heavy-load vehicles produced by Scania, Sweden. The core functionality of the FLD is to correctly estimate the remaining fuel in the vehicle and display the value to the driver. The fuel estimation feature is implemented as a software function deployed on the Coordinator (COO) Electronic Control Unit (ECU) system. The remaining fuel is calculated based on of the sensed fuel level obtained from the fuel tank provided by the fuel sensor (fuelSensor) placed inside the fuel tank (FT) and the current fuel consumption rate provided by the Engine Management ECU system (EMS). The system is classified as *safety critical*, meaning that its proper functioning must be ensured at all times, otherwise it may lead to hazardous situations that can endanger human lives.

A simplified architectural break-down of the FLD system, including all fore mentioned parts is given in Figure 2. The design description is based on the concept of *element*, which represents an extension over Heterogeneous Rich Components [20]. The elements represented as rectangles in Figure 2 (ex: FLD, PBS, etc.) are used to model all entities in the system's design description, including both physical and logical ones. The interface of an element is represented via one or more *ports* (ex: actualParkingBrake, actualFuelVolume, etc.), which are used to model the tangible entities of the element as seen by an external observer, that is, from the environment. The communication between different elements is via their ports. The behavior of an element is defined through a set of constraints over its ports. The requirements for the FLD system are specified using the contract-based approach, through assertions of type *assumption-guarantee*, represented with dashed lines in Figure 2. A subset of the FLD requirements that are used as the working example in the rest of the paper includes the following:

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.

The FLD system has been developed by a small team of engineers physically located at the same site, who have followed an agile development process that revolves around an evolving prototype supported by obscure informal documentation. The comprehensive documentation of the system, including the requirements specification, is created only later in the development to serve as a basis for verification. For the FLD system, the requirements specification exists only for the higher levels of abstraction, with no documentation describing the concepts on lower abstraction levels. For our study, the requirements specification at this level has been reverse engineered from the various behavioral models or the production code. Consequently, the system requirements specification embeds implementation details such as Dmac_enableCh() (see SSR^1_{DMAC}). For the detailed functional description of the system we redirect the readers to other work [20].

3. PRELIMINARIES

In this section, we introduce the concepts that are used in the rest of the paper. First, we briefly recall the temporal logic used for encoding the system specification. Second, we give the formal definition of consistency; finally we present an overview of the SMT concept and the Z3 tool.

3.1 (Timed) Computation Tree Logic

Computation Tree Logic (CTL) is a branching time logic [14] used for the formal specification of finite-state systems. The interpretation of CTL is defined over a model M that

consists of a non-empty set of states S, a successor relation R and a labeling function *Label* that assigns a set of atomic propositions to each state in the model. Timed CTL (TCTL) is an extension of CTL for specifying real-time systems [1]. Time is a non-negative real-valued variable manipulated by clock formulas expressing constraints over 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"*) represents the basic path operator, which is combined with either of the two path quantifiers. The "until" operator serves at defining all other temporal operators. The F (*Future*) operator denotes a formula that eventually becomes true ($F\varphi \Leftrightarrow true\, U\, \varphi$), while the G (*Globally*) operator denotes that a given formula is valid in all states along a given path ($G\varphi \Leftrightarrow \neg F \neg \varphi$) [1]. There exists a weaker version of the U operator called *"weak-until"* denoted as W, which is used to capture formulas where the right hand side term might never be satisfied. The semantics of W is defined as: $\varphi\ W\ \psi \equiv (\varphi\ U\ \psi) \vee G\varphi$.

In TCTL, each of the path-specific temporal operators has a timed version that uses clock constraints. In this paper, we use the following syntax to express the timed operators: $Oper_{\sim T}$, where Oper $\in$ {U, F, G, W}; $\sim \in \{=, <, >, \leq, \geq\}$ and T is a numeric bound of clock variables. For instance, the formula $EF_{\leq T}\varphi$ requires that there exists an execution path along which φ eventually becomes true within T time units. For more details we refer the reader to previous work [1] [14].

3.2 Formal Definition of Consistency

Let us assume that the system requirements specification has been formally encoded, that is, represented as a set of logical formulas. For such specification we consider the following definition of inconsistency:

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. We say that the set* Φ *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 which satisfies the conjunction of all the formulas. Checking whether there exists an interpretation that satisfies the Boolean conjunction given above represents a classical Boolean satisfiability (SAT) problem.

3.3 Satisfiability Modulo Theories and Z3

The SAT problem requires to determine if a Boolean formula can be made true by assigning true/false values to the constituent Boolean variables. The solution to the SAT problem is a model containing the values of the variables in the formula. Satisfiability Modulo Theories (SMT) is an extension of SAT where the interpretation of some symbols is constrained by a background theory [6]. One such example is the theory of arithmetic that restricts the interpretation of symbols to: {+, -, ≤, 0, 1}.

For performing SMT analysis we use the Z3 tool [5] developed by Microsoft, which is a state-of-the-art SMT solver and theorem prover widely used in academia. 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 [3] or alternatively by using the various APIs for common programming languages such as C#, Python, Java, etc. Declarations can be either constants or functions. Constants are functions with arity 0, while functions are n-ary. 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. Z3 supports two types of quantifiers: universal (`ForAll`) and existential (`Exists`). The tool also provides a library with a number of tactics, which can be used to optimize the decision procedures.

The command `check-sat` determines whether the current formulas on the Z3 stack are satisfiable or not. If the formulas are satisfiable, Z3 returns *SAT* that in our case proves the analyzed consistency. If they are unsatisfiable, Z3 returns *UNSAT* thus proving that the set of requirements is inconsistent. Z3 may also return *UNKNOWN* if it cannot determine whether a formula is satisfiable or not. When the command `check-sat` returns *SAT*, the 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 tool can generate the minimal inconsistent set of formulas by calling the `unsat-core` command.

4. REQUIREMENTS SPECIFICATION: TEXTUAL TO TCTL

The set of FLD requirements are originally specified in free text using a general purpose text editor. Such specifications are readable and expressive, yet sometimes ambiguous and definitely not amenable to automated analysis. The first step towards a framework for automated analysis and verification is to convert the natural language specification into a formal counterpart. In this section, we describe Step 1 of the method proposed in Figure 1. We use TCTL to formally express the system requirements. Such decision is motivated by the fact that TCTL is suitable for capturing time-constrained requirements, which are within those that the FLD system is required to fulfill. Moreover, a simplified subset of TCTL can be used for model checking eventual system realizations, for instance by using UPPAAL [16].

To be able to automate the process of formalizing the requirements, we use the Specification Pattern System (SPS) [7] [15]. SPS represents a catalog of reusable solutions called *patterns*, for specifying reactive and real-time systems requirements, expressed in different formal notations (e.g., LTL, (T)CTL, MTL, etc.) but also in CNL. The method has been shown expressive enough to formalize requirements of automotive systems [10] [19]. We have formalized the complete set of FLD requirements, by using our SESAMM Specifier tool [9]. Below, we express (via SPS) the requirements from Section 2 first in CNL and

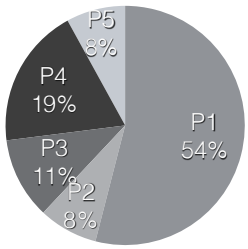

Figure 3: Pattern distribution for the formalized FLD requirements

then in TCTL, as follows:

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

$\mathbf{FSR_{ICL}}$ After CAN2 = DD & DD ≠ ERR holds until CAN2 ≠ 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.

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

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

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

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

The formalization results show that only five patterns are enough to generate the complete set of FLD requirements in TCTL. The list below contains the used patterns (P1 to P5), while the percentage of FLD requirements covered by each pattern is given in Figure 3. Our findings are aligned with the earlier formalization attempts [10] [19], which reveal 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 φ Until θ Universally ψ: $AG(\varphi \Rightarrow A(\psi\, W_{\leq T}\, \theta))$

P5: Timed After φ Until θ Universally ψ: $AG(AG_{\leq T}(\varphi) \Rightarrow A(\psi\, W_{\leq T}\, \theta))$

5. REQUIREMENTS SPECIFICATION: TCTL TO FOL

In this section, we present the transformation of the TCTL patterns into FOL formulas. The importance of this transformation is twofold: i) to bridge the semantic gap between TCTL and Z3, and ii) to ensure the preservation of information between the two. This activity is captured by Step 3 in Figure 1. Due to space limitations 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 TCTL to FOL transformation is carried out by instantiating the semantics of the TCTL operators according to the definitions given by Katoen [14], assuming a timed transition system as the underlying semantic model of our system. The semantics uses the following concepts: σ denotes a single path from the set of all paths $P_M(s)$ starting from a given state s. A position in the path is a pair (i, d_i), where i is the location number, whereas d_i is the time delay, that is, a real number that corresponds to the time elapsed during the delay transitions; a set of such points characterizes the states traversed along σ while going from state s_i to the successor s_{i+1} for any $i \in \mathbb{N}$. $Pos(\sigma)$ denotes the set of all positions in a given path σ. The time elapsed on a path σ from the initial state s_0 to any state s_i is defined as:

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

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

We denote a clock valuation by v. Based on this, the action $(reset\ z\ in\ v)$ is defined 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 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 in [14].

Lemma 1 (P1, P4 into FOL)**.** *Given a transition system M, predicates φ, ψ, θ, s a state of M, and ω a clock valuation formula, the following two equivalences hold:*

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

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

Proof:

(2)
$s, \omega \models AG_{\geq 0}(\varphi \Rightarrow A(\psi\, W_{\leq T}\, \theta))$
$\Leftrightarrow$ {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 φ}
$s, reset\ z\ in\ \omega \models AG_{\geq 0}(\neg\varphi \vee A((\psi\ \wedge z \leq T)\, U\, \theta) \vee \neg E(True\, U$
$(\neg\psi \wedge z \leq T)))$
$\Leftrightarrow$ {Definition of $AG_{\geq T}, A(\varphi\, U\, \psi), E(\varphi\, U\, \psi)$}
$\forall\sigma \in P_M(s).(\forall(i,d) \in Pos(\sigma).\sigma(i,d), (reset\ z\ in\ \omega)+$
$\Delta(\sigma, i) \models (\neg\varphi \vee (\forall\sigma' \in P_M(\sigma(i,d)).(\exists(j,d')>>(i,d)$
$\in Pos(\sigma').\sigma'(j,d'), (reset\ z\ in\ \omega) + \Delta(\sigma', j) \models (\theta \wedge z \leq T)$
$\wedge (\forall(k,d'')<<(j,d') \in Pos(\sigma').\sigma'(k,d''), (reset\ z\ in\ \omega)$
$+ \Delta(\sigma', k) \models (\psi \wedge z \leq \Delta(\sigma', j)))) \vee \neg(\exists\sigma' \in P_M(\sigma(i,d)).$
$(\exists(j,d')>>(i,d) \in Pos(\sigma').\sigma'(j,d'), (reset\ z\ in\ \omega)+$
$\Delta(\sigma', j) \models (\neg\psi \wedge z \leq T) \wedge (\forall(k,d'')<<(j,d') \in Pos(\sigma').$
$\sigma'(k,d''), (reset\ z\ in\ \omega) + \Delta(\sigma', k) \models True))))))$
$\Leftrightarrow$ {Logic, definition of total order, semantics of reset z in ω}
$\forall\sigma \in P_M(s).(\forall(i,d) \in Pos(\sigma).\sigma(i,d), (z = \Delta(\sigma, i)) \models$
$(\neg\varphi \vee (\forall\sigma' \in P_M(\sigma(i,d)).(\exists(j,d').(j>i \vee (j = i \wedge d \leq d'))$
$\in Pos(\sigma').\sigma'(j,d'), (z = \Delta(\sigma', j)) \models (\theta \wedge z \leq T)) \wedge$
$(\forall(k,d'').(k<j \vee (k = j \wedge d'' \leq d')) \in Pos(\sigma').\sigma'(k,d''),$
$(z = \Delta(\sigma', k)) \models (z<\Delta(\sigma', j) \wedge \psi))) \vee (\forall\sigma' \in P_M(\sigma(i,d)).$
$(\forall(j,d').(j>i \vee (j = i \wedge d \leq d' \leq d+T) \in Pos(\sigma').$
$\sigma'(j,d'), (z = \Delta(\sigma', j)) \models (\psi \wedge z \leq T)))))))$

□

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

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

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

$P5: \forall\sigma \in P_M(s).(\forall(i,d) \in Pos(\sigma).\sigma(i,d), (z = \Delta(\sigma, i)) \models$
$(\neg(\forall\sigma' \in P_M(\sigma(i,d)).(\forall(j,d').(j>i \vee (j = i \wedge d \leq d' \leq T)$
$\in Pos(\sigma').\sigma'(j,d'), (z = \Delta(\sigma', j)) \models$
$(z \leq T \wedge \varphi))) \vee (\forall\sigma' \in P_M(\sigma(i,d)).(\exists(j,d').(j>i \vee (j = i \wedge d \leq d')$
$\in Pos(\sigma').\sigma'(j,d), (z = \Delta(\sigma', j)) \models (z \leq T \wedge \theta))) \wedge (\forall(k,d'')$
$\in Pos(\sigma').(k<j \vee (k = j \wedge d'' \leq d') \in Pos(\sigma').(\sigma'(k,d''),$
$(z = \Delta(\sigma', k)) \models$
$(z \leq \Delta(\sigma', j) \wedge \psi))) \vee (\forall\sigma' \in P_M(\sigma(i,d)).(\forall(j,d').(j>i \vee$
$(j = i \wedge d \leq d' \leq d+T)) \in Pos(\sigma').\sigma'(j,d), (z = \Delta(\sigma', j)) \models$
$(z \leq T \wedge \psi)))))$

Returning to Definition 1, by instantiating the above patterns in FOL, one obtains the conjunction of all requirements that need to be fed to Z3 in order to check consistency.

6. REQUIREMENTS ENCODING IN Z3

FOL formulas can be abstracted as Z3-compliant formulas, analyzable with Z3, by applying these three encoding rules:

R1: Directly map the FOL constructs into Z3 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 (σ) 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 the rules R1, R2 and R3 on the set of patterns of Section 5 can be illustrated as follows:

$$P_i \xrightarrow{R1,R2} P_i' \xrightarrow{R3} P_{i\,Z3},\ i \in [1,5]$$

The application of rule R1 results in a Z3 script where each assertion corresponds to an individual requirement, with quantifiers and Boolean expressions encoded in Z3. The traceability between the requirements and the Z3 assertions is ensured via unique identifiers. Recall that TCTL formulas are interpreted over a branching model (Section 3.1), so the assertions are quantified over the following 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 negative impact the decidability of the SMT procedure [17]. Our initial attempts to verify the directly mapped Z3 assertions failed due to the three quantified variables mentioned above. To remedy this, we propose an abstraction technique (rules R2 and R3) that reduces the number of quantified variables in the assertions, while abstracting away only the information related to variables that cannot be sources of requirements inconsistency (e.g., σ, and i). Further, we collect location and time variables into a tuple position, denoted by pos [14]. To access the location component of the position we write $pos.i$. Similarly, time valuation in that position is obtained by $pos.d$.

The abstraction relies on the interpretation of TCTL over a model, according to which the progress in the system happens by changing position on a given path, either by making a transition from one location to another or by delaying in a given location [4]. In our case, 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 (chosen via the "select" operator) suffices. Consequently, the quantified path disappears in our Z3 encoding.

All our patterns rely on semantic models in which progress is ensured by discrete transitions (in no time), in which location index ($pos.i$) increases, or via delay transitions, in which the time distance from the initial location ($pos.d$) increases. The progress along the path is modeled by the binary operator "<<" that compares positions, defined as: "$pos << pos' \iff pos.i < pos'.j \vee pos.i = pos'.j \wedge pos.d < pos'.d$". Possible inconsistencies can arise from contradicting formulas that should hold in/from each position, e.g. φ, ψ 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:

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

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

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

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

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

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 Z3 assertions that contain only one quantified variable, ($pos.d$) that is represented by "`time`" Z3 quantified variable. Also, the predicates (φ, ψ, θ) in the FOL patterns are substituted with Boolean expressions over the system ports represented as functions over time denoted as $var_i(time)$, $i \in \mathbb{Z}$. The complete set of patterns expressed in Z3 is presented in the list below:

$P1_{Z3}$ $ForAll(time, \varphi(time) == val_1)$

$P2_{Z3}$ $ForAll(time, Implies(Implies(var_1(time) == val_1,$
$Not(Exists(time_1, And(var_1(time) \neq val_1, time_1 \geq time,$
$time_1 \leq time + T))), var_2(time) == val_2))$

$P3_{Z3}$ $ForAll(time, Implies(var_1(time) == val_1, Exists(time_1,$
$And(time_1 > time, time_1 < time + T, var_2(time) == val_2))))$

$P4_{Z3}$ $ForAll(time, Implies(var_1(time) == val_1, Or(Exists(time_1,$
$And(time_1 \geqslant time, time_1 \leqslant time + T, var_2(time_1) == val_2),$
$Not(Exists(time_2, And(time_2 \geqslant time, time_2 \leqslant time_1,$
$var_2(time_2) == val_2, var_3(time_2) \neq val_3))))), Not(Exists($
$time_1, And(time_1 \geqslant time, time_1 = time + T, var_2(time_1) \neq$
$val_2)))))$

$P5_{Z3}$ $ForAll(time, Implies(Implies(var_1(time) \neq val_1, Not(Exists($
$time_1, And(time_1 \geq time, time_1 \leq time + T, var_1(time_1) \neq$
$val_1))))), Or(Not(Exists(time_2, And(time_2 \geq time + T, time_2$
$\leq time + 2T, var_2(time_2) \neq val_2))), Exists(time_2,$
$And(time_2 \geq time + T, time_2 \leq time + 2T, var_3(time_2) == val_3,$
$Not(Exists(time_3, And(time_3 \geq time + T, time_3 < time_2,$
$var_3(time_3) == val_3, var_3(time_3) \neq val_3)))))))))$

It is obvious that the original set of requirements expressed via patterns P1,..., P5 are stronger than their counterparts encoded in Z3. Therefore, theoretically, proving the inconsistency of the encoded versions means proving the inconsistency of the original ones, but a 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 the Z3 ones are satisfiable, based on our argumentation around the only possible sources of inconsistency.

7. CONSISTENCY ANALYSIS IN Z3

By applying the Z3 patterns of Section 6 we obtain the set of assertions to be analyzed for consistency. The analysis process can be additionally optimized by encoding the domain knowledge that is not explicitly stated into the requirements. Typical cases are the assertions encoding physical concepts and limitations of the system, such as the actual fuel level that cannot be less than zero or greater than the tank size, or the speed that can be reached by the vehicle. The code that follows shows an example of such optimization denoted by assertion `actualFuelBound` that bounds the value of the `actualFuelVolume` parameter to a set of allowed values used in the *FSR_ICL* requirement of Section 2.

```
FSR_ICL = ForAll(time, Implies(And(CAN2(time) ==
        DD(time), DD(time) ≠ ERR), Or(Exists(time1,
        And(time1 ≥ time, time1 ≤ time + T,
        CAN2(time1) == DD(time1), DD(time1) == ERR,
        Not(Exists(time2, And(time2 ≥ time, time2 ≥
        time1, CAN2(time2) == DD(time2), DD(time2)
        ≠ ERR, iFV(time2) ≠ DD(time2)))))))))

actualFuelBound = ForAll(time, And(aFV(time) ≥ 0,
        aFV(time) ≤ TANK_SIZE))
```

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

```
solver = Then('smt','simplify', 'qe').solver()
        solver.set('mbqi', True)
        solver.set('mbqi.max_iterations', 1000)
        solver.set('pull_nested_quantifiers', True)
        solver.set('unsat_core', True)
```

Four out of five TCTL patterns include implication, which 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 eliminate trivial satisfaction, we explicitly instruct the solver to check for satisfiability when all of the antecedents hold, being careful not to enable requirements that model complementary behavior at the same time, to avoid false positive inconsistencies.

The SMT procedure for the whole set of FLD requirements that includes 36 Z3 assertions is run on a Linux machine with 2.4 GHz Dual Core processor and 4GB RAM. The procedure using the unbounded model-based quantifier instantiation (mbqi) does not terminate within 48 hours, whereas bounding the iterations (up to 1000) of the mbqi procedure 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; a Z3 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, ForAll(time`$_1$`))` or `ForAll(time, Exists(time`$_1$`))`. For optimization reasons, the nested universal quantifier is converted into an existential one by using the conversion rule: $\forall x : p(x) \Longleftrightarrow \neg\exists x : \neg p(x)$.

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 in Z3. If the W operator is used within an invariant property (e.g. P4 in Section 4), an extra universal quantifier is created, yielding three levels of nested operators. Even when the optimization is applied, that is, conversion of universal into existential quantifiers, such Z3 assertions cannot be solved by the tool as such.

Mitigating Non-solvable Requirements. In order to tackle the requirements formalized using patterns P4 and P5, one of the nested quantifiers must be eliminated. To determine which of the nested quantifiers is to be eliminated, we analyze the semantics of the patterns. We find that an additional existential quantifier is added to model the sporadic occurrence of events, over a variable bounded from above. Such existential quantifiers can be eliminated by providing a witness value from the set of allowed values. In this way, all the sporadic events in the requirements are modeled as periodic with their period equal to the upper bound of the allowed interval. For illustration, let us recall the FSR_{ICL} requirement (see Section 4) that 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 $\text{FSR}_{ICL} : AG(CAN2 = DD \wedge DD \neq ERR \Rightarrow iFV = DD\, U_{=100}\, CAN2 \neq DD)$. 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* with a valid model. The procedure terminates successfully within seconds on a standard workstation computer as described in the text above. Since the consistency analysis is performed on an operational industrial system, the *SAT* verdict is expected. To validate that our approach can detect temporal inconsistencies, we have deliberately injected faulty assertions. Examples of such assertions include: enabling requirements expressing mutually exclusive behaviors (SSR^1_{DMAC} and 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.

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. [13]. The authors define a *model-free* sanity-checking procedure including consistency for system requirements specification in Linear Temporal Logic (LTL) by means of model checking. The notion of consistency is reduced to checking whether an automaton A 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. Despite the exhaustiveness, the approach suffers from the inherent complexity of transforming the LTL formulas into automata, especially for complex systems, potentially making it unusable in industrial settings. A similar approach for consistency checking of requirements specified in LTL is proposed by Ellen et al. [8]. 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. 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 analysis procedure has been tailored for an industrial application.

The work by Post et al. [18] 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 constrains. The checking for rt-inconsistency is reduced to model checking.

The notion of consistency is also checked for requirements specified in domain-specific notations. Heimdhal and Leveson [11] 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. [12]. The suite provides tools for requirements specification, symbolic execution and formal analysis.

Despite the fact that the approaches above [2] [13] [18] 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 approach can be used as a complementary approach to the above listed methods for consistency checking.

9. CONCLUSIONS

In this paper, we have presented an SMT-based method for consistency checking of system requirements specifications, which, without demanding a behavioral system model, checks whether the systems' specification is realizable. Unlike previous work mentioned in Section 8, our method is suitable for early debugging of system specifications. The application of our method prevents the propagation of inconsistencies into the subsequent artifacts such as models and code. It can also be used in more agile methods to increase the confidence in the correctness of the specification, thus reducing the likelihood of flawed verification results. Additionally, the proposed technique can be applied when the specification contains complicated temporal properties with nested path and temporal operators that cannot be verified using model checking.

To make SMT-based analysis possible, our method provides means for transforming textual requirements into the input language of the Z3 SMT solver input, using temporal logics and FOL as intermediate representations to bridge the semantic gap. Our findings show that, due to their complexity, patterns with more than two levels of nested quantifiers cannot be analyzed using existing SMT solvers. We propose a mitigation strategy for three levels of nested quantifiers that, based on the existence of a witness, eliminates the innermost quantifier. The strategy relies on a straightforward transformation from sporadic to periodic events, which is possible due to the timed nature of the analyzed systems (e.g. FLD). For more complex cases, appropriate witness-selection heuristics should be investigated.

SMT-based approaches are suited for checking realizability, i.e. the existence of a model that satisfies the system specification. For instance, a system specified as: $\mathrm{AG}(\varphi) \wedge \mathrm{EF}(\neg\varphi)$ is not realizable. This could be regarded as a narrow definition of inconsistency. For detecting more subtle inconsistencies in the specification, a broader definition coupled with more exhaustive analysis should be applied. Consequently, the proposed method cannot replace the model-checking-based consistency analysis; rather, it should be used as a complementary technique for detecting errors early on, hence reducing verification efforts in later phases.

At present, we are working towards automating the complete method, including the formalization, transformation and analysis steps, to minimize the need for user interaction. By providing automated tool support, we seek for more industrial penetration and analysis of new case studies.

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. Analysing sanity of requirements for avionics systems. *Formal Aspects of Computing*, pages 1–19, 2016.

[3] 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`.

[4] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, pages 244–263, 1986.

[5] L. De Moura and N. Bjørner. Z3: An efficient smt solver. TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] L. De Moura and N. Bjørner. Satisfiability modulo theories: Introduction and applications. volume 54, pages 69–77, New York, NY, USA, Sept. 2011. ACM.

[7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99*, pages 411–420. ACM, 1999.

[8] C. Ellen, S. Sieverding, and H. Hungar. Detecting consistencies and inconsistencies of pattern-based functional requirements. In *FMICS*, volume 8718 of *LNCS*, pages 155–169. Springer, 2014.

[9] P. Filipovikj, T. Jagerfield, G. Rodriguez-Navas, M. Nyberg, and C. Seceleanu. Integrating pattern-based formal requirements specification in an industrial tool-chain. In *QUORS*, pages 167–173, 2016.

[10] P. Filipovikj, M. Nyberg, and G. Rodriguez-Navas. Reassessing the pattern-based approach for formalizing requirements in the automotive domain. In *RE'14*, pages 444–450, 2014.

[11] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, 1996.

[12] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications, 1996.

[13] P. B. Jiri Barnat and L. Brim. Checking sanity of software requirements. In *SEFM 2012, LNCS 7504*, pages 48–62, 2012.

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

[15] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *ICSE '05*, pages 372–381. ACM, 2005.

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

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

[18] A. Post, J. Hoenicke, and A. Podelski. rt-inconsistency: a new property for real-time requirements. In *FASE 2011*, number 6603 in LNCS, pages 34–49. Springer, 2011.

[19] 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.

[20] 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.