

Completeness and Consistency of Tabular Requirements: an SMT-Based Verification Approach

Claudio Menghi, Eugene Balai, Darren Valovcin, Christoph Stickse, Akshay Rajhans

Abstract—Tabular requirements assist with the specification of software requirements using an “if-then” paradigm and are supported by many tools. For example, the Requirements Table block in Simulink® supports writing executable specifications that can be used as test oracles to validate an implementation. But even before the development of an implementation, automatic checking of consistency and completeness of a Requirements Table can reveal errors in the specification. Fixing such errors earlier than in later development cycles avoids costly rework and additional testing efforts that would be required otherwise. As of version R2022a, Simulink® supports checking completeness and consistency of Requirements Tables when the requirements are stateless, that is, do not constrain behaviors over time. This paper (i) formally defines the syntax and semantics of Requirements Tables and their completeness and consistency, (ii) proposes eight encodings from two categories (namely, bounded and unbounded) that support stateful requirements, and (iii) implements THEANO, a solution supporting checking completeness and consistency using these encodings. We empirically assess the effectiveness and efficiency of our encodings in checking completeness and consistency by considering a benchmark of 160 Requirements Tables for a timeout of two hours. Our results show that THEANO can check the completeness of all the Requirements Tables in our benchmark, it can detect the inconsistency of the Requirements Tables, but it can not confirm their consistency within the timeout.

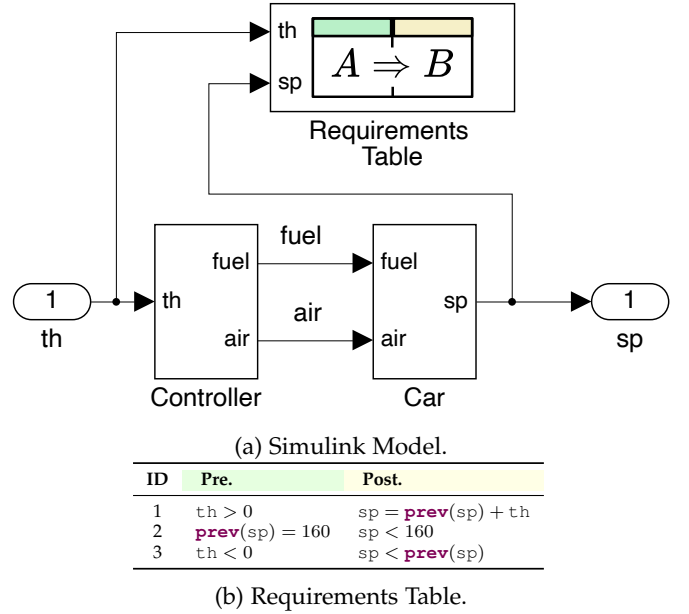


Figure 1: Illustrative Example.

1 INTRODUCTION

Requirements typically start in a textual form and are often captured informally in natural language specifications. Formalization of specifications, e.g., using logic formulas [1], [2], [3], [4], [5] or temporal assessments [6] from Simulink® Test™ [7] ensures precise and unambiguous semantics that can be leveraged for automated analysis. *Tabular requirements specifications* (e.g. [8], [9], [10]) support specification and analysis of requirements in a table format. They have been extensively studied by the academic community (e.g., [11], [12], [13], [14], [15]), and used to model the software requirements of industrial systems (e.g., [16], [17]). Tabular requirements have been used for requirements specification of the space shuttle [16] and the shutdown system of the Darlington Canadian Nuclear Generation Station [17].

In this paper, we focus on the Requirements Table block [18] from Requirements Toolbox™ [19] which makes

it possible to author formal specifications as part of a *specification model*. Figure 1b presents a Requirements Table for an illustrative block diagram model of a vehicle from Figure 1a. The Requirements Table specification consists of a set of requirements, each represented by one row of the table, consisting of an identifier (ID), a precondition (**Pre.**), and a postcondition (**Post.**). In addition to each individual requirement row meant to be satisfied by the model, a table format further specifies that the *set* of three requirements are together satisfied.

Checking for the *completeness and consistency* of Requirements Tables helps gaining confidence in the correctness of a set of requirements. Intuitively, tabular requirements specifications are *complete* if they describe the system behavior for every possible input; they are *consistent* if the requirements do not force the system to behave in two different (and contradictory) ways for the same input [20]. Analysis tools for tabular requirements specifications help software engineers check the consistency and completeness of requirements (e.g., [13], [21]). For example, Simulink supports

C. Menghi is with the University of Bergamo, Bergamo, Italy and the McMaster University, Hamilton, Canada - e-mail: {claudio.menghi@unibg.it} E. Balai, D. Valovcin, C. Stickse, and A. Rajhans are with MathWorks, MA, USA - email: {ebalai,dvalovci,cstickse,arajhans}@mathworks.com

identifying inconsistent and incomplete requirements [22] among other checks under some syntactic conditions on the requirements.

Flexible requirements enable engineers to avoid the specification of exact values in the postcondition of a requirement. For example, it is common to provide only a range of values, which allows multiple implementations that are equally valid. A requirement that gives only concrete values in its postcondition is called *rigid*.

While Simulink can analyze many flexible requirements for consistency and completeness, some constructs can only be analyzed when the requirement is rigid.

Stateful requirements use the previous value of a variable (a.k.a. “previous state”) in the precondition of a requirement. This enables specifying temporal behaviors of a system.

Simulink and many other tools have only limited support for the consistency and completeness analysis of stateful requirements, since reasoning about temporal behaviors introduces additional complexity.

For example, the first requirement from the Requirements Table in Figure 1b is a stateful requirement, since, when the precondition is satisfied ($t_h > 0$), its postcondition ($sp = \text{prev}(sp) + t_h$) increments the value of the variable (sp) from its previous value ($\text{prev}(sp)$) by a value equal to the value of the throttle (t_h). That is, the value a variable assumes depends on the value assumed by a variable in the previous step (a.k.a. state). The second requirement from the Requirements Table in Figure 1b is flexible since, when the precondition is satisfied ($\text{prev}(sp) = 160$), its postcondition ($sp < 160$) does not specify the exact value the speed variable (sp) should assume.

In practice, requirements often contain at least some number of flexible requirements with state. The availability of tools to analyze such requirements would enable earlier validation of requirements, possibly leading to better requirements, which in turn may result in obtaining safer implementations faster.

This paper provides the first solution to address the challenge of analyzing the completeness and consistency of Requirements Tables containing flexible requirements with states. To reach this goal, the contributions of this paper are as follows:

(i) A formal definition of the *syntax* and *semantics* of Requirements Tables (Section 3). This definition precisely specifies when a trace (recording the values assumed by the system variables over time) satisfies or violates the conditions described by a Requirement Table and enables rigorously defining the notions of completeness and consistency. In our definitions, we considered traces with variable and fixed sample steps, which change and maintain constant the step size between records and are generated by variable-step and fixed-step solvers [23].

(ii) A formal definition of the notions of *completeness* and *consistency* (Section 4). We illustrated our definitions with a set of practical examples. We also show that checking the completeness and consistency of Requirements Tables is an undecidable problem.

(iii) Eight complementary encodings to *check the incompleteness and inconsistency* of Requirements Tables (Section 5). Our encodings rely on translating Requirements Tables into

logic-based formulae checked for satisfiability. We defined four bounded and unbounded encodings, respectively.

(iv) The *implementation of our solution* into THEANO, a tool that checks the completeness and consistency of Requirements Tables (Section 6). Our tool relies on two components: a MATLAB (R2023a) script that extracts the requirements from the Requirements Tables, an antlr [24] (v4.5) component that translates the requirements into a logical formula executable from a Python script that uses the Z3Py [25] Python API to run the SMT solver Z3 [26]. We made our implementation publicly available [27].

(v) An extensive *empirical evaluation* (Section 7). Our empirical evaluation assesses the effectiveness and efficiency of our encoding in checking the completeness and consistency of Requirements Tables. This assessment is performed by considering a benchmark of 160 Requirements Tables. Our results show that the encodings from the unbounded category were effective in detecting complete Requirements Tables within two hours, while they failed to report on their incompleteness. For the consistency check, the encodings from the unbounded category failed to report on the consistency and inconsistency of the Requirements Tables of our benchmark within two hours. Unlike the encodings from the unbounded category, when provided with a sufficiently large bound value,¹ the encodings from the bounded category were effective and efficient in checking the completeness and incompleteness of all the Requirements Tables within two hours. For the consistency check, our encodings were effective in detecting inconsistent Requirements Tables within two hours, while they failed to report on their consistency.

(vi) An extensive *discussion and reflection* on the practical implications of our results and threats to validity (Section 8).

This paper is organized as follows. Section 2 provides background notions related to Simulink models and fixed and variable step traces. Section 3 presents the syntax and semantics of Requirements Tables. Section 4 formally defines the notions of completeness and consistency. Section 5 presents our encodings to check the completeness and consistency of Requirements Tables. Section 6 describes the implementation of THEANO. Section 7 evaluates our contribution. Section 8 discusses our results and threats to validity. Section 9 discusses related work. Section 10 presents our conclusion and future work.

2 BACKGROUND

In this section we present required preliminaries about Simulink necessary for presenting the rest of the paper.

2.1 Simulink

Model-Based Design (MBD) is a process that enables early and continuous validation with opportunities for formal methods to increase trust in the final product. A central artifact in MBD is a computational model of the system that can be analyzed on a computer. A simulation environment such as Simulink enables faster exploration of system behaviors for fast design iterations [28]. The model typically

1. Reflections about the selection of the value for the bound are presented in Section 8.

starts at a high level of abstraction and gets refined to increasing fidelity. By decomposing the model into units, a team of developers can collaborate, or delegate part of the implementation to a third party. A key element of MBD is automated code generation that frees the user from having to write code and interfaces themselves, allowing them to focus on algorithms and applying their domain knowledge at a higher level of abstraction. This increases productivity and eliminates sources of errors.

Our illustrative example concerns a simple controller (Figure 1) developed using Simulink for a vehicle. The controller monitors the values of input variables and acts on output variables. We use \mathcal{U} and \mathcal{Y} to indicate the input and output variable sets. The controller from Figure 1a has one input variable, the throttle (th), and one output variable, the vehicle's speed (sp). The controller monitors the throttle (th) applied by the user and regulates the speed (sp) of the vehicle by changing the amount of fuel (fl) and air (air) of the engine.

Figure 1b presents the Requirements Table specifying the requirements for the controller. These requirements are only used for illustration purposes and do not aim to be representative and realistic. The three requirements are identified with identifiers (ID) 1, 2, and 3. Requirement 1 specifies that the speed value shall increment from its previous value ($\text{prev}(\text{sp})$) by the value of the throttle (th) when the value of the throttle (th) is higher than zero. Requirement 2 specifies that the next speed value (sp) shall be lower than that threshold when its previous value equals the threshold value of 160km/h. Finally, requirement 3 specifies the speed value shall decrease when the value of the throttle (th) is lower than zero.

Since the requirements are stateful and flexible, Simulink cannot analyze their completeness and consistency. We propose a solution that addresses this limitation and considers Simulink models executed with fixed or variable sample step solvers.

2.2 Fixed and Variable Step Traces

With the requirements expressed as an executable specification model, the design model can be constructed. This can follow Model-Based Design principles and use simulation and validation. Once the design model is complete, we can use the executable specification in the Requirements Table to act as a test oracle to judge whether an observed behavior of the design model is allowed by the specification.

Given a representation of an input trace, the design model can be run in conjunction with the Requirements Table in Simulink as shown in Figure 1a. The result of this simulation is an output trace capturing how the values of output variables change over time, as well as a judgement whether the design model is within the specification at each point in time during the simulation.

Let \mathbb{N} be the set of integer numbers, \mathbb{R} be the set of real numbers, \mathbb{B} be the set of Boolean values, \mathbb{N}^0 be the set of non-negative integers numbers, \mathbb{R}^0 be the set of non-negative real numbers and $\mathbb{D} = \mathbb{N} \cup \mathbb{R} \cup \mathbb{B}$. We use $\text{dom}(u)$ and $\text{dom}(y)$ to indicate the domain of an input $u \in \mathcal{U}$ and output $y \in \mathcal{Y}$ variable.

Definition 1 (Trace). A trace π is a tuple $\langle \iota_{\mathcal{U}}, \iota_{\mathcal{Y}}, \iota_{\tau} \rangle$, where the input $\iota_{\mathcal{U}} : \mathbb{N}^0 \times \mathcal{U} \rightarrow \mathbb{D}$ and output $\iota_{\mathcal{Y}} : \mathbb{N}^0 \times \mathcal{Y} \rightarrow \mathbb{D}$ interpretations respectively associate each input $u \in \mathcal{U}$ and output $y \in \mathcal{Y}$ variable and position $i \in \mathbb{N}^0$ with a value $\iota_{\mathcal{U}}(i, u)$ and $\iota_{\mathcal{Y}}(i, y)$ from their domain $\text{dom}(u)$ and $\text{dom}(y)$. The timestamp interpretation $\iota_{\tau} : \mathbb{N}^0 \rightarrow \mathbb{R}$ associates each position i to a timestamp value. \square

Figure 2 presents two examples of traces for the model from Figure 1a. The traces show how the values of the input (th) and the output (sp) variables change over time. The function ι_{τ} associates each position with a timestamp. For example, for the trace in Figure 2a, $\iota_{\tau}(3) = 0.6\text{s}$ and $\iota_{\tau}(5) = 1.0\text{s}$. The functions $\iota_{\mathcal{U}}$ and $\iota_{\mathcal{Y}}$ associate a value for each timestamp position and variable. For example, for the trace in Figure 2a, $\iota_{\mathcal{U}}(3, \text{th}) = 5$ and $\iota_{\mathcal{Y}}(5, \text{sp}) = 140.4$. We use the symbol Π to denote the universe of all the traces. A trace record refers to the values assumed by the timestamp and the input and output variables at a specific position of the trace. For example, the trace records r_3 of the two traces in Figure 2 are delimited by a square frame. When we refer to a generic variable v , that can refer both to an input or an output variable, we use $\iota(i, v)$ to indicate its interpretation in position i , that is $\iota(i, v) = \iota_{\mathcal{U}}(i, v)$ when v is an input variable, and $\iota(i, v) = \iota_{\mathcal{Y}}(i, v)$ when v is an output variable. For example, for the trace in Figure 2a, the interpretation of the variable th at position 3 is $\iota(3, \text{th}) = \iota_{\mathcal{U}}(3, \text{th}) = 5$.

Definition 2 (Fixed and Variable Step Traces). A trace π has a fixed sampling time T_s if for every index $i \in \mathbb{N}^0$, $\iota_{\tau}(i+1) - \iota_{\tau}(i) = T_s$. A trace with a fixed sampling time is a fixed step trace. Otherwise, the trace is a variable step trace since it has a variable sampling time. \square

The trace from Figure 2a has a fixed sampling time since for all index $i \in \mathbb{N}^0$, $\iota_{\tau}(i+1) - \iota_{\tau}(i) = 0.2$. The trace from Figure 2b has a variable sampling time since the difference (0.7) between the timestamp (0.9) in position 2 and the timestamp (0.2) in position 1 does not correspond with the difference (0.9) between the timestamp (1.8) in position 3 and the timestamp (0.9) in position 2.

The semantics of Requirements Tables can be defined by considering fixed and variable step traces as detailed in the following section.

3 REQUIREMENTS TABLES: SYNTAX AND SEMANTICS

We present the syntax (Section 3.1) and semantics (Section 3.2) of Requirements Tables.

3.1 Syntax

A Requirements Table is defined over a set of variables $\mathcal{V} = \mathcal{U} \cup \mathcal{Y}$, where \mathcal{U} and \mathcal{Y} are the sets of the input and output variables.² We use u , y , and v to denote input and output variables from \mathcal{U} and \mathcal{Y} , and a generic variable from \mathcal{V} , respectively. We use $\text{dom}(v)$ to indicate the domain of the values for variable v .

2. Requirements Tables can also consider internal variables (local and state variables) by adding additional output variables to the Simulink model that carry the value of the internal variables and using these variables within the Requirements Tables.

th	20	15	10	5	-5	2	-1
sp	140	144	147	149	140	140.4	140
τ	0	0.2	0.4	0.6	0.8	1.0	1.2
i	0	1	2	3	4	5	6

Record r_3

(a) Fixed Step Trace.

th	20	15	10	5	-5	3	-1
sp	140	144	154.5	163.5	169.5	165	167.4
τ	0	0.2	0.9	1.8	3.0	4.9	5.7
i	0	1	2	3	4	5	6

Record r_3

(b) Variable Step Trace.

Figure 2: Two examples of a trace.

$t ::= c \mid v \mid \text{prev}(v) \mid t_1 \odot t_2$
 $le ::= t_1 \oplus t_2 \mid \neg le \mid \text{dur}(le) \geq c \mid le_1 \oslash le_2$
 $pre ::= le$
 $post ::= le$
 $req ::= pre[d] \Rightarrow post$
 $rt ::= req \mid req, rt$
 $c \in \mathbb{R}, v \in \mathcal{V}, \odot \in \{+, -, *, /\}, \oplus \in \{>, <, \leq, \geq, =, \neq\}, \oslash \in \{\wedge, \vee, \Rightarrow\}.$

Figure 3: Requirements Tables: Syntax.

Figure 3 presents the grammar used in this work to define a Requirements Table. A *term* t is either a real constant (i.e., c), an input or output variable (i.e., v), the term $\text{prev}(u)$ indicating the previous value of the input variable u , or a combination $t_1 \odot t_2$ of two terms t_1, t_2 with an arithmetic operator $\odot \in \{+, -, *, /\}$.

A logical expression le is either the combination $t_1 \oplus t_2$ of two terms t_1 and t_2 with a relational operator $\oplus \in \{>, <, \leq, \geq, =, \neq\}$, the negation $\neg le$ of a logical expression le , the duration term $\text{dur}(le) \geq c$ indicating that a logical expression le holds for at least c seconds, or the combination $le_1 \oslash le_2$ of two logical expressions obtained by considering the Boolean operators $\oslash \in \{\wedge, \vee, \Rightarrow\}$.

Preconditions pre and postconditions $post$ are logical expressions. A requirement req is in the form $pre[d] \Rightarrow post$ and relates a precondition pre with a postcondition $post$. Requirements Tables also provide an optional duration column (d). Unlike the duration term, the duration column expresses the time (in seconds), during which the precondition must be true before evaluating the rest of the requirement. A Requirements Table (rt) is a requirement (req) or a requirement (req) concatenated with a Requirements Table (rt).

With a slight abuse of notation, we use the symbols $rt, req, post, pre, le, t$ also to indicate instances of Requirements Tables, requirements, preconditions, postconditions, logical expressions, and terms created following the syntax presented in Figure 3.

Figure 4 reports a graphical representation of a Requirements Table. Each subfigure represents one Requirements Table containing two requirements. Requirements are identified by an index (ID labeled column) followed by the pre (Pre. labeled column) and post-conditions (Post. labeled column).

We introduce notation that will be used to define the semantics of Requirements Tables (Section 3.2) and the notions of completeness and consistency (Section 4). We use $vars(post)$ to denote the set of variables occurring

in $post$. We use $Pre(rt, y)$ to represent the preconditions of a Requirements Table rt that are implicants of a postcondition ($post$) defined using the output variable y (i.e., $y \in vars(post)$). That is, $Pre(rt, y)$ is the set of preconditions $\{pre \mid pre \Rightarrow post \in rt \text{ and } y \in vars(post)\}$.

3.2 Semantics

We first define (Definition 3) when a position (i) of a trace (π) satisfies (the requirements of) a Requirements Table (rt). Then, we define (Definition 4) when a trace (π) satisfies a Requirements Table (rt).

Definition 3 (Semantics). Let $rt, \pi = \langle \iota_u, \iota_y, \iota_\tau \rangle$, and $i \geq 0$ respectively be a Requirements Table, a trace, and a position. The satisfaction $i, \pi \models rt$ of the Requirements Table rt in position i of the trace π is recursively defined in Figure 5.

The semantics from Figure 5 defines the semantics of the different operators of the Requirements Tables. For the operator $\text{dur}(le)$ and the requirement expressed using the duration column (d), which requires considering the values assumed by the timestamp values, we defined two semantics: the fixed (Figure 5b) and the variable (Figure 5c) step semantics that respectively refer to fixed and variable step traces. We describe the semantics of each operator informally in the following.

The operator \vdash specifies the interpretation of the terms in different positions of a trace π . For a position i , the value of the constant is the interpretation of the term c . The interpretation of a variable v at position i is value $\iota(i, v)$. For example, for Figure 2a, the interpretation of the variable th at position 3 is $\iota(3, th) = 5$. The interpretation of the term $\text{prev}(v)$ at position i is the value $\iota(i - 1, v)$ assumed by the variable v in the previous position $i - 1$ of the trace, or the $\iota(0, v)$ the initial value assigned to the variable if the position is 0. For example, for Figure 2a, the interpretation $\text{prev}(th)$ of the variable th at position 3 is $\iota(2, th) = 10$. Requirements Tables also enable engineers to leave the initial value assigned to a variable undefined.³ For simplicity, in this work, we assume that every variable is assigned to an initial value. The interpretation of $t_1 \odot t_2$ is obtained by applying the arithmetic operator \odot to the interpretations of t_1 and t_2 . For example, for Figure 2a, the interpretation of $th + sp$ at position 3 is 154, that is the sum of the interpretation of the input variable th (i.e., 5) and the interpretation of the output variable sp (i.e., 149) at position 3.

3. <https://www.mathworks.com/help/slrequirements/ref/getprevious.html>.

ID	Pre.	Post.	ID	Pre.	Post.	ID	Pre.	Post.	ID	Pre.	Post.
1	$\text{dur}(\text{th} > 3) \geq 0.4$	$\text{sp} = 1$	1	$\text{dur}(\text{th} > 3) > 0.4$	$\text{sp} = 1$	1	$\text{dur}(\text{th} > 3) \geq 0.4$	$\text{sp} = 1$	1	$\text{dur}(\text{th} > 3) > 0.4$	$\text{sp} = 1$
2	$\text{dur}(\text{th} > 3) \leq 0.5$	$\text{sp} = 2$	2	$\text{dur}(\text{th} > 3) < 0.4$	$\text{sp} = 2$	2	$\text{dur}(\text{th} > 3) \leq 0.5$	$\text{sp} = 2$	2	$\text{dur}(\text{th} > 4) \leq 0.5$	$\text{sp} \geq 1$

(a) Example 1. (b) Example 2. (c) Example 3. (d) Example 4.

Figure 4: Examples of Requirements Tables.

$i, \pi \vdash c$	$= c$
$i, \pi \vdash v$	$= \iota(i, v)$
$i, \pi \vdash \text{prev}(v)$	$= \begin{cases} i > 0 & \iota(i-1, v) \\ i = 0 & \iota(i, v) \end{cases}$
$i, \pi \vdash t_1 \odot t_2$	$= (i, \pi \vdash t_1) \odot (i, \pi \vdash t_2)$
$i, \pi \models t_1 \oplus t_2$	$\text{iff } (i, \pi \models t_1) \oplus (i, \pi \models t_2)$
$i, \pi \models \neg \text{le}$	$\text{iff } i, \pi \not\models \text{le}$
$i, \pi \models \text{le}_1 \odot \text{le}_2$	$\text{iff } (i, \pi \models \text{le}_1) \odot (i, \pi \models \text{le}_2)$
$i, \pi \models \text{pre} \Rightarrow \text{post}$	$\text{iff } (i, \pi \models \text{pre}) \Rightarrow (i, \pi \models \text{post})$
$i, \pi \models \text{req}, \text{rt}$	$\text{iff } (i, \pi \models \text{req}) \wedge (i, \pi \models \text{rt})$

(a) Semantics for the operators not based on timestamp values.

$i, \pi \models \text{dur}(\text{le}) \geq c$	$\text{iff } (\iota_\tau(i) \geq c) \wedge (\forall k \in \mathbb{N}^+.$ $\quad (\iota_\tau(i) - \iota_\tau(i-k) \leq c_r \Rightarrow (k, \pi \models \text{le})))$
$i, \pi \models \text{pre}[d] \Rightarrow \text{post}$	$\text{iff } ((\iota_\tau(i) \geq c) \wedge (\forall k \in \mathbb{N}^+.$ $\quad (\iota_\tau(i) - \iota_\tau(i-k) \leq d_r \Rightarrow (k, \pi \models \text{pre}))))$ $\Rightarrow i, \pi \models \text{post}$

* $c_r = \lceil \frac{c}{T_s} \rceil \cdot T_s$ and $d_r = \lceil \frac{d}{T_s} \rceil \cdot d$. The operators are defined for $c \geq T_s$ and $d \geq T_s$.

(b) Fixed Step Semantics.

$i, \pi \models \text{dur}(\text{le}) \geq c$	$\text{iff } \exists k \in \mathbb{N}^+. (k \leq i \wedge \iota_\tau(i) - \iota_\tau(k) \geq c \wedge$ $\quad \forall j \in \mathbb{N}^+. ((k \leq j \leq i) \Rightarrow j, \pi \models \text{le}))$
$i, \pi \models \text{pre}[d] \Rightarrow \text{post}$	$\text{iff } (\exists k \in \mathbb{N}^+. (k \leq i \wedge \iota_\tau(i) - \iota_\tau(k) \geq d \wedge$ $\quad \forall j \in \mathbb{N}^+. ((k \leq j \leq i) \Rightarrow j, \pi \models \text{pre})))$ $\Rightarrow i, \pi \models \text{post}$

(c) Variable Step Semantics.

Figure 5: Requirements Tables: Semantics.

369 The semantics of an expression of the form $t_1 \oplus t_2$
370 assesses whether the interpretation of the terms t_1 and t_2
371 at position i satisfy the relation specified by the relational
372 operator \oplus . For example, for Figure 2a, the expression
373 $\text{sp} < \text{th}$ does not hold at position 2 since the value 147
374 is not lower than 10. The semantics of an expression of the
375 form $\neg \text{le}$ specifies that the expression holds at position
376 i if le does not hold at that position. For example, for
377 Figure 2a, the expression $\neg(\text{sp} < \text{th})$ holds at position 2
378 since $\text{sp} < \text{th}$ does not hold. The semantics of the logical
379 expression $\text{le}_1 \odot \text{le}_2$ specifies that the expression holds at
380 position i if the evaluation of the logical expressions le_1
381 and le_2 at position i satisfy the relation specified by the
382 logical operator \odot . For example, for Figure 2a, the logical
383 expression $(\text{th} < 20) \wedge (\text{sp} < 150)$ holds at position 2
384 since both $\text{th} < 20$ and $\text{sp} < 150$ hold in that position.
385 The semantics of $\text{pre} \Rightarrow \text{post}$ (for the case in which the
386 duration column is not used) specifies that the expression
387 holds at position i if the postcondition post holds when
388 the precondition pre is satisfied. For example, for the first

requirement of Figure 1b and the position 5 of the trace the
formula is not satisfied since the precondition ($\text{th} > 0$) is
satisfied, but the postcondition ($\text{sp} = \text{prev}(\text{sp}) + \text{th}$) is
not. The semantics of req, rt specifies that the expression
holds at position i if the requirement req holds and the
remaining requirements in the requirement table are also
satisfied.

Two semantics for the operator $\text{dur}(\text{le}) \geq c$ and re-
quirements expressed using a duration column are defined.

- *Fixed step semantics.* For the operator $\text{dur}(\text{le}) \geq c$, we
assume that every constant c used within the duration
operator is greater than the sample time T_s ($c \geq T_s$).
Additionally, the value $c_r = \lceil \frac{c}{T_s} \rceil \cdot T_s$ is used to define
the operator's semantics, where $\lceil \cdot \rceil$ is the ceil function.
This value accounts for the value c not being a multiplier
of the fixed sampling time T_s . For example, consider the
expression operator $\text{dur}(\text{th} < 20) \geq 0.3$, the trace from
Figure 2a, and the position $i = 3$ with timestamp 0.6. For
 $\text{dur}(\text{th} < 20) \geq 0.3$ to hold in position 3, the expression
 $\text{th} < 20$ should hold also on the record at positions 1,
2, and 3 with timestamps 0.2, 0.4, and 0.6 since this will
imply that $\text{th} < 20$ holds for at least 0.4s (0.6-0.4s) which
is more than 0.3s. For this case, $c_r = 0.4$ i.e., $c_r = \lceil \frac{0.3}{T_s} \rceil \cdot T_s =$
 $\lceil \frac{0.3}{0.2} \rceil \cdot 0.2 = 2 \cdot 0.2 = 0.4$. For $c_r = 0.4$, the records with
timestamps included within the interval $[0.2, 0.6]$, i.e., 0.2,
0.4, and 0.6, are considered to assess the evaluation of the
expression $\text{dur}(\text{th} < 20) \geq 0.3$ at position 3. Formally, the
fixed step semantics (Figure 5b) specifies that the current
timestamp $\iota_\tau(i)$ is greater than the value c (that is, at least
 c elapsed since the system execution started), and for all
the records in a position k such that the difference between
the timestamps $\iota_\tau(i)$ and $\iota_\tau(i-k)$ is lower than c , the
expression le is satisfied (i.e., $k, \pi \models \text{le}$). For example,
for Figure 5b, the expression $\text{dur}(\text{th} > 3) \geq 0.4$ holds
at position 3 since for the positions 1, 2 and 3 (i.e., the
positions for which $\iota_\tau(i) - \iota_\tau(i-k) \leq 0.4$ hold) the
expression $\text{th} > 3$ holds. The fixed step semantic of a
requirement expressed using a duration column uses the
value $d_r = \lceil \frac{d}{T_s} \rceil \cdot d$ and is defined if the duration d is
greater than the sample time T_s ($d \geq T_s$). The fixed step
semantic of a requirement expressed using a duration col-
umn requires the postcondition (post) to hold in position
 i if (a) the current timestamp $\iota_\tau(i)$ is greater than the
value c (that is, at least c elapsed since the system execution
started), and for all the records in a position k such that the
difference between the timestamps $\iota_\tau(i)$ and $\iota_\tau(i-k)$ is
lower than c , the precondition is satisfied (i.e., $k, \pi \models \text{pre}$).
- *Variable step semantics.* The variable step semantics (Fig-
ure 5c) specifies that the expression holds at position i
if there exists a position k lower than i such that the
difference $\iota_\tau(i) - \iota_\tau(k)$ of the timestamps associated with
positions i and k is greater than or equal to the value c and
that the logical expression le holds from k to i . For exam-

ple, for Figure 2b, the expression $\text{dur}(\text{th} > 3) \geq 1$ holds at position 3 since at positions 1, 2 and 3 the expression $\text{th} > 3$ holds and the difference between the timestamp 1.8 at position 3 and the timestamp 0.2 at position 1 is greater than or equal to 1.

The variable step semantic of a requirement expressed using a duration column requires the postcondition (`post`) to hold in position i if there exists a position k lower than i such that the difference $\iota_\tau(i) - \iota_\tau(k)$ of the timestamps associated with positions i and k is greater than or equal to the value c and that the precondition (`pre`) holds from k to i .

Definition 4 (Semantics). *A trace π satisfies a Requirements Table rt , if $i, \pi \models rt$ holds for every position i . We use the notation $\pi \models rt$ to indicate that the trace π satisfies a Requirements Table rt .*

For example, the Requirements Table from Figure 4a is violated by the fixed step trace from Figure 2, since the precondition ($\text{dur}(\text{th} > 3) \geq 0.4$) of the first requirement of the Requirements Table holds in position 3 of the trace, but its postcondition ($\text{sp} = 1$) does not hold.

4 COMPLETENESS AND CONSISTENCY

Completeness and consistency are two properties of the Requirements Table. A Requirements Table is complete if it specifies the system behavior for every input. A Requirements Table is consistent if there are no contradictions in the requirements that prevent engineers from developing a system that satisfies them. In the following, we formalize the notions of completeness and consistency.

Definition 5 (Completeness). *A Requirements Table rt is complete if for every output variable $y \in \mathcal{Y}$, trace $\pi \in \Pi$, and position i , there exist a precondition $\text{pre} \in \text{Pre}(rt, y)$ satisfied by the trace π at position i , i.e., $i, \pi \models \text{pre}$.* \square

Intuitively, a Requirements Table is complete if at least one precondition is satisfied in every position to force the system to behave according to the corresponding postcondition. For example, the Requirements Table from Figure 4a is complete since, for every trace, in every position either $\text{dur}(\text{th} > 3) \geq 0.4$ or $\text{dur}(\text{th} > 3) < 0.4$ holds. The Requirements Table from Figure 4b is not complete since when $\text{dur}(u > 3) = 0.4$ none of the preconditions are satisfied.

Theorem 1. *Checking the completeness of Requirements Tables is an undecidable problem.*

Proof Sketch. We consider the tenth Hilbert problem [29], i.e., the problem of finding the values of a set of values for a set of integer variables of a polynomial equation with integer coefficients (a.k.a. Diophantine equation [30]) that solve the equation, which is undecidable. We reduce the tenth Hilbert problem to the problem of checking the completeness of a Requirements Table. Specifically, for each Diophantine equation of the form $A = 0$, we can create a table $A \neq 0 \Rightarrow y = 0$. The table is incomplete if and only if equation $A = 0$ has a solution. Therefore, being able to check the completeness of this Requirements Table would mean being able to solve the Diophantine equation

(which is an undecidable problem). This reduction proves that checking the completeness of Requirements Tables is undecidable. \square

Definition 6 (Consistency). *A Requirements Table rt is consistent if for every input interpretation $\iota_{\mathcal{U}}$ there exists an output $\iota_{\mathcal{Y}}$ and a timestamp ι_τ interpretation, such that the corresponding trace $\pi = \langle \iota_{\mathcal{U}}, \iota_{\mathcal{Y}}, \iota_\tau \rangle$ satisfies the Requirements Table, i.e., $\pi \models rt$.* \square

For example, the Requirements Table from Figure 4c is inconsistent since for every input interpretation that satisfies the preconditions $\text{dur}(\text{th} > 3) \leq 0.5$ and $\text{dur}(\text{th} > 3) \geq 0.4$, it does not exist an output interpretation that satisfies both the postcondition $\text{sp} = 1$ and the postcondition $\text{sp} = 2$. The Requirements Table from Figure 4d is consistent, for every input interpretation that satisfies the preconditions $\text{dur}(\text{th} > 3) > 0.4$ and $\text{dur}(\text{th} > 4) \leq 0.5$ it exists an output interpretation (i.e., $\text{sp} = 1$) that satisfies both the postcondition $\text{sp} = 1$ and the postcondition $\text{sp} \geq 1$.

Theorem 2. *Checking the consistency of Requirements Tables is an undecidable problem.*

Proof Sketch. We reduce the tenth Hilbert problem to the problem of checking the consistency of a Requirements Table. Specifically, for each Diophantine equation of the form $A = 0$, we can create a table $\text{true} \Rightarrow A = 0$. The table is consistent if and only if equation $A = 0$ has a solution. This reduction proves that checking the consistency of Requirements Tables is undecidable. \square

We do not provide separate bounded versions of our definitions for the bounded verification. Instead, we will discuss (Section 5 and Section 8) how the value of the bound influences the correctness of the result obtained with our bounded encodings.

Our discussion section (Section 8) will reflect on constraints that can be added to the Requirements Tables to make the completeness and consistency checking problems decidable. Note that the consistency and completeness procedures implemented by the Simulink Requirements Toolbox currently prompt the message “*incompatible for requirements table analysis with Simulink Design Verifier*” when the solver (i.e., the Simulink Design Verifier) can not process the current instance of the Requirements Table.

5 CHECKING COMPLETENESS AND CONSISTENCY

Our solution takes a Requirements Table as input and checks its completeness and consistency following the approach from Figure 6. The RT2LOGIC component (1) converts the Requirements Table into a logic formula. Different logic formulae are constructed depending on whether the completeness or consistency of the Requirements Tables need to be verified and whether variable or fixed step traces need to be considered. The SATCHECK component (2) verifies the satisfiability of the logic formula: the result of the satisfiability check specifies whether the Requirements Table is complete or incomplete and consistent or inconsistent.

We present two approaches to converting Requirements Tables into logic formulae. Both approaches follow the

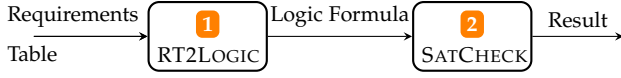


Figure 6: THEANO: Approach.

intuition described in Figure 6. The first approach (Section 5.2) is based on an unbounded encoding of Requirements Tables. The second approach (Section 5.3) relies on a bounded encoding. Our evaluation (Section 7) will compare these two approaches empirically. Before presenting how to convert Requirements Tables into logic formulae, we first (Section 5.1) describe how traces can be encoded into logic formulae, since the conversions will be based on these encodings. Finally, Section 5.4 summarizes the different encodings.

5.1 Encoding of the Traces

We present two encodings for traces that rely on uninterpreted functions or arrays for trace representation.

Encoding Based on Arrays (Ar). Our encoding creates one array \bar{u} and \bar{y} for each input $u \in \mathcal{U}$ and output $y \in \mathcal{Y}$ variable. We use the symbol \bar{v} to indicate the array associated with a variable v when it is not relevant if v is an input or an output variable. We use $\bar{\mathcal{U}}$ and $\bar{\mathcal{Y}}$ to indicate the sets of the input and output arrays generated for the input and output variables in \mathcal{U} and \mathcal{Y} . The encoding also creates the array $\bar{\tau}$ that contains the timestamps corresponding to each trace position. The trace array $\bar{\pi}$ is the tuple $\langle \bar{\mathcal{U}}, \bar{\mathcal{Y}}, \bar{\tau} \rangle$ containing the input, outputs and timestamp arrays. We use **Ar** to refer to an encoding based on arrays.

Encoding Based on Uninterpreted Functions (Uf). Unlike the encoding based on arrays, this encoding relies on uninterpreted functions to represent the timestamps. An uninterpreted function [31] is a function that has a name and signature (domain and co-domain) but no definition. With a slight abuse of notation, we rely on the same symbols used for the encoding based on arrays to denote the uninterpreted functions used to encode the traces. The encoding creates the uninterpreted function⁴ $\bar{\tau} : \mathbb{N}^0 \rightarrow \mathbb{R}$ that contains the timestamps corresponding to each trace position. The trace function $\bar{\pi}$ is the tuple $\langle \bar{\mathcal{U}}, \bar{\mathcal{Y}}, \bar{\tau} \rangle$ containing the input, outputs arrays and the timestamp function. We use **Uf** to refer to an encoding based on uninterpreted functions.

In the following, we will avoid considering the structures used for representing traces for the definition of the unbounded and the bounded encodings of Requirements Tables: We will use the symbols $\bar{v}[i]$ and $\bar{\tau}[i]$ to seamlessly indicate the values associated by the uninterpreted functions \bar{v} and $\bar{\tau}$ to the value $i \in \mathbb{N}^0$, or the value of the arrays \bar{v} and $\bar{\tau}$ in position $i \in \mathbb{N}^0$. Our evaluation (Section 7) will assess how the two structures for representing traces influence the effectiveness and efficiency of THEANO.

For our trace encoding to be correct, we also need to constraint the timestamp evolution. Table 1 presents the trace encodings based on uninterpreted functions and arrays for the traces, for the unbounded and the bounded encodings, depending on whether the traces have a variable or fixed

Table 1: Encodings τ_m of the traces.

Encoding	Sample Time	Formula
Unbounded	Variable Fixed	$\forall i \geq 0, \bar{\tau}[i] < \bar{\tau}[i+1]$ $\forall i \geq 0, \bar{\tau}[i] = i \cdot T_s$
Bounded	Variable Fixed	$\bigwedge_{i \in [0, b)} \bar{\tau}[i] < \bar{\tau}[i+1]$ $\bigwedge_{i \in [0, b)} \bar{\tau}[i] = i \cdot T_s$

* $T_s > 0$

$\mathbf{h}_{i,\bar{\pi}}(c)$	$\equiv c$
$\mathbf{h}_{i,\bar{\pi}}(v)$	$\equiv \bar{v}[i]$
$\mathbf{h}_{i,\bar{\pi}}(\text{prev}(u))$	$\equiv ((i > 0) \Rightarrow (u[i-1])) \wedge ((i = 0) \Rightarrow u[i] = 0)$
$\mathbf{h}_{i,\bar{\pi}}(t_1 \odot t_2)$	$\equiv \mathbf{h}_{i,\bar{\pi}}(t_1) \odot \mathbf{h}_{i,\bar{\pi}}(t_2)$
$\mathbf{h}_{i,\bar{\pi}}(t_1 \oplus t_2)$	$\equiv \mathbf{h}_{i,\bar{\pi}}(t_1) \oplus \mathbf{h}_{i,\bar{\pi}}(t_2)$
$\mathbf{h}_{i,\bar{\pi}}(\neg \text{le})$	$\equiv \neg \mathbf{h}_{i,\bar{\pi}}(\text{le})$
$\mathbf{h}_{i,\bar{\pi}}(\text{le}_1 \odot \text{le}_2)$	$\equiv \mathbf{h}_{i,\bar{\pi}}(\text{le}_1) \odot \mathbf{h}_{i,\bar{\pi}}(\text{le}_2)$
$\mathbf{h}_{i,\bar{\pi}}(\text{req})$	$\equiv \mathbf{h}_{i,\bar{\pi}}(\text{pre}) \Rightarrow \mathbf{h}_{i,\bar{\pi}}(\text{post})$
$\mathbf{h}_{i,\bar{\pi}}(\text{rt}, \text{req})$	$\equiv \mathbf{h}_{i,\bar{\pi}}(\text{rt}) \wedge \mathbf{h}_{i,\bar{\pi}}(\text{req})$

* $\bar{\pi} = \langle \bar{\tau}, \bar{\mathcal{U}}, \bar{\mathcal{Y}} \rangle$

(a) Translation for the operators not based on timestamp values.

$\mathbf{h}_{i,\bar{\pi}}(\text{dur}(\text{le}) \geq c)$	$\equiv \bar{\tau}[i] \geq c \wedge c \geq T_s$ $\forall k, (i - \lceil \frac{c}{T_s} \rceil \leq k \leq i) \Rightarrow \mathbf{h}_{k,\bar{\pi}}(\text{le})$
$\mathbf{h}_{i,\bar{\pi}}(\text{pre}[d] \Rightarrow \text{post})$	$\equiv (\bar{\tau}[i] \geq d \wedge d \geq T_s)$ $\forall k, (i - \lceil \frac{d}{T_s} \rceil \leq k \leq i) \Rightarrow \mathbf{h}_{k,\bar{\pi}}(\text{pre})$ $\Rightarrow \mathbf{h}_{i,\bar{\pi}}(\text{post})$

(b) Fixed Step Semantics (Fs).

$\mathbf{h}_{i,\bar{\pi}}(\text{dur}(\text{le}) \geq c)$	$\equiv \exists j, (j \leq i \wedge \bar{\tau}[j] - \bar{\tau}[j] \geq c \wedge$ $\forall k, (j \leq k \leq i) \Rightarrow \mathbf{h}_{k,\bar{\pi}}(\text{le})))$
$\mathbf{h}_{i,\bar{\pi}}(\text{pre}[d] \Rightarrow \text{post})$	$\equiv (\exists j, (j \leq i \wedge \bar{\tau}[j] - \bar{\tau}[j] \geq d \wedge$ $\forall k, (j \leq k \leq i) \Rightarrow \mathbf{h}_{k,\bar{\pi}}(\text{pre})))$ $\Rightarrow \mathbf{h}_{i,\bar{\pi}}(\text{post})$

(c) Variable Step Semantics (Vs).

Figure 7: Unbounded Encoding of Requirements Tables.

sample step. Intuitively, for the variable step encoding, the constraint forces the timestamp to monotonically increase. For the fixed-step encoding, the constraint forces the timestamps to be multiples of the sample time T_s . We will use τ_m to refer to the timestamp constraint.

5.2 The Unbounded Encoding of Requirements Tables

The unbounded encoding of Requirements Tables is defined by the function **h** from Figure 7 and behaves as follows. We use **Ue** to refer to the unbounded encoding.

Figure 7a present the translation of the operators that are not based on timestamp values. For a constant c , the translation returns the corresponding value. For a variable v it returns the value $\bar{v}[i]$. For **prev**(v) the translation returns $u[i-1]$ if i is greater than 0, $u[i]$ otherwise; for $t_1 \odot t_2$ it returns $\mathbf{h}_i(t_1) \odot \mathbf{h}_i(t_2)$; for $t_1 \oplus t_2$ it returns $\mathbf{h}_i(t_1) \oplus \mathbf{h}_i(t_2)$; for $\neg \text{le}$ it returns $\neg \mathbf{h}_i(\text{le})$; for $\text{le}_1 \odot \text{le}_2$ it returns $\mathbf{h}_i(\text{le}_1) \odot \mathbf{h}_i(\text{le}_2)$; for **req** it returns $\mathbf{h}_i(\text{pre}) \Rightarrow \mathbf{h}_i(\text{post})$, and for $\mathbf{h}_i(\text{rt}, \text{req})$ it returns $\mathbf{h}_i(\text{rt}) \wedge \mathbf{h}_i(\text{req})$.

The encoding of the operator **dur**(le) $\geq c$ and requirements expressed using a duration column depends on whether the fixed or variable step semantics are considered (see Section 3.2).

- Fixed step semantics.* For the fixed step semantics (Figure 7b) and the operator **dur**(le) $\geq c$, it returns a formula requiring (a) the timestamp $\bar{\tau}[i]$ at position i to be greater than c ,

4. <https://microsoft.github.io/z3guide/docs/logic/Uninterpreted-functions-and-constants/>.


```

1  $\exists \bar{\pi}, \exists i, (\forall j \geq 0, \bar{\tau}[j] < \bar{\tau}[j+1] \wedge$ 
2  $\neg(\bar{\tau}[i] \geq 0.4 \wedge 0.4 \geq 0.2 \wedge \forall k, ((i-2 \leq k \leq i) \Rightarrow \text{th}[i] > 3)) \wedge$ 
3  $\neg(\bar{\tau}[i] \geq 0.4 \wedge 0.4 \geq 0.2 \wedge \exists k, ((i-2 < k \leq i) \wedge \neg \text{th}[i] > 3)))$ 

```

(a) Encoding for checking the completeness of the Requirements Table from Figure 4b.

```

1  $\exists \text{th}, \exists \bar{\tau}, (\forall j \geq 0, \bar{\tau}[j] < \bar{\tau}[j+1] \wedge$ 
2  $\forall \text{sp}, (\exists i, \neg($ 
3  $\quad (\bar{\tau}[i] \geq 0.4 \wedge 0.4 \geq 0.2 \wedge$ 
4  $\quad \quad \forall k, (((i-2 \leq k \leq i) \Rightarrow \text{th}[i] > 3) \Rightarrow \text{sp}[i] = 1)) \vee$ 
5  $\quad \quad (\bar{\tau}[i] \geq 0.5 \wedge 0.5 \geq 0.2 \wedge$ 
6  $\quad \quad \forall k, (\neg(((\bar{\tau}[i] - \bar{\tau}[i-k] \leq 0.5) \Rightarrow \text{th}[i] > 3) \Rightarrow \text{sp}[i] = 2))))))$ 

```

(b) Encoding for checking the consistency of the Requirements Table from Figure 4c.

Figure 8: Unbounded Encoding for checking the completeness and consistency of the Requirements Table from Figure 4.

(b) the value of the c to be greater than the sample time T_s ,
(c) the formula $\mathbf{h}_k(\text{le})$ to hold for every position k within $[i - \lceil \frac{c}{T_s} \rceil, i]$. Note that the definition of the translation for the operator $\mathbf{h}_k(\text{le})$ is consistent with the fixed step semantics of Requirements Tables from Section 3.2. The formula returned for the requirements expressed using a duration column is similar and follows the fixed step semantics of Requirements Tables from Section 3.2.

- *Variable step semantics.* For the variable step semantics (Figure 7c) and the operator $\mathbf{dur}(\text{le}) \geq c$, it returns a formula requiring the existence of an index j lower than i such that the difference $\bar{\tau}[i] - \bar{\tau}[j]$ between the timestamp in position i and the timestamp in position j is greater than or equal to c and for every position k , such that $j \leq k \leq i$, the formula $\mathbf{h}_k(\text{le})$ holds. The formula returned for the requirements expressed using a duration column is similar and follows the fixed step semantics of Requirements Tables from Section 3.2.

We will use Fs and Vs to refer to an encoding based on the fixed or variable step semantics.

Completeness. Let rt be a Requirements Table, the logic formula $\mathbf{h}_{cp}(\text{rt})$ to verify the completeness of the Requirements Table rt is

$$\exists \bar{\pi}, \exists i, \left(\tau_m \wedge \left(\bigvee_{y \in \mathcal{Y}} \left(\bigwedge_{\text{pre} \in \text{Pre}(\text{rt}, y)} \neg \mathbf{h}_{i, \bar{\pi}}(\text{pre}) \right) \right) \right)$$

The formula specifies the existence of a trace ($\bar{\pi}$) and a position (i) where for at least one output variable (y), all the conditions (pre) of the corresponding preconditions ($\text{Pre}(\text{rt}, y)$) do not hold ($\neg \mathbf{h}_{i, \bar{\pi}}(\text{pre})$). For example, Figure 8a presents the encoding for Requirements Table from Figure 4b with sample step 0.2. Line 2 contains the encoding for the requirement 1 from Figure 4b. Line 3 contains the encoding for the requirement 2 from Figure 4b.

Theorem 3. Let rt be a Requirements Table, the formula $\mathbf{h}_{cp}(\text{rt})$ is satisfiable if and only if the Requirements Table rt is incomplete.

Proof Sketch. The encoding from Figure 7 is obtained by applying a one-to-one mapping of the semantics of the constructs of the Requirements Tables from Figure 5. There-

fore, checking for the presence of an array (or uninterpreted function) and an integer that satisfies formula $\mathbf{h}_{cp}(\text{rt})$ corresponds to verifying the presence of an output variable $y \in \mathcal{Y}$, a trace $\pi \in \Pi$, and a position i that violates the conditions of Definition 5. \square

For example, Figure 8a presents the encoding for Requirements Table from Figure 4b with sample step 0.2. For example, the formula from Figure 8a is satisfiable: consider the case in which $\forall k, ((i-2 < k \leq i) \Rightarrow \text{th}[i] > 3)$, but $i-2 = k \wedge \neg \text{th}[i] > 3$ the subformulae reported in Lines 2 and 3 are true since both the formulae within the parentheses do not hold, for the one reported in Line 2 $k = i-2$ and $\text{th}[i] > 3$ make the implication false since the left side of the implication is true and the right side of the implication is true, for the one reported in Line 3 $k = i-2$ and $\neg \text{th}[i] > 3$ make the implication false since the left side of the conjunction is false.

Consistency. Let rt be a Requirements Table, the logic formula $\mathbf{h}_{cs}(\text{rt})$ to verify the consistency of the Requirements Table rt is

$$\exists \bar{\mathcal{U}}, \exists \bar{\tau}, (\tau_{mon} \wedge \forall \bar{\mathcal{Y}}, (\exists i, (\neg \mathbf{h}_{i, \bar{\pi}}(\text{rt}))))$$

with $\bar{\pi} = \langle \bar{\mathcal{U}}, \bar{\mathcal{Y}}, \bar{\tau} \rangle$. The formula specifies the existence of a set uninterpreted functions (or arrays) for the input variables and for the timestamp variables, such that for every set of uninterpreted functions (or arrays) for the output variables the requirements of the Requirements Table do not hold ($\exists i, (\neg \mathbf{h}_{i, \bar{\pi}}(\text{rt}))$).

For example, Figure 8b presents the encoding for Requirements Table from Figure 4c with sample step 0.2. Lines 2, 3 contain the encoding for the requirement 1 from Figure 4c. Lines 3, 4 contain the encoding for the requirement 2 from Figure 4c.

Theorem 4. Let rt be a Requirements Table, the formula $\mathbf{h}_{cs}(\text{rt})$ is satisfiable if and only if the Requirements Table rt is inconsistent.

Proof Sketch. The encoding from Figure 7 is obtained by applying a one-to-one mapping of the semantics of the constructs of the Requirements Tables from Figure 5. Therefore, checking for the presence of a set uninterpreted functions (or arrays) for the input variables and for the timestamp variables that satisfies formula $\mathbf{h}_{cs}(\text{rt})$ corresponds to verifying the presence of an input interpretation $\iota_{\mathcal{U}}$ that satisfies the conditions of Definition 6. \square

For example, the formula from Figure 8b is satisfiable since the interpretation for the input (th) and the timestamp ($\bar{\tau}$) variables ensuring the following conditions $\bar{\tau}[10] \geq 0.5$, and $0.4 < \bar{\tau}[10] - \bar{\tau}[9] < 0.5$, and $\text{th}[10] > 3$ is such that for every interpretation of the output variable (sp) there exists an i such as either $\text{sp}[i] = 1$, which makes the first condition of the disjunction satisfied, or $\text{sp}[i] = 2$ which makes the second condition of the disjunction satisfied.

Our encoding generates formulae within the AUFNIRA logic, i.e., closed formulas with free function and predicate symbols over a theory of arrays of integer index and real value [32]. The problem of deciding if an AUFNIRA formula is satisfiable is undecidable, i.e., there is no algorithm that always answers if the formula is satisfiable.

The unbounded approach relies on infinite traces (i.e., infinite sequences of inputs, outputs, and timestamps) represented as uninterpreted functions or unbounded arrays. The following section presents a bounded approach. Unlike the unbounded approach, the bounded approach demonstrates incompleteness and inconsistency by relying on a finite (bounded) trace prefix. Our evaluation (Section 7) will compare the two approaches empirically.

5.3 The Bounded Encoding of Requirements Tables

The bounded encoding checks if a Requirements Table is incomplete and inconsistent by considering a bound (b) on the maximum number of trace records (positions) to be considered in the analysis. For example, if the bound b is five, the algorithm checks for inconsistency and incompleteness by considering traces that have at most five records. We use the identifier Be to refer to the bounded encoding.

Figure 9 presents the bounded encoding for the operator $\text{dur}(1e) \geq c$ and requirements expressed using a duration column of the Requirements Tables; For the remaining operators, the bounded encoding corresponds to the unbounded encoding from Figure 7. The encoding for the operator $\text{dur}(1e) \geq c$ and requirements expressed using a duration column replaces the quantifiers \forall and \exists by unrolling the existential and universal quantifiers with the conjunction (\wedge) and disjunction (\vee) operators.

- *Fixed step semantics.* For the operator $\text{dur}(1e) \geq c$, the translation from Figure 9a forces the formula $1e$ to hold for every position n within $[i - \lceil \frac{c}{T_s} \rceil, i]$ by creating a conjunction of formulae $\text{h}_{n \leftarrow k, \pi}(1e)$ each representing the value of $1e$ in a position n . We use the notation $n \leftarrow k$ to indicate that the variable k is replaced by the value n . The conjunction is true only if all the conjunct formulae are true. For the requirements expressed using a duration column, the formula returned for the requirements expressed using a duration column follows the fixed step semantics of Requirements Tables from Section 3.2.
- *Variable step semantics.* For the operator $\text{dur}(1e) \geq c$, the disjunction operator from Figure 9b forces the existence of a position j with a timestamp $\bar{\tau}[j]$ ensuring that $\bar{\tau}[i] - \bar{\tau}[j] \geq c$, while the conjunction operator forces the formula $1e$ to hold in every position n in $[j, i]$ by creating a conjunction of formulae $\text{h}_{n \leftarrow k, \pi}(1e)$ each representing the value of $1e$ in a position n . For the requirements expressed using a duration column, the formula returned for the requirements expressed using a duration column follows the variable step semantics of Requirements Tables from Section 3.2.

The logic formula $\text{h}_{bcp}(\text{rt})$ to verify the incompleteness of a Requirements Table rt is

$$\exists \bar{\pi}, \left(\tau_{\text{mon}} \bigvee_{i \in [0, b], y \in \mathcal{Y}} \left(\bigwedge_{\text{pre} \in \text{Pre}(\text{rt}, y)} \neg \text{h}_{i, \bar{\pi}}(\text{pre}) \right) \right)$$

For example, Figure 10a reports the encoding for the Requirements Table from Figure 4b for the fixed step semantics and bound $b = 2$. To simplify our presentation, we assumed that $T_s = 0.4$ and omitted portions of the formulae that are not useful to our discussion (e.g., $0.4 \geq 0.4$).

$$\begin{aligned} \text{h}_{i, \bar{\pi}}(\text{dur}(1e) \geq c) &\equiv \bar{\tau}[i] \geq c \wedge c \geq T_s \wedge \bigwedge_{k \in [i - \lceil \frac{c}{T_s} \rceil, i]} \text{h}_{n \leftarrow k, \bar{\pi}}(1e) \\ \text{h}_{i, \bar{\pi}}(\text{pre}[d] \Rightarrow \text{post}) &\equiv \bar{\tau}[i] \geq d \wedge d \geq T_s \wedge \\ &\quad ((\bigwedge_{k \in [i - \lceil \frac{d}{T_s} \rceil, i]} \text{h}_{n \leftarrow k, \bar{\pi}}(\text{pre})) \Rightarrow \text{h}_{n \leftarrow i, \bar{\pi}}(\text{post})) \end{aligned}$$

(a) Fixed Step Semantics.

$$\begin{aligned} \text{h}_{i, \bar{\pi}}(\text{dur}(1e) \geq c) &\equiv \bigvee_{j \in [0, i]} (\bar{\tau}[i] - \bar{\tau}[j] \geq c \wedge \bigwedge_{k \in [j, i]} \text{h}_{n \leftarrow k, \bar{\pi}}(1e)) \\ \text{h}_{i, \bar{\pi}}(\text{pre}[d] \Rightarrow \text{post}) &\equiv \bigvee_{j \in [0, i]} (\bar{\tau}[i] - \bar{\tau}[j] \geq d \wedge \\ &\quad (\bigwedge_{k \in [j, i]} \text{h}_{n \leftarrow k, \bar{\pi}}(\text{pre})) \Rightarrow \text{h}_{n \leftarrow i, \bar{\pi}}(\text{post})) \end{aligned}$$

(b) Variable Step Semantics.

Figure 9: Bounded Encoding.

$$\begin{aligned} 1 & \exists \bar{\pi}, (\bar{\tau}[0] = 0 \wedge \bar{\tau}[1] = 0.4 \wedge \bar{\tau}[2] = 0.8 \wedge \\ 2 & (\neg(\bar{\tau}[0] \geq 0.4 \wedge \text{th}[0] > 3 \wedge \text{th}[1] > 3) \wedge \\ 3 & \neg(\bar{\tau}[0] \geq 0.4 \wedge (\neg \text{th}[0] > 3))) \\ 4 & \vee \\ 5 & (\neg(\bar{\tau}[1] \geq 0.4 \wedge \text{th}[0] > 3 \wedge \text{th}[1] > 3 \wedge \text{th}[2] > 3) \wedge \\ 6 & \neg(\bar{\tau}[1] \geq 0.4 \wedge (\neg \text{th}[0] > 3 \vee \neg \text{th}[1] > 3))) \end{aligned}$$

(a) Bounded encoding for checking the incompleteness of the Requirements Table from Figure 4b for bound $b = 2$.

$$\begin{aligned} 1 & \exists \bar{\pi}, (\bar{\tau}[0] = 0 \wedge \bar{\tau}[1] = 0.4 \wedge \bar{\tau}[2] = 0.8 \wedge \\ 2 & (\neg(\bar{\tau}[0] \geq 0.4 \wedge \text{th}[0] > 3) \Rightarrow \text{sp}[0] = 1) \vee \\ 3 & \neg(\bar{\tau}[0] \geq 0.5 \wedge \text{th}[0] > 3) \Rightarrow \text{sp}[0] = 2)) \\ 4 & \vee \\ 5 & ((\neg(\bar{\tau}[1] \geq 0.4 \wedge \text{th}[0] > 3 \wedge \text{th}[1] > 3) \\ 6 & \Rightarrow (\text{sp}[0] = 1 \wedge \text{sp}[1] = 1))) \vee \\ 7 & (\neg(\bar{\tau}[1] \geq 0.5 \wedge \text{th}[0] > 3 \wedge \text{th}[1] > 3) \\ 8 & \Rightarrow (\text{sp}[0] = 2 \wedge \text{sp}[1] = 2))) \end{aligned}$$

(b) Bounded encoding for checking the inconsistency of the Requirements Table from Figure 4b for bound $b = 2$.

Figure 10: Bounded Encoding for checking the incompleteness and inconsistency of the Requirements Table from Figure 4.

Theorem 5. Let b be a bound and rt be a Requirements Table, if the formula $\text{h}_{bcp}(\text{rt})$ is satisfiable, then the Requirements Table rt is incomplete.

Proof Sketch. Since $\text{h}_{bcp}(\text{rt})$ is satisfiable for bound b , the formula $\text{h}_{cp}(\text{rt})$ is also satisfiable since the satisfiability of the formulae from Figure 9a and Figure 9b implies the satisfiability of the formulae from Figure 7b and Figure 7c. If the formula $\text{h}_{cp}(\text{rt})$ is satisfiable the Requirements Table rt is incomplete. Therefore, if the formula $\text{h}_{bcp}(\text{rt})$ is satisfiable, then the Requirements Table rt is incomplete. \square

The bounded encoding shows the incompleteness of the Requirements Table: by setting $\bar{\tau}[0] = 0$, $\bar{\tau}[1] = 0.4$, $\bar{\tau}[2] = 0.8$, $\text{th}[0] > 3$, $\text{th}[1] > 3$, and $\text{th}[2] \leq 3$, the formula from Figure 4b is satisfiable. Since the formula is satisfiable, the Requirements Table is incomplete.

The logic formula $\text{h}_{bcs}(\text{rt})$ to verify the inconsistency of a Requirements Table rt is

$$\exists \bar{\mathcal{U}}, \exists \bar{\tau}, \left(\tau_{\text{mon}} \wedge \forall \bar{\mathcal{Y}}, \left(\bigvee_{i \in [0, b]} \neg \text{h}_{i, \bar{\pi}}(\text{rt}) \right) \right)$$

For example, Figure 10b reports the encoding for the Requirements Table from Figure 4c for the fixed step semantics, $T_s = 0.4$ and bound $b = 2$.

Theorem 6. Let b be a bound and rt be a Requirements Table, if the formula $h_{bcs}(rt)$ is satisfiable, then the Requirements Table rt is inconsistent.

Proof Sketch. Since $h_{bcs}(rt)$ is satisfiable for bound b , the formula $h_{cs}(rt)$ is also satisfiable since the satisfiability of the formulae from Figure 9a and Figure 9b implies the satisfiability of the formulae from Figure 7b and Figure 7c. If the formula $h_{cs}(rt)$ is satisfiable the Requirements Table rt is inconsistent. Therefore, if the formula $h_{bcs}(rt)$ is satisfiable, then the Requirements Table rt is inconsistent. \square

For example, the formula from Figure 10b is satisfiable since the interpretation for the input (th) and the timestamp ($\bar{\tau}$) variables ensuring the following conditions $\bar{\tau}[1] \geq 0.5$, and $0 < \bar{\tau}[0] - \bar{\tau}[1] < 0.5$, and $th[1] > 3$ is such that for every interpretation of the output variable (sp) either $sp[0] \geq 1$ and $sp[1] = 1$, which makes the first condition of the disjunction satisfied, or $sp[0] = 2$ and $sp[1] = 2$ which makes the second condition of the disjunction satisfied. Since the formula is satisfiable, the Requirements Table is inconsistent.

Like the unbounded encoding (Section 5.2) the bounded encoding also generates formulae from the AUFNIRA logic.

Although our bounded encoding enables engineers to determine that Requirements Tables are incomplete and inconsistent, it does not enable them (in general) to determine that the Requirements Tables are complete or consistent: that is, we can not draw any conclusion about the completeness and consistency of the Requirements Tables if $h_{bcp}(rt)$ and $h_{bcs}(rt)$ are not satisfiable. However, in many practical cases, it may be possible also to determine that the Requirements Tables are complete or consistent by setting a sufficiently large bound. Since the goal of this work is to present and illustrate our encoding, the task of precisely identifying these cases is out of scope, and will be considered in future works.

5.4 Summary

Table 2 summarizes the set of encodings proposed for the Requirements Tables by the previous sections. The type of encoding depends on whether a bounded (Be) or unbounded (Ue) encoding is selected for the Requirements Tables, whether arrays (Ar) or uninterpreted functions (Uf) is used to represent traces, and on whether a fixed (Fs) or a variable (Vs) sample step semantics is selected. As shown in Section 4 checking the completeness and consistency of Requirements Tables containing flexible requirements with state is undecidable. Therefore, although the unbounded encoding (Ue) provides a more precise encoding than the bounded encoding (Be), checking whether the produced AUFNIRA formula is satisfiable is undecidable. Therefore, the tool produces an answer only when it can deduce that the AUFNIRA formula is satisfiable or unsatisfiable. Our evaluation (Section 7) will empirically assess how frequent this situation is in practical cases. On the other hand, we conjecture that the bounded encoding (Be) might help deduce whether a Requirement Table is incomplete (Theorem 5) or inconsistent (Theorem 6). Our evaluation (Section 7) will assess whether this conjecture is supported by empirical evidence. Arrays have already been used to

Table 2: Encodings proposed for the Requirements Tables.

Encoding	Bound	Trace	Semantics
UeUfFs	Unbounded	Function	Fixed Step
UeUfVs	Unbounded	Function	Variable Step
UeArFs	Unbounded	Array	Fixed Step
UeArVs	Unbounded	Array	Variable Step
BeUfFs	Bounded	Function	Fixed Step
BeUfVs	Bounded	Function	Variable Step
BeArFs	Bounded	Array	Fixed Step
BeArVs	Bounded	Array	Variable Step



Figure 11: Implementation of THEANO.

model traces in the research literature (e.g., [33]). In this work, we also propose using uninterpreted functions as an alternative. Our evaluation will compare the two solutions. Finally, we proposed both a fixed (Fs) or a variable (Vs) sample step semantics. Users need to select the appropriate semantics depending on whether a fixed or variable sample step solver is used in their simulations.

6 IMPLEMENTATION

Figure 11 presents a high-level description of the implementation of THEANO. The implementation relies on three components: RT2TEXT (1), TEXT2Z3 (2), and SATCHECK (3).

The RT2TEXT component (1) is a MATLAB (v.R2023a) script. It takes a Simulink model as input, extracts the Requirements Tables, and generates a text (.rt) file for each Requirements Table. The encoding of each Requirements Table in the text file follows the syntax from Figure 3. Additionally, it extracts from the inputs of the Requirements Table the type of each input, which is added to the text file to be used later in the encoding. Figure 12 presents an example of the intermediate (.rt) file produced by the RT2TEXT for the Requirement Table from Figure 1b. The tokens *vardef* and *endvardef* delimit the variables declaration. Each line contains the declaration of a variable: the name of the variable is followed by its type and an identifier that indicates if the variable is an input or an output variable. The tokens *reqdef* and *endreqdef* delimit the requirements declaration. Each line contains the declaration of a requirement specified by a precondition followed by a postcondition separated by a comma character.

The TEXT2Z3 component (2) is developed using antlr [24], a parser generator able to read grammars and generate a recognizer for the language. We selected antlr since it is a widely used tool [34] and a grammar for parsing MATLAB expressions was publicly available [35]. TEXT2Z3 implements the translation from Section 5 and generates a Python file that encodes the formulae in Z3 [26]. Z3 was chosen since it is an industry-strength awarded tool to check the satisfiability of logic-based formulae [36], [37]. Specifically, we used Z3Py [25], a Z3 API in Python, since it is a tool that is easy to use and install, that we used in previous research (e.g., [33]). Figure 13 presents an example of the

```

1 vardef
2   th, Real, input;
3   sp, Real, output;
4 endvardef
5 reqdef
6   th>0, sp == prev(sp)+th;
7   prev(sp)==160, sp<160;
8   th<0, sp<prev(sp);
9 endreqdef

```

Figure 12: Intermediate (.rt) file produced by the RT2TEXT for the Requirement Table from Figure 1b.

```

1 from z3 import *;
2 s = Solver()
3
4 # Signal definition
5 th=Array('th',I,R)
6 sp=Array('sp',I,R)
7
8 # Timestamp definition
9 tau = Function('tau', IntSort(), RealSort())
10
11 # Timestamp monotonicity constraint
12 s.add(And(tau(0)<tau(1)))
13
14 # Requirements Table Encoding
15 s.add(
16   Or(
17     And(Not(th[0]>0),Not(th[0]<0),Not(sp[0]==160)),
18     And(Not(th[1]<0),Not(th[1]>0),Not(sp[0]==160)),
19     And(Not(th[2]>0),Not(th[2]<0),Not(sp[1]==160))
20   )
21 )
22
23 # Result Processing
24 res=s.check()
25 if (res.r == Z3_L_FALSE):
26   print('Requirements Table Complete (unsat)')
27   sys.exit(1)
28 else:
29   if (res.r == Z3_L_TRUE):
30     print('Requirements Table Incomplete (sat)')
31     sys.exit(-1)
32   else:
33     print('unknown')
34     sys.exit(0)

```

Figure 13: Python (.py) file produced by the TEXT2Z3 component for the file from Figure 12 for the BeUfVs encoding.

python (.py) file produced by the TEXT2Z3 component for the file from Figure 12 to check completeness for the BeUfVs encoding. The file consists of five parts: The signal definition part (Lines 4-6) defines the variables used to represent the signal values. The timestamp definition part (Lines 8-9) defines the constraint used to represent the timestamp values. The timestamp monotonicity constraint (Lines 11-12) defines the constraint τ_m on the timestamp evolution. The Requirements Table Encoding (Lines 14-21) encodes the Requirement Table for the completeness and consistency check as detailed in Section 5.2 and Section 5.3. Finally, the Result Processing (Lines 23-34) runs the satisfiability checker and returns the completeness and consistency result depending on the satisfiability of the Z3 formula.

The SATCHECK component (3) is implemented using Z3 [26]. The SATCHECK component executes the Python

files associated with each Requirements Table. Z3 returns whether the formula generated for the Requirements Table is satisfiable. Depending on this result, THEANO returns a value indicating whether the Requirements Table is complete or incomplete and consistent or inconsistent.

We decided to use several external tools that must be executed outside the MATLAB environment for the following reasons. The parser we found for MATLAB formulae is mparser [38] which is outdated, i.e., the last update of mparser was on 1 Sep 2011. In addition, this tool relies on libantlr3c [39], an ANTLRv3 parsing library for C, that needs to be installed separately, complicating the installation process.

7 EVALUATION

Our evaluation answers the following questions:

RQ1 *How effective is THEANO in checking the completeness and consistency of Requirements Tables? How do the encodings from Table 2 compare?* (Section 7.2)

Checking the completeness and consistency of a Requirements Table containing flexible requirements with states is undecidable (Section 4). This research question empirically assesses in how many cases THEANO can produce a definitive answer (completeness/incompleteness and consistent/inconsistent).

RQ2 *How efficient is THEANO in checking the completeness and consistency of Requirements Tables? How do the encodings from Table 2 compare?* (Section 7.3)

This research question assesses the time required to compute the results when THEANO could prove the completeness/incompleteness and consistency/inconsistency of the Requirements Table.

To answer these research questions, we rely on the following benchmark.

7.1 Benchmark

We could not find publicly available benchmark tables containing flexible requirements with states that we could use to answer RQ1 and RQ2, since these requirements are currently not supported by Simulink Requirements Tables [40]. Therefore, we artificially synthesized a benchmark of Requirements Tables containing these requirements. To synthesize our benchmark, we decided to consider a small set of representative baseline Requirements Tables that reflect the needs of practitioners. We generated the Requirements Tables of our benchmark from these tables.

Figure 14 presents our baseline Requirements Tables, where thr is a parameter representing a threshold value used to instantiate the Requirements Tables. The baseline Requirements Tables are described below.

Baseline Table 1 (Figure 14a). The requirement 1 of the Requirement Table specifies that the value 0 shall be initially assigned to the variable y . The requirement 2 specifies that, after the startup instant, the value of y shall equal its previous value plus one if the value assumed by the input u is greater than zero. The requirement 3 specifies that, after the startup instant, the value 23 shall be assigned to the variable y if the previous value of the variable y

ID	Pre.	Post.
1	isStartup	$y == 0;$
2	$\neg \text{isStartup} \wedge u > 0$	$y == \text{prev}(y) + 1;$
3	$\neg \text{isStartup} \wedge \text{prev}(y) == \text{thr}$	$y < \text{thr};$

(a) Baseline Table 1.

ID	Pre.	Post.
1	isStartup	$y == 0;$
2	$\neg \text{isStartup} \wedge u == \text{prev}(u) + 1$	$y == \text{prev}(y) + 1;$
3	$\neg \text{isStartup} \wedge u \neq \text{prev}(u) + 1 \wedge y < \text{thr}$	$y < \text{thr};$

(c) Baseline Table 3.

ID	Pre.	Post.
1	isStartup	$y == 0;$
2	$\neg \text{isStartup}$	$y == \text{prev}(y) + 1;$
3	$\neg \text{isStartup} \wedge \text{prev}(y) == \text{thr}$	$y < \text{thr};$

(b) Baseline Table 2.

ID	Pre.	Post.
1	isStartup	$y == 0;$
2	$\neg \text{isStartup} \wedge u == \text{prev}(u) + 1$	$y == \text{prev}(y) + 1;$
3	$\neg \text{isStartup} \wedge u \neq \text{prev}(u) + 1$	$y < \text{thr};$

(d) Baseline Table 4.

Figure 14: Baseline Requirements Tables used for the generation of our benchmark.

corresponds to the threshold value thr . Requirements Tables generated from this baseline are *incomplete and inconsistent*. The Requirements Tables are incomplete since none of the preconditions for the output variable y are triggered when for a trace π and position $i > 0$, the value u is lower than or equal to zero, and the previous value of y is not equal to the threshold value thr . The Requirements Tables are inconsistent since it does not exist any output interpretation that satisfies the requirements of the Requirements Table for an input u that is always greater than the value 0: for this input, the value of y is constantly increasing and will reach the threshold value thr , however, when $u > 0$ and $\text{prev}(y) = \text{thr}$ it does not exist any assignment that satisfies both requirement 2 and requirement 3.

Baseline Table 2 (Figure 14b). Unlike the baseline Table 1, the requirement 2 of the baseline Table 2 specifies that the value of y shall equal its previous value plus one after the startup instant. Requirements Tables generated from this baseline are *complete and inconsistent*. The Requirements Tables are complete since for every output variable, trace, and position i , there exists a precondition satisfied by the trace at that position. The Requirements Tables are inconsistent for the same reasons as the baseline Table 1.

Baseline Table 3 (Figure 14c). The requirement 1 of the Requirement Table specifies that the value 0 shall initially be assigned to the variable y . The requirement 2 specifies that the value of y shall equal its previous value plus one when the system is not in its startup instant and the value of the input u equals its previous value plus one. The requirement 3 specifies that the output variable y shall be assigned to a value lower than 23 when the system is not in its startup instant, the value of the input u differs from its previous value plus one, and the value of y is lower than the threshold value thr . Requirements Tables generated from this baseline are *incomplete and consistent*. The Requirements Tables are incomplete since none of the preconditions for the output variable y are triggered for a trace containing a position with a value of y greater than (or equal to) thr and for which the value of u does not equal its previous value plus one. The Requirements Tables are consistent since there exists an output and a timestamp interpretation, such that the corresponding trace satisfies the Requirements Table for every input interpretation: to construct this trace, it is sufficient to set the values of the output variable y according to the rules specified by the requirements of the Requirements Tables that have mutually exclusive preconditions.

Baseline Table	Completeness	Consistency
Table 1	✗	✗
Table 2	✓	✗
Table 3	✗	✓
Table 4	✓	✓

Table 3: Expected results for the completeness and consistency checks for the Requirements Tables generated from the baseline Tables from Figure 14.

Baseline Table 4 (Figure 14d). The requirements for the baseline Table 4 corresponds to the baseline Table 3, except for the requirement 3. Unlike the baseline Table 3, the precondition of the requirement 3 of the baseline Table 4 is the dual of the precondition of the requirement 2 when the system is not in its startup. Requirements Tables generated from this baseline are *complete and consistent*. The Requirements Tables are complete since there exists a precondition of one of the requirements that is satisfied for every output variable, trace, and position: the precondition of requirement 1 in the first position of the trace, and one of the preconditions of requirements 2 and 3 for the remaining positions. The Requirements Tables are consistent. The explanation corresponds to the one presented for the baseline Table 3.

Table 3 summarizes the expected results for the completeness and consistency checks for the Requirements Tables generated from the baseline Tables from Figure 14.

Our benchmark is obtained by instantiating 160 Requirements Tables: 40 for each baseline type. We instantiated the Requirements Tables by considering 40 threshold values changing from 50 to 2000 with increments of 50. The Requirements Tables generated from the baseline tables 1, 2, 3, and 4 contain flexible requirements with states since (a) they use the previous value of a variable (a.k.a. “previous state”) in the definition of the requirement (see the requirement 2 of the baseline tables) and (b) avoid the specification of the exact values for a variable of the Requirements Table (see the requirement 3 of the baseline tables).

7.2 Effectiveness of THEANO (RQ1)

To assess the effectiveness of THEANO in checking the completeness and consistency of Requirements Tables and compare the encodings from Table 2, we proceeded as follows.

Methodology. We consider each of the 160 Requirements Tables of our benchmark. For each Requirements

Table, we run THEANO by considering each of the encodings from Table 2 and considered a timeout of two hours. We executed experiments on a large computing platform with 1109 nodes, 64 cores, memory 249G or 2057500M, CPU 2 x AMD Rome 7532 2.40 GHz 256M cache L3.⁵ We repeated every run 10 times to account for variations in the performance of the platform used to run our experiment and the SMT solver, as done in similar works (e.g., [33]). Therefore, in total, we run THEANO 12800 ($10 \times 160 \times 8$) times, since we executed it ten times, for 160 Requirements Tables and eight encodings. For each encoding, the experiments were executed 1600 times (10×160).

For the bounded encodings, we had to set the value of the bound. For our Requirements Tables, setting the value of the bound higher than the threshold value also enables THEANO to determine if the Requirements Tables are complete and consistent. Therefore, for our experiments, the value of the bound considered equals the threshold value used to generate the Requirements Table plus one. Notice that a value lower than the threshold value may produce an unsound result, i.e., it may not be sufficient to detect incompleteness and inconsistencies. Engineers may not know a priori which bound values are required for the soundness of the results. We will reflect on the bound selection process in our discussion (Section 8).

To assess the effectiveness of each encoding in checking completeness and consistency, we checked the correctness of the result whenever a definitive result (complete, incomplete, consistent, or inconsistent) was returned within two hours. For the remaining cases, either THEANO did not end within two hours or an “unknown” result was returned. We measured the percentage of runs (over the total number of runs) in which THEANO returned a definitive result within two hours.

Results. Running all the experiments required several months. This time was reduced to approximately two weeks thanks to the parallelization facilities of our computing platform. Table 4 presents our results. Each row of the table reports the results related to one of the encodings. The columns of the table are divided into two sets, respectively containing the results of the completeness and the consistency checks. Each column reports the results related to each baseline table, i.e., the percentage of runs (over 1250) in which THEANO returned a definitive completeness and consistency result. For example, for the encoding BeArVs and the baseline table 2 THEANO returned a completeness result for 99.50% of the runs. The “Total” labeled columns report, for each encoding, the percentage of runs over 1600 in which THEANO returned a completeness or consistency result. The rows of the table are divided into two sets that respectively contain the results of the unbounded and the bounded encodings. In the following, we discuss the results of the unbounded and the bounded encodings.

Unbounded Encoding. For the completeness check, THEANO could confirm the completeness of the Requirements Tables generated from Tables 2 and 4. The UeUfFs, UeUfVs, UeArFs instances could confirm the completeness of 100.00% of the Requirements Tables generated from Tables 2 and 4. The UeArVs instance could respectively

confirm the completeness of 100.00% and 99.75% of the Requirements Tables generated from Tables 2 and 4. THEANO produced an *unknown* result for all the instances for the completeness check in less than two minutes for Tables 1 and 2. The SMT solver provided “*smt tactic failed to show goal to be sat/unsat (incomplete quantifiers)*” as an explanation for the unknown result.

For the consistency check, THEANO produced an *unknown* result for all the considered instances in less than one minute. The SMT solver provided “*smt tactic failed to show goal to be sat/unsat (incomplete (theory array))*” as an explanation for the unknown result. To understand which parts of the formula caused the generation of the *unknown* verdict by the SMT solver, we iteratively removed parts of the formulae from the Z3 encoding. The unbounded condition specified by the timestamp constraint (T_s) prevented the SMT solver from verifying the satisfiability of the formula. Unlikely, it is not possible to remove this constraint since it is necessary for the values of the timestamps to be monotonically increasing.

The results of the completeness and consistency checks of the unbounded encodings confirm the need for the bounded approaches.

Bounded Encoding. For the completeness check, THEANO configured with each of the bounded encodings from Table 2 produced a definitive result (complete, incomplete) within two hours for all (100%) the Requirements Tables of our benchmark in at least one of the 10 runs. Therefore, THEANO configured with one of the bounded encodings from Table 2 effectively checks the completeness of our Requirements Tables. For the Requirements Tables of our benchmark, and the configuration setup described in our methodology, our results do not show significant differences in the effectiveness of the different encodings in checking the completeness of the Requirements Tables. On average, for the fixed step semantics, the encoding BeUfFs based on uninterpreted functions is slightly more effective than the encoding BeArFs based on arrays (99.37% vs 99.12%). For the variable step semantics, the effectiveness of the encoding BeUfVs based on uninterpreted functions equals the one of the encoding BeArVs based on arrays (99.56% vs 99.56%).

For the consistency check, THEANO configured with each of the bounded encodings from Table 2 produced a definitive result within two hours for all (100%) the inconsistent Requirements Tables (Tables 1 and 2) of our benchmark in at least one of the 10 runs. Therefore, THEANO configured with one of the bounded encodings from Table 2 effectively checks the inconsistency of our Requirements Tables. For the Requirements Tables of our benchmark, and the configuration setup described in our methodology, our results do not show significant differences in the effectiveness of the different encodings in checking the inconsistency of the Requirements Tables. Additionally, our results show that THEANO could not confirm the consistency of the Requirements Tables (Tables 3 and 4) in any of the runs. For these cases, THEANO could not terminate within the timeout.

To understand why THEANO could not confirm the consistency of the Requirements Tables, we reduced the threshold values considered in our experiments to assess if, for lower threshold values, THEANO could detect that

5. <https://www.computeontario.ca/>

Table 4: Percentages of Requirements Tables generated from the Baseline Tables from Figure 14 for which the completeness and consistency produced a result for each encoding from Table 2.

Encoding	Completeness					Consistency				
	Table 1	Table 2	Table 3	Table 4	Total	Table 1	Table 2	Table 3	Table 4	Total
UeUfFs	0.00%	100.00%	0.00%	100.00%	50.00%	0.00%	0.00%	0.00%	0.00%	0.00%
UeUfVs	0.00%	100.00%	0.00%	100.00%	50.00%	0.00%	0.00%	0.00%	0.00%	0.00%
UeArFs	0.00%	100.00%	0.00%	100.00%	50.00%	0.00%	0.00%	0.00%	0.00%	0.00%
UeArVs	0.00%	100.00%	0.00%	99.75%	49.94%	0.00%	0.00%	0.00%	0.00%	0.00%
BeUfFs	99.00%	99.50%	99.00%	100.00%	99.37%	98.75%	99.5%	0.00%	0.00%	49.56%
BeUfVs	98.75%	100.00%	99.50%	100.00%	99.56%	99.00%	99.00%	0.00%	0.00%	49.50%
BeArFs	99.25%	99.50%	98.25%	99.50%	99.12%	98.25%	99.75%	0.00%	0.00%	49.50%
BeArVs	100.00%	99.50%	99.00%	99.75%	99.56%	98.75%	98.75%	0.00%	0.00%	49.87%

Table 5: Percentages of Requirements Tables generated from the Baseline Tables from Figure 14 for which the consistency produced a result for each encoding from Table 2 for threshold values within 10 and 50.

	Threshold Value								
Table	10	15	20	25	30	35	40	45	50
Table 3	100.00%	0.00%	0.00%	0.00%	50.00%	0.00%	0.00%	0.00%	15.00%
Table 4	82.50%	25.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

the Requirements Tables were consistent. Therefore, we considered the same baseline tables, generated the Requirements Tables for the threshold values from 10 to 50 with increments of five, and ran THEANO. Running these experiments required approximately 50 additional days. This time was reduced to approximately a few days thanks to the parallelization facilities of our computing platform.

For each baseline table, Table 5 presents the percentage (for the different encodings and runs) of cases in which THEANO could confirm that the Requirements Tables are complete. For the Requirements Tables generated from the baseline Table 3, the results show that the bounded encoding could confirm the completeness for all the runs for the threshold value 10, for 50% of the runs for the threshold value 30, and 15% of the runs for the threshold value 50. For the remaining cases, THEANO could not confirm the completeness of the Requirements Tables. Notice that some variability of the results is expected due to the non-determinism of the Z3 solver. For the Requirements Tables generated from the baseline Table 4, the results show the bounded encoding could confirm the completeness for all the runs for the threshold value 10, and for 25% of the runs for the threshold value 15. For the remaining cases, THEANO could not confirm the completeness of the Requirements Tables. These results confirm that for lower threshold values THEANO can also confirm the consistency of the Requirements Tables. Therefore, THEANO may be able to produce a verdict by increasing the timeout.

RQ1 - Effectiveness

THEANO could confirm the completeness of the Requirements Tables within two hours when configured with the unbounded encodings. When configured with the bounded encodings, it could confirm both the completeness and incompleteness of the Requirements Tables of our benchmark within two hours and also effectively detect inconsistent Requirements Tables.

7.3 Efficiency of THEANO (RQ2)

To assess how efficient THEANO is in checking the completeness and consistency of Requirements Tables, we want to:

- 1) Analyze the time required to confirm the completeness of the Requirements Tables for the unbounded encodings,
- 2) Understand which threshold values can be analyzed within two hours and estimate the performances engineers should expect when the threshold value increases for the bounded encodings, and
- 3) To assess how the efficiency of the bounded encodings from Table 2 compare in analyzing the completeness and consistency of Requirements Tables, we analyze how their performance compares as the threshold value increases.

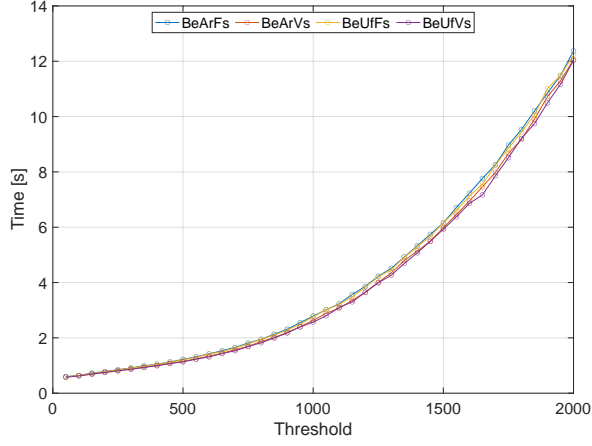
To perform these assessments, we proceeded as follows.

Methodology. We considered the results for RQ1. We considered only the runs that ended with a definitive result. Therefore, we only considered the results obtained using the bounded encodings. We analyzed the time required to compute that results.

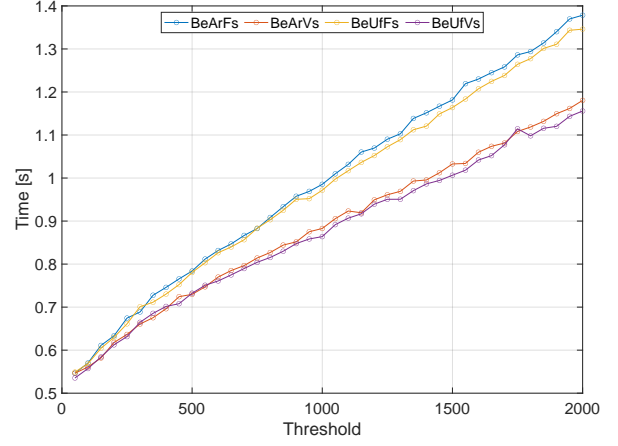
Results. In the following, we will discuss the results of the unbounded and the bounded encodings.

Unbounded Encoding. THEANO could confirm the completeness of the Requirements Tables efficiently: To confirm the completeness, THEANO required less than two minutes for all the runs ($avg=0.70s$, $min=0.48s$, $max=0.97s$, $StdDev=0.19$).

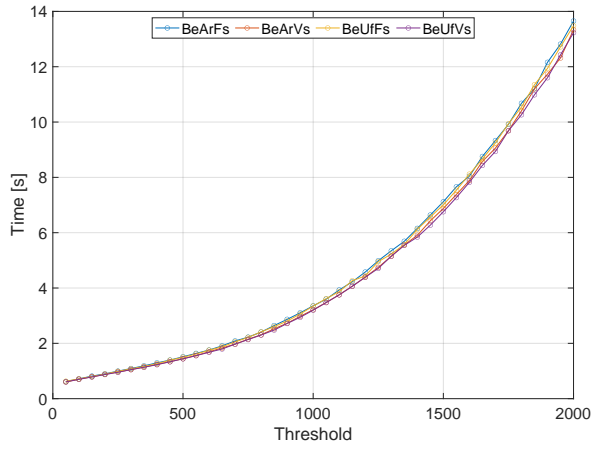
Bounded Encoding. Figures 15 and 16 present the time required by the bounded encodings to produce the complete/incomplete and inconsistent verdicts. Each subfigure is associated with one of the baseline Tables. There is no subfigure for the consistency check and Tables 3 and 4 since THEANO could not confirm the consistency of these tables in any of its runs (see Table 4). The x -axis of each diagram represents the threshold values used to generate the different Requirements Tables, i.e., each value in the x -axis refers to one Requirements Table generated using that



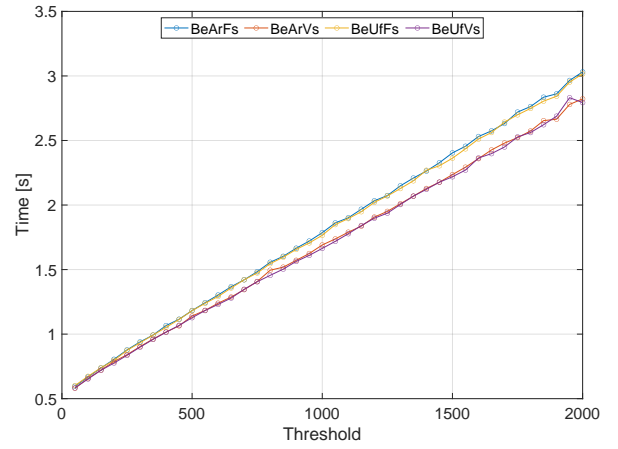
(a) Table 1: Incomplete.



(b) Table 2: Complete.

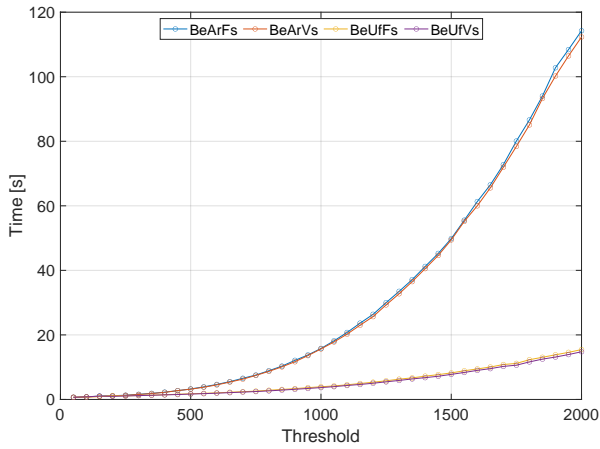


(c) Table 3: Incomplete.

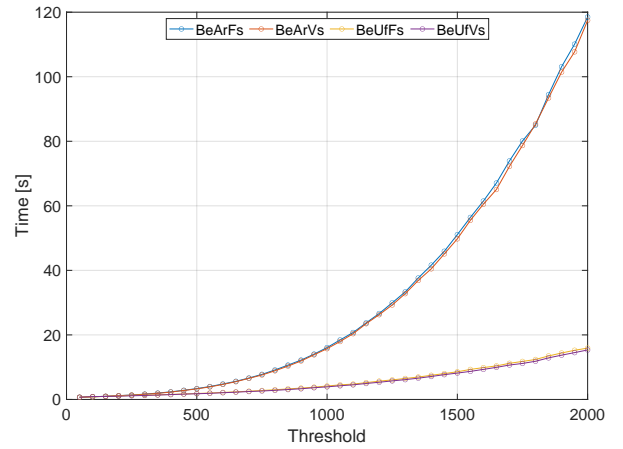


(d) Table 4: Complete.

Figure 15: Efficiency of the different encodings from Table 2 to check completeness of the Requirements Tables.



(a) Table 1: Inconsistent.



(b) Table 2: Inconsistent.

Figure 16: Efficiency of the different encodings from Table 2 to check consistency of the Requirements Tables.

Table 6: Average, minimum, maximum, and standard deviation of the time required by THEANO to confirm the consistency of the Requirements Tables generated with threshold values within 10 and 50.

Table	Measure	Threshold Value								
		10	15	20	25	30	35	40	45	50
Table 3	<i>avg</i>	1164.53	-	-	-	3032.27	-	-	-	6046.28
	<i>min</i>	2.81	-	-	-	2374.46	-	-	-	5353.86
	<i>max</i>	5388.16	-	-	-	3477.36	-	-	-	6860.71
	<i>std</i>	2039.59	-	-	-	269.54	-	-	-	507.08
Table 4	<i>avg</i>	1754.88	529.12	-	-	-	-	-	-	-
	<i>min</i>	222.95	503.51	-	-	-	-	-	-	-
	<i>max</i>	5988.96	574.10	-	-	-	-	-	-	-
	<i>std</i>	1523.09	29.90	-	-	-	-	-	-	-

"-": None of the runs could confirm the consistency of the Requirements Tables.

threshold value. The y -axis contains the average time (across the 10 times each run was repeated) required to produce a definitive result. Different lines and shapes refer to the different encodings. In the following, we discuss the results of the completeness and consistency checks.

For the completeness check (Figure 15), THEANO could return a result within 15 seconds (Figures 15a and 15c) for the Requirements Tables that are incomplete (i.e., the Requirements Tables generated from the baseline Tables 1 and 3). For those cases, THEANO could detect the satisfiability of the logic formulae generated for the Requirements Tables (see Section 5). THEANO could return a result within five seconds (Figures 15b and 15d) for the Requirements Tables that are complete (i.e., the Requirements Tables generated from the baseline tables 2 and 4). For those cases, THEANO could detect the unsatisfiability of the logic formulae generated for the Requirements Tables (see Section 5). Therefore, THEANO configured with one of the bounded encodings from Table 2 is effective in checking the completeness/incompleteness of our Requirements Tables in practical time (less than one minute) for the set of representative Requirements Tables from our benchmark.

On average (across the different runs) when configured with the BeUfFs, BeUfVs, BeArFs, and BeArVs encodings, THEANO required 2.9 ($min=0.5$, $max=16.3$, $StdDev=3.0$), 2.8 ($min=0.5$, $max=15.5$, $StdDev=2.9$), 2.9 ($min=0.5$, $max=16.8$, $StdDev=3.0$), and 2.8 ($min=0.5$, $max=16.0$, $StdDev=3.0$) seconds to check the completeness of the Requirements Tables. For the Requirements Tables that are incomplete (Figures 15a and 15c), the time required to detect the incompleteness increases exponentially with the threshold value. Additionally, the results show that, when the Requirements Tables are incomplete, there is no significant difference between the efficiency of the different encodings. For the Requirements Tables that are complete (Figures 15b and 15d), the time required to produce the completeness verdict increases linearly with the threshold value. The Wilcoxon rank sum and the Vargha-Delaney effect size tests confirm that, for the complete Requirements Tables, the encodings based on uninterpreted functions are more efficient than the ones based on arrays. However, the improvement in terms of efficiency is negligible (less than one second).

For the consistency check (Figure 16), THEANO could return a result within three minutes (Figures 16a and 16b) for the Requirements Tables that are inconsistent (i.e., the Requirements Tables generated from the baseline tables 1

and 2). For those cases, THEANO could detect the satisfiability of the logic formulae generated for the Requirements Tables (see Section 5). THEANO could not return a result within the timeout for the consistent Requirements Tables (i.e., the Requirements Tables generated from the baseline tables 3 and 4 — see Section 7.2). For those cases, THEANO could not detect the unsatisfiability of the logic formulae generated for the Requirements Tables (see Section 5). Therefore, THEANO configured with one of the bounded encodings from Table 2 is effective in finding inconsistencies in our Requirements Tables in practical time (less than three minutes) for the set of representative Requirements Tables from our benchmark.

On average (across the different runs) when configured with the BeUfFs, BeUfVs, BeArFs, and BeArVs encodings, THEANO required 5.67 ($min=0.59$, $max=19.07$, $StdDev=4.43$), 5.39 ($min=0.57$, $max=18.82$, $StdDev=4.22$), 32.32 ($min=0.60$, $max=155.60$, $StdDev=35.02$), and 31.53 ($min=0.57$, $max=155.21$, $StdDev=34.46$) seconds to detect the inconsistency of the Requirements Tables. Figure 16 presents the results obtained for the inconsistent Requirements Tables generated from the baseline tables 1 and 2. For the Requirements Tables that are inconsistent (Figures 16a and 16b), the time required to detect the inconsistencies increases exponentially with the threshold value. Additionally, the results show that, when the Requirements Tables are inconsistent, the encodings based on uninterpreted functions (BeUfFs, BeUfVs) are more efficient than the ones based on arrays (BeArFs, BeArVs). The Wilcoxon rank sum and the Vargha-Delaney effect size tests confirm that the encodings based on uninterpreted functions (BeUfFs, BeUfVs) are more efficient than the ones based on arrays (BeArFs, BeArVs). For the Requirements Tables considered in our experiment, using the encodings based on uninterpreted functions produces a significant time saving that can reach almost two minutes.

As discussed in Section 7.2, since THEANO could not confirm the consistency of the Requirements Tables within the timeout of two hours, we considered another benchmark that contained Requirements Tables generated with smaller threshold values. Table 6 reports the average, minimum, maximum, and standard deviation for the time required by THEANO to analyze the Requirements Tables with smaller threshold values. The results reported in the table confirm that there are significant variations across the different threshold values, and runs, for the time required by THEANO to prove the consistency of the Requirements

Tables. Even for small (10) threshold values, proving the consistency of the Requirements Tables may require more than one hour ($5988.96s \approx 100m$).

RQ2 - Efficiency

When THEANO was configured with the unbounded encodings, THEANO could check the completeness of Requirements Tables in practical time (less than two minutes). When THEANO was configured with the bounded encodings, THEANO could verify the completeness and incompleteness of all Requirements Tables in practical time (less than one minute). For the consistency check, THEANO configured with the bounded encodings could detect the inconsistency of all Requirements Tables in practical time (less than three minutes). However, even for Requirements Tables generated with small threshold values, THEANO configured with the bounded encodings requires significant time (more than one hour) to confirm their consistency.

8 REFLECTIONS AND THREATS TO VALIDITY

This section reflects on (i) the decidability of the completeness and consistency checking procedures, (ii) our experimental results, (iii) the selection of the bound for the bounded encoding, and (iv) the threats to the validity of our results.

Decidability. For the completeness check, when the requirements do not contain operators from Figure 3 concerning flexible requirements with states (i.e., $\text{prev}(v)$, and $\text{dur}(le) \geq c$), the formula $h_{i,\pi}(\text{pre})$ does not depend on the value considered for the position i and do not contain any quantification operator (\forall or \exists). Therefore, checking the satisfiability of $h_{cp}(rt)$ reduces to verifying the satisfiability of a set of algebraic equations ($=$) or in equations ($<$, \leq , \geq , $>$) defined over the domains of the input and output variables. Unlike requirements containing flexible requirements with states, the problem is decidable for linear and polynomial constraints over linear variables and linear constraints over integer variables. For example, finding the satisfiability of a set of linear equalities on variables defined over real numbers is decidable. For the consistency check, despite the formula $h_{i,\pi}(\text{pre})$ does not depend on the value considered for the position i , the formula $h_{cp}(rt)$ contains a first-order logic formula containing one universal quantifier ($\forall \bar{y}$). Despite checking the satisfiability of these formulae is undecidable in general, there are fragments of first-order logic for which the problem is decidable [41]. For these cases, THEANO provides an effective method for deriving the correct answer.

Experimental Results. Our experimental results assess the effectiveness and efficiency of the bounded encodings to check the completeness of Requirements Tables. For the consistency check, the encoding was effective and efficient in finding inconsistencies in the Requirements Tables but failed to show their consistency. Despite the tool may not be able to confirm consistency, it is still beneficial for engineers since, in practical applications, they can run THEANO with different (increasing) values for the bound and the timeout

searching for the inconsistency of the Requirements Tables. If inconsistencies are not found, these runs will increase their confidence in the consistency of the Requirements Tables.

Across the different encodings, the encodings that used uninterpreted functions to represent the timestamp were significantly faster than the ones that used arrays. Therefore, as also noticed by other recent research (e.g., [42], [43], [44], [33], [45], [46]), small variations of the encoding can lead to significant differences in the effectiveness and efficiency of the tools. We can not claim that our encoding is the most effective and efficient. More effective and efficient encodings (especially to confirm the consistency of the Requirements Tables) may exist and be proposed over time. For some of our Requirements Tables, the SMT solver returned an *unknown* result within the timeout. The solver provided different explanations for the unknown results (i.e., “*smt tactic failed to show goal to be sat/unsat (incomplete quantifiers)*”, and “*smt tactic failed to show goal to be sat/unsat (incomplete (theory array))*”). For our experiments, we relied on the standard configuration of the SMT solver and decided not to modify the tactics used to check for the satisfiability of the logic-based formula. Future empirical investigations can assess and compare the benefits of using different tactics to analyze the completeness and consistency of Requirements Tables.

Since a benchmark of Requirements Tables containing flexible requirements with states does not exist, we considered a benchmark containing a set of Requirements Tables that were automatically generated. However, our benchmark is representative: since (a) it was obtained by generating the Requirements Tables from a set of representative baseline tables, and (b) we considered different increasing threshold values.

Selection of the bound. Consider the translation reported in Figure 7. The bounded encoding can prove consistency and completeness only if the bound is sufficiently large. Many approaches in the literature guide engineers in selecting the values for the bounds for different applications (e.g., [47], [48], [49]). These approaches are application-specific. Analyzing how they can be adapted to support encoding the Requirements Tables proposed in this work is part of our future work.

Threats to Validity. There are several threats to the external and internal validity of our results. The Requirements Tables used to answer RQ1 and RQ2 could threaten the *external validity* of our results since they influence the effectiveness and efficiency of the completeness and consistency checks. Unfortunately, despite their industrial relevance, there does not exist any existing benchmark of Requirements Tables containing flexible requirements with states. However, (a) the baseline Requirements Tables were defined by considering the experience of the authors of the papers, which include industrial experts with significant experience in the definition and analysis of Requirements Tables, and (b) the Requirements Tables were generated from the baseline tables by considering 40 different threshold values that generate instances that are easier and harder to check with the consistency and completeness procedures. This mitigates this threat to validity. In addition, we made our benchmark publicly available to enable experiment replication. Finally, future empirical analysis can confirm or refute

our conclusions.

The value selected for the bound could threaten the internal validity of the results obtained for the bounded encodings. However, in practice, engineers can select values for the bounds based on their computational resources and the computational time available. Additionally, we selected the same bound when comparing the different bounded encodings. Therefore, our configuration does not favor any of the encodings. Developing procedures that can guide engineers in the bound selection process is part of our future work.

9 RELATED WORK

This section relates our contribution to existing works that (a) support the automated analysis of tabular requirement specifications, (b) support the automated analysis of requirements specified using formal languages, (c) use bounded solutions to analyze system specifications, and (d) approaches that check the completeness and consistency of natural language requirements.

Tabular Requirement Specifications. There exist alternative toolboxes (e.g., [14], [15]) and approaches (e.g., [11]) that support the creation, modification, and verification of the completeness and disjointedness of tabular expressions (often specified using the Software Cost Reduction — SCR — methodology [50]). However, these toolboxes rely on alternative syntax (e.g., [51], [50]) and semantics of completeness and disjointedness (e.g., [51], [11], [8], [52]) that significantly differ from the ones considered in this work. From the syntax perspective, unlike this work, existing works (e.g., [51], [50]) do not support flexible requirements with states since they do not support both the operator **prev** and **dur**. From the semantic perspective, the definitions of completeness and consistency considered by this work, which match the one considered by Simulink Requirements Tables [20], differ from the semantics of completeness and disjointedness proposed in other works (e.g., [51], [11], [8]). We discuss the differences of our definitions in the following. For the definition of completeness, the difference is as follows. Unlike Definition 5 that defines a Requirements Table rt to be complete if for every output variable $y \in \mathcal{Y}$, trace $\pi \in \Pi$, and position i , there exists a precondition $pre \in Pre(rt, y)$ satisfied by the trace π at position i , other existing definitions specify that a Tabular Expression is complete if the formula obtained by computing the disjunction of the preconditions is always true, that is the preconditions cover the entire input domain. That is, the definition considered in this work depends on the output variables that are used by the Requirements Tables, traces, and their positions and not only on the preconditions of the Requirements Tables. For the definition of consistency, the difference is as follows. Unlike Definition 6 that defines a Requirements Table rt to be consistent if for every input interpretation ι_U there exists an output ι_Y and a timestamp ι_τ interpretation, such that the corresponding trace π satisfies the Requirements Table, other existing definitions specify that a Tabular Expression is disjoint if, for every pair of preconditions, the formula obtained by computing the conjunction of the preconditions is unsatisfiable, that is two preconditions can not be active at the same time. Therefore, the definition of consistency

considered in this work enables more than one precondition to be active as long as for every input interpretation ι_U there exists an output ι_Y and a timestamp ι_τ interpretation, such that the corresponding trace π satisfies the Requirements Table.

Pang et al. [53] proposed an approach that supports analyzing timing requirements within tabular expression. Specifically, the approach can support requirements that use an infix *Held For* operator, which is similar to the operator **dur**, and the operator *Timer_I* measuring how long a monitored condition has been enabled. Unlike our work, Pang et al. consider the semantics of completeness and disjointedness previously discussed and do not support the **prev** operator. Therefore, there are significant differences with our work: (a) considering the semantics of completeness and consistency used by the Requirements Tables makes our solution consistent with definitions used by industrial practitioners that use the Simulink toolbox, and (b) considering the **prev** operator (and our semantics) introduces significant technical difficulties (e.g., it makes the problem of checking the completeness and consistency of the tabular requirements undecidable). To address these problems, THEANO proposed multiple encodings and empirically assessed and compared the results obtained by these encodings.

Existing works from the literature on tabular requirement specifications (a) analyzed how to simplify tabular expressions to facilitate testing driven by the standard ISO 26262 [54], (b) proposed approaches that enable their usage for supporting rigorous development of critical systems [55], [56], (c) developed tools that generate tabular expressions representing requirements from textual requirements documents [57], and the IEC 61131-3 Function Block Diagrams [58], (d) designed tools that generate formal models from tabular expressions [13], and (e) described methods that automatically generate test cases from tabular expressions [59].

Finally, several works assessed the benefits of tabular requirement specification on industrial applications [60], [61], [62], [63], [64].

Formal Languages Support. Several logic-based (e.g., [65], [66], [67], [68], [69], [70], [71], [72], [73], [74]), and domain-specific (e.g., [75], [76], [77], [78], [79], [80], [81], [82]) languages, discussed by recent surveys on the topic (e.g. [83], [84], [85], [86]), were proposed by the research community to specify and analyze software requirements. Automated analyses support engineers in several activities including refactoring requirements [87], simplifying requirements formalization [88], and analyzing the applicability of existing tools on industrial CPS [89], [90], and their consistency [91], [92], [93], [94]. Unlike these works, THEANO considers requirements expressed using tabular expressions defined by Simulink Requirements Tables and provides automated support to check their completeness and consistency.

Bounded verification. Bounded verification has been extensively used to analyze system specifications, for example by proving that a model of a system satisfies its requirements (e.g., [44], [95]), verifying legal compliance (e.g., [96]) extracting topological proofs showing why a property holds on a system (e.g., [42], [43]), validate B and Event-B models (e.g., [97]), verifying safety of syn-

chronous fault-tolerant algorithms (e.g., [98]), analyze Alloy specifications (e.g., [99], [100], [101], [99]), find models that satisfy temporal-logic specifications [102], [103], [104], [105] and trace-checking (e.g., [106], [33]). Unlike these works, THEANO uses bounded verification to verify the consistency and completeness of Requirements Tables.

Natural Language. Several works studied consistency and completeness of requirements expressed in natural language [107], [108], [109], [110], [111], [112], [113] often relying on domain modeling (e.g., [114], [115]). These works are discussed and analyzed by many surveys and studies on the topic (e.g. [116], [117], [118], [119], [116], [120], [121], [122], [120], [123], [124], [125]). Unlike these works, THEANO supports formalized requirements and relies on satisfiability checking to verify the completeness and consistency of requirements specifications.

10 CONCLUSIONS

This paper proposes an approach to check the completeness and consistency of Tabular Requirements containing flexible requirements with state expressed using Simulink Requirements Tables. To define our approach, we formalized the syntax and semantics of Requirements Tables and the notions of completeness and consistency, we proposed eight encodings from two categories (bounded and unbounded) that check the completeness and consistency of Requirements Tables, and we implemented THEANO, a tool that supports our approach.

We considered a benchmark of 160 Requirements Tables and empirically assessed the effectiveness and efficiency of our approach. Our results show that, unlike the encodings from the unbounded category that could only confirm the completeness of the Requirements Tables within two hours, the encoding from the bounded category can also detect incompleteness in the Requirements Tables. For the consistency check, the encodings from the unbounded category could confirm neither the consistency nor the inconsistency of the Requirements Tables. Differently, the bounded encodings could detect the inconsistency of the Requirements Tables within the timeout and, for Requirements Tables generated with smaller threshold values, they were also able to confirm their consistency within two hours.

ACKNOWLEDGMENT

This research was enabled in part by support provided by Compute Ontario (www.computeontario.ca) and Compute Canada (www.computecanada.ca).

We thank Prof. Mark Lawford, Prof. Richard F. Paige, Prof. Alan Wasssyng, and Dr. Srinath Avadhanula for their feedback during the early stage of this work.

REFERENCES

- [1] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan, "S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 254–257.
- [2] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *Computer Aided Verification*. Springer, 2010, pp. 167–170.
- [3] B. Hoxha, N. Mavridis, and G. Fainekos, "Vispec: A graphical tool for elicitation of mtl requirements," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2015, pp. 3486–3492.
- [4] D. Ničković, O. Lebeltel, O. Maler, T. Ferrère, and D. Ulus, "Amt 2.0: qualitative and quantitative trace analysis with extended signal temporal logic," *International Journal on Software Tools for Technology Transfer*, vol. 22, pp. 741–758, 2020.
- [5] D. Ničković and T. Yamaguchi, "Rtam: Online robustness monitors from stl," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2020, pp. 564–571.
- [6] A. Rajhans, A. Mavrommati, P. J. Mosterman, and R. G. Valenti, "Specification and runtime verification of temporal assessments in simulink," in *International Conference on Runtime Verification*. Springer, 2021, pp. 288–296.
- [7] (2023) Simulink test. [Online]. Available: <https://www.mathworks.com/products/simulink-test.html>
- [8] K. L. Heninger, "Specifying software requirements for complex systems: New techniques and their application," *Transactions on Software Engineering*, no. 1, pp. 2–13, 1980, IEEE.
- [9] S. R. Faulk, "State determination in hard-embedded systems." 1990.
- [10] S. Meyers and S. White, "Software requirements methodology and tool study for a6-e technology transfer," 1983.
- [11] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," *Transactions on Software Engineering and Methodology*, vol. 5, no. 3, pp. 231–261, 1996, ACM.
- [12] L. Pang, C. Wang, M. Lawford, and A. Wasssyng, "Formalizing and verifying function blocks using tabular expressions and PVS," in *Formal Techniques for Safety-Critical Systems*, vol. 419. Springer, 2013, pp. 125–141.
- [13] N. K. Singh, M. Lawford, T. S. Maibaum, and A. Wasssyng, "Use of tabular expressions for refinement automation," in *Model and Data Engineering*. Springer, 2017, pp. 167–182.
- [14] D. K. Peters, M. Lawford, and B. T. y. Widemann, "An ide for software development using tabular expressions," in *Conference of the Center for advanced studies on Collaborative research*, 2007, pp. 248–251, IBM.
- [15] C. Eles and M. Lawford, "A tabular expression toolbox for matlab/simulink," in *NASA Formal Methods*. Springer, 2011, pp. 494–499.
- [16] J. Crow and B. Di Vito, "Formalizing space shuttle software requirements: Four case studies," *Transactions on software engineering and methodology*, vol. 7, no. 3, pp. 296–332, 1998, ACM.
- [17] A. Wasssyng, M. S. Lawford, and T. S. Maibaum, "Software certification experience in the canadian nuclear industry: Lessons for the future," in *International Conference on Embedded Software*. ACM, 2011, p. 219–226.
- [18] MathWorks. (2023) Use a Requirements Table Block to Create Formal Requirements. [Online]. Available: <https://www.mathworks.com/help/slrequirements/ug/use-requirements-table-block.html>
- [19] (2023) Requirements toolbox. [Online]. Available: <https://www.mathworks.com/products/requirements-toolbox.html>
- [20] MathWorks. (2023) Identify Inconsistent and Incomplete Formal Requirement Sets. [Online]. Available: <https://www.mathworks.com/help/slrequirements/ug/check-requirements-table-block.html>
- [21] M. Chechik and J. Gannon, "Automatic analysis of consistency between requirements and designs," *Transactions on Software Engineering*, vol. 27, no. 7, pp. 651–672, 2001, IEEE.
- [22] (2023) Identify inconsistent and incomplete formal requirement sets. [Online]. Available: <https://www.mathworks.com/help/slrequirements/ug/check-requirements-table-block.html>
- [23] (2023) Compare solvers. [Online]. Available: <https://www.mathworks.com/help/simulink/ug/compare-solvers.html>
- [24] antlr. (2023) antlr. [Online]. Available: <https://www.antlr.org/>
- [25] (2020) Z3. [Online]. Available: <https://ericpony.github.io/z3py-tutorial/guide-examples.htm>
- [26] (2020) Z3. [Online]. Available: <https://github.com/Z3Prover/z3>
- [27] (2023) Theano. [Online]. Available: <https://github.com/foselab/Theano/>
- [28] (2023) Simulink is for model-based design. [Online]. Available: <https://www.mathworks.com/products/simulink>
- [29] D. Hilbert, "Mathematical problems," in *Mathematics*. Chapman and Hall/CRC, 2019, pp. 273–278.

- [30] L. J. Mordell, *Diophantine equations*. Academic press, 1969.
- [31] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions," in *Computer Aided Verification*. Springer, 2002, pp. 78–92.
- [32] (2023) Smt-lib the satisfiability modulo theories library. [Online]. Available: <https://smtlib.cs.uiowa.edu/logics.shtml>
- [33] C. Menghi, E. Viganò, D. Bianculli, and L. C. Briand, "Trace-checking cps properties: Bridging the cyber-physical gap," in *International Conference on Software Engineering*. IEEE, 2021, pp. 847–859.
- [34] Wikipedia. (2023) ANTLR. [Online]. Available: <https://en.wikipedia.org/wiki/ANTLR>
- [35] Tom Everett. (2023) MATLAB Grammar. [Online]. Available: <https://github.com/antlr/grammars-v4/blob/master/matlab/matlab.g4>
- [36] "ACM SIGPLAN - Programming Languages Software Award," <http://www.sigplan.org/Awards/Software/>, 2020.
- [37] "ETAPS 2018 Test of Time Award," <https://etaps.org/about/test-of-time-award/test-of-time-award-2018>, 2020.
- [38] David Wingate. (2023) mparser. [Online]. Available: <https://www.mathworks.com/matlabcentral/fileexchange/32769-mparser>
- [39] antlr3. (2023) ANTLRv3 parsing library for C. [Online]. Available: <https://www.antlr3.org/>
- [40] "Identify inconsistent and incomplete formal requirement sets," 2023. [Online]. Available: <https://www.mathworks.com/help/slrequirements/ug/check-requirements-table-block.html>
- [41] Wikipedia. (2023) Decidability (logic). [Online]. Available: [https://en.wikipedia.org/wiki/Decidability_\(logic\)](https://en.wikipedia.org/wiki/Decidability_(logic))
- [42] C. Menghi, A. M. Rizzi, A. Bernasconi, and P. Spoletini, "TORPEDO: witnessing model correctness with topological proofs," *Formal Aspects of Computing*, vol. 33, no. 6, pp. 1039–1066, 2021, springer.
- [43] C. Menghi, A. M. Rizzi, and A. Bernasconi, "Integrating topological proofs with model checking to instrument iterative design," in *Fundamental Approaches to Software Engineering*. Springer, 2020, pp. 53–74.
- [44] C. Menghi, M. M. Bersani, M. Rossi, and P. S. Pietro, "Model checking MITL formulae on timed automata: A logic-based approach," *Transactions on Computational Logic*, vol. 21, no. 3, 2020, ACM.
- [45] M. M. P. Kallehbasti, M. Rossi, and L. Baresi, "On how bit-vector logic can help verify LTL-based specifications," *Transactions on Software Engineering*, vol. 48, no. 4, pp. 1154–1168, 2020, IEEE.
- [46] L. Baresi, M. M. P. Kallehbasti, and M. Rossi, "Efficient scalable verification of LTL specifications," in *International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 711–721.
- [47] A. Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan, "Linear encodings of bounded LTL model checking," *Logical Methods in Computer Science*, vol. 2, 2006, episciences.org.
- [48] E. Clarke, D. Kroening, J. Ouaknine, and O. Strichman, "Completeness and complexity of bounded model checking," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2004, pp. 85–96.
- [49] D. Kroening, J. Ouaknine, O. Strichman, T. Wahl, and J. Worrell, "Linear completeness thresholds for bounded model checking," in *Computer Aided Verification*. Springer, 2011, pp. 557–572.
- [50] C. Heitmeyer, "Software cost reduction," *Encyclopedia of software engineering*, vol. 2, pp. 1374–1380, 2002.
- [51] Y. Jin and D. L. Parnas, "Defining the meaning of tabular mathematical expressions," *Science of Computer Programming*, vol. 75, no. 11, pp. 980–1000, 2010, Elsevier.
- [52] A. Wasssyng and R. Janicki, "Tabular expressions in software engineering," in *Proceedings of ICSSEA*, vol. 3, 2003, pp. 1–46.
- [53] L. Pang, C.-W. Wang, M. Lawford, A. Wasssyng, J. Newell, V. Chow, and D. Tremaine, "Formal verification of real-time function blocks using pvs," *arXiv preprint arXiv:1506.03557*, 2015.
- [54] M. Bialy, M. Lawford, V. Pantelic, and A. Wasssyng, "A methodology for the simplification of tabular designs in model-based development," in *FME Workshop on Formal Methods in Software Engineering*. IEEE/ACM, 2015, pp. 47–53.
- [55] N. K. Singh, M. Lawford, T. S. Maibaum, and A. Wasssyng, "A formal approach to rigorous development of critical systems," *Journal of Software: Evolution and Process*, vol. 33, no. 4, p. e2334, 2021, wiley Online Library.
- [56] C.-W. Wang, J. S. Ostroff, and S. Hudon, "Precise documentation and validation of requirements," in *Formal Techniques for Safety-Critical Systems*. Springer, 2014, pp. 262–279.
- [57] Y. Jin, J. Zhang, W. Hao, P. Ma, Y. Zhang, H. Zhao, and H. Mei, "A concern-based approach to generating formal requirements specifications," *Frontiers of Computer Science in China*, vol. 4, pp. 162–172, 2010, Springer.
- [58] J. Newell, L. Pang, D. Tremaine, A. Wasssyng, and M. Lawford, "Formal translation of iec 61131-3 function block diagrams to pvs with nuclear application," in *NASA Formal Methods*. Springer, 2016, pp. 206–220.
- [59] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *European Software Engineering Conference Held Jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE. Springer-Verlag, 1999, p. 146–162.
- [60] M. Lawford, J. McDougall, P. Froebel, and G. Moum, "Practical application of functional and relational methods for the specification and verification of safety critical software," in *International Conference on Algebraic Methodology and Software Technology*. Springer, 2000, pp. 73–88.
- [61] R. Bharadwaj and C. Heitmeyer, "Developing high assurance avionics systems with the scr requirements method," in *Digital Avionics Systems Conference*, vol. 1. IEEE, 2000, pp. 1D1–1.
- [62] J. Newell, L. Pang, D. Tremaine, A. Wasssyng, and M. Lawford, "Translation of IEC 61131-3 function block diagrams to pvs for formal verification with real-time nuclear application," *Journal of Automated Reasoning*, vol. 60, pp. 63–84, 2018, Springer.
- [63] L. Heitmeyer and R. D. Jeffords, "Applying a formal requirements method to three nasa systems: Lessons learned," in *Aerospace Conference*. IEEE, 2007, pp. 1–10.
- [64] T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. Shore, "Software requirements for the a-7e aircraft," Technical Report NRL-9194, Naval Research Lab., Wash., DC, Tech. Rep., 1992.
- [65] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer, 2004, pp. 152–166.
- [66] L. Brim, P. Dluhoš, D. Šafránek, and T. Vojtisek, "STL*: Extending signal temporal logic with signal-value freezing operator," *Information and computation*, vol. 236, pp. 52–67, 2014, Elsevier.
- [67] C. Menghi, S. Nejati, K. Gaaloul, and L. C. Briand, "Generating automated and online test oracles for simulink models with continuous and uncertain behaviors," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, p. 27–38.
- [68] C. Menghi, E. Viganò, D. Bianculli, and L. C. Briand, "Trace-checking cps properties: Bridging the cyber-physical gap," in *International Conference on Software Engineering*. IEEE/ACM, 2021, pp. 847–859.
- [69] A. Bakhirkin, T. Ferrère, T. A. Henzinger, and D. Ničković, "Keynote: The first-order logic of signals," in *International Conference on Embedded Software*, 2018, pp. 1–10.
- [70] R. Alur and T. A. Henzinger, "Real-time logics: complexity and expressiveness," *Information and Computation*, vol. 104, no. 1, pp. 35–77, 1993, Elsevier.
- [71] T. Ferrère, O. Maler, and D. Ničković, "Mixed-time signal temporal logic," in *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 2019, pp. 59–75.
- [72] A. Cimatti, M. Roveri, and S. Tonetta, "HRELTL: A temporal logic for hybrid systems," *Information and Computation*, vol. 245, pp. 54–71, 2015, Elsevier.
- [73] C. Ghezzi, D. Mandrioli, and A. Morzenti, "TRIO: A logic language for executable specifications of real-time systems," *Journal of Systems and software*, vol. 12, no. 2, pp. 107–123, 1990, Elsevier.
- [74] C. Menghi, C. Tsigkanos, M. Askarpour, P. Pelliccione, G. Vázquez, R. Calinescu, and S. García, "Mission specification patterns for mobile robots: Providing support for quantitative properties," *Transactions on Software Engineering*, vol. 49, no. 4, pp. 2741–2760, 2023, IEEE.
- [75] D. Giannakopoulou, T. Pressburger, A. Mavridou, and J. Schumann, "Generation of formal requirements from structured natural language," in *Requirements Engineering: Foundation for Software Quality*. Springer, 2020, pp. 19–35.
- [76] D. Giannakopoulou, A. Mavridou, J. Rhein, T. Pressburger, J. Schumann, and N. Shi, "Formal requirements elicitation with

- FRET," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*, ser. CEUR Workshop Proceedings, vol. 2584, 2020, cEUR-WS.org.
- [77] J. Anderson, M. Hekmatnejad, and G. Fainekos, "PyFoReL: A domain-specific language for formal requirements in temporal logic," in *International Requirements Engineering Conference*. IEEE, 2022, pp. 266–267.
- [78] A. Arrieta, J. A. Agirre, and G. Sagardui, "A tool for the automatic generation of test cases and oracles for simulation models based on functional requirements," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2020, pp. 1–5.
- [79] C. Boufaied, C. Menghi, D. Bianculli, L. Briand, and Y. I. Parache, "Trace-checking signal-based temporal properties: A model-driven approach," in *International Conference on Automated Software Engineering*, 2020, pp. 1004–1015, IEEE/ACM.
- [80] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger, "Specification patterns for robotic missions," *Transactions on Software Engineering*, vol. 47, no. 10, pp. 2208–2224, 2021, IEEE.
- [81] P. Filipovikj, G. Rodriguez-Navas, M. Nyberg, and C. Seculeanu, "Automated SMT-based consistency checking of industrial critical requirements," *SIGAPP Applied Computing Review*, vol. 17, no. 4, pp. 15–28, 2018, ACM.
- [82] D. F. Mendonça, G. N. Rodrigues, R. Ali, V. Alves, and L. Baresi, "GODA: A goal-oriented requirements engineering framework for runtime dependability analysis," *Information and Software Technology*, vol. 80, pp. 245–264, 2016, Elsevier.
- [83] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, "Formal specification and verification of autonomous robotic systems: A survey," *Computing Surveys*, vol. 52, no. 5, 2019.
- [84] D. Weyns, M. U. Iftikhar, D. G. De La Iglesia, and T. Ahmad, "A survey of formal methods in self-adaptive systems," in *International c* conference on computer science and software engineering*. ACM, 2012, pp. 67–79.
- [85] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan, "Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications," *Lectures on Runtime Verification: Introductory and Advanced Topics*, pp. 135–175, 2018, Springer.
- [86] C. Sánchez, G. Schneider, W. Ahrendt, E. Bartocci, D. Bianculli, C. Colombo, Y. Falcone, A. Francalanza, S. Krstić, J. M. Lourenço *et al.*, "A survey of challenges for runtime verification from advanced application domains (beyond software)," *Formal Methods in System Design*, vol. 54, pp. 279–335, 2019, Springer.
- [87] M. Farrell, M. Luckcuck, O. Sheridan, and R. Monahan, "Towards refactoring fretish requirements," in *NASA Formal Methods Symposium*. Springer, 2022, pp. 272–279.
- [88] C. M. d. Ferro, A. Mavridou, M. Dille, and F. Martins, "Simplifying requirements formalization for resource-constrained mission-critical software," in *International Conference on Dependable Systems and Networks Workshops*. IEEE/IFIP, 2023, pp. 263–266.
- [89] M. Farrell, M. Luckcuck, O. Sheridan, and R. Monahan, "Fretting about requirements: formalised requirements for an aircraft engine controller," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2022, pp. 96–111.
- [90] M. Luckcuck, M. Farrell, O. Sheridan, and R. Monahan, "A methodology for developing a verifiable aircraft engine controller from formal requirements," in *Aerospace Conference*. IEEE, 2022, pp. 1–12.
- [91] J. S. Becker, "Analyzing consistency of formal requirements," *Electronic Communications of the EASST*, vol. 76, 2019.
- [92] M. P. Heimdahl and N. G. Leveson, "Completeness and consistency analysis of state-based requirements," in *International Conference on Software Engineering*. ACM, 1995, pp. 3–14.
- [93] N. Mahmud, C. Seculeanu, and O. Ljungkrantz, "ReSA Tool: Structured requirements specification and sat-based consistency-checking," in *Federated Conference on Computer Science and Information Systems*. IEEE, 2016, pp. 1737–1746.
- [94] J. Bendik, "Consistency checking in requirements analysis," in *International Symposium on Software Testing and Analysis*. ACM, 2017, p. 408–411.
- [95] S. Holzer, P. Frangoudis, C. Tsigkanos, and S. Dustdar, "Smt-as-a-service for fog-supported cyber-physical systems," in *International Conference on Distributed Computing and Networking*. ACM, 2024, p. 154–163.
- [96] N. Feng, L. Marsso, M. Sabetzadeh, and M. Chechik, "Early verification of legal compliance via bounded satisfiability checking," in *Computer Aided Verification*. Springer-Verlag, 2023, p. 374–396.
- [97] J. Schmidt and M. Leuschel, "SMT solving for the validation of b and event-b models," *International Journal on Software Tools for Technology Transfer*, vol. 24, no. 6, pp. 1043–1077, 2022, Springer.
- [98] I. Stoilkovska, I. Konnov, J. Widder, and F. Zuleger, "Verifying safety of synchronous fault-tolerant algorithms by bounded model checking," *International Journal on Software Tools for Technology Transfer*, vol. 24, no. 1, pp. 33–48, 2022, Springer.
- [99] A. Cunha, "Bounded model checking of temporal formulas with alloy," in *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 2014, pp. 303–308.
- [100] S. G. Brida, G. Regis, G. Zheng, H. Bagheri, T. Nguyen, N. Aguirre, and M. Frias, "Bounded exhaustive search of alloy specification repairs," in *International Conference on Software Engineering*. IEEE, 2021, pp. 1135–1147.
- [101] A. Milicevic, J. P. Near, E. Kang, and D. Jackson, "Alloy*: A general-purpose higher-order relational constraint solver," *Formal Methods in System Design*, vol. 55, pp. 1–32, 2019, Springer.
- [102] N. Macedo, J. Brunel, D. Chemouil, and A. Cunha, "Pardinus: a temporal relational model finder," *Journal of Automated Reasoning*, vol. 66, no. 4, pp. 861–904, 2022, Springer.
- [103] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Handbook of satisfiability*, vol. 185, no. 99, pp. 457–481, 2009.
- [104] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal methods in system design*, vol. 19, pp. 7–34, 2001, Springer.
- [105] M. M. Bersani, M. Rossi, and P. San Pietro, "An smt-based approach to satisfiability checking of mitl," *Information and Computation*, vol. 245, pp. 72–97, 2015.
- [106] M. M. Bersani, D. Bianculli, C. Ghezzi, S. Krstić, and P. San Pietro, "Smt-based checking of soloist over sparse traces," in *Fundamental Approaches to Software Engineering: 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 17*. Springer, 2014, pp. 276–290.
- [107] V. Bertram, H. Kausch, E. Kusmenko, H. Nqiri, B. Rumpe, and C. Venhoff, "Leveraging natural language processing for a consistency checking toolchain of automotive requirements," in *International Requirements Engineering Conference*. IEEE, 2023, pp. 212–222.
- [108] M. Kamalrudin, J. Hosking, and J. Grundy, "MaramaAIC: tool support for consistency management and validation of requirements," *Automated software engineering*, vol. 24, pp. 1–45, 2017, Springer.
- [109] D. Luitel, S. Hassani, and M. Sabetzadeh, "Using language models for enhancing the completeness of natural-language requirements," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2023, pp. 87–104.
- [110] F. Dalpiaz, I. Van der Schalk, and G. Lucassen, "Pinpointing ambiguity and incompleteness in requirements engineering via information visualization and nlp," in *Requirements Engineering: Foundation for Software Quality*. Springer, 2018, pp. 119–135.
- [111] I. Menzel, M. Mueller, A. Gross, and J. Doerr, "An experimental comparison regarding the completeness of functional requirements specifications," in *International Requirements Engineering Conference*. IEEE, 2010, pp. 15–24.
- [112] A. Ferrari, F. dell'Orletta, G. O. Spagnolo, and S. Gnesi, "Measuring and improving the completeness of natural language requirements," in *Requirements Engineering: Foundation for Software Quality*. Springer, 2014, pp. 23–38.
- [113] A. Russo, B. Nuseibeh, and J. Kramer, "Restructuring requirements specifications for managing inconsistency and change: a case study," in *International Symposium on Requirements Engineering*. IEEE, 1998, pp. 51–60.
- [114] C. Arora, M. Sabetzadeh, and L. C. Briand, "An empirical study on the potential usefulness of domain models for completeness checking of requirements," *Empirical Software Engineering*, vol. 24, pp. 2509–2539, 2019, Springer.
- [115] D. Zowghi and V. Gervasi, "The three Cs of requirements: consistency, completeness, and correctness," in *International Workshop on Requirements Engineering: Foundations for Software Quality*, 2002, pp. 155–164.

- 1992 [116] M. Kamalrudin and S. Sidek, "A review on software require-
1993 ments validation and consistency management," *International*
1994 *journal of software engineering and its applications*, vol. 9, no. 10,
1995 pp. 39–58, 2015.
- 1996 [117] I. Hadar, A. Zamansky, and D. M. Berry, "The inconsistency
1997 between theory and practice in managing inconsistency in re-
1998 quirements engineering," *Empirical Software Engineering*, vol. 24,
1999 no. 6, pp. 3972–4005, 2019, Springer.
- 2000 [118] R. Delima, K. Mustofa, A. K. Sari *et al.*, "Automatic requirements
2001 engineering: Activities, methods, tools, and domains—a system-
2002 atic literature review," *Jurnal RESTI (Rekayasa Sistem dan Teknologi*
2003 *Informasi)*, vol. 7, no. 3, pp. 564–578, 2023.
- 2004 [119] M. Kamalrudin, N. Mustafa, and S. Sidek, "A preliminary study:
2005 Challenges in capturing security requirements and consistency
2006 checking by requirement engineers," *Journal of Telecommunication,*
2007 *Electronic and Computer Engineering*, vol. 10, no. 1-7, pp. 5–9, 2018.
- 2008 [120] D. Zowghi and V. Gervasi, "On the interplay between consis-
2009 tency, completeness, and correctness in requirements evolution,"
2010 *Information and Software technology*, vol. 45, no. 14, pp. 993–1009,
2011 2003.
- 2012 [121] A. Chattopadhyay, G. Malla, N. Niu, T. Bhowmik, and
2013 J. Savolainen, "Completeness of natural language requirements:
2014 A comparative study of user stories and feature descriptions,"
2015 in *International Conference on Information Reuse and Integration for*
2016 *Data Science*. IEEE, 2023, pp. 52–57.
- 2017 [122] G. D. Hadad, C. S. Litvak, J. H. Doorn, and M. Ridao, "Dealing
2018 with completeness in requirements engineering," in *Encyclopedia*
2019 *of Information Science and Technology, Third Edition*. IGI Global,
2020 2015, pp. 2854–2863.
- 2021 [123] B. Nuseibeh, S. Easterbrook, and A. Russo, "Leveraging in-
2022 consistency in software development," *Computer*, vol. 33, no. 4, pp.
2023 24–29, 2000.
- 2024 [124] A. Russo, B. Nuseibeh, and J. Kramer, "Restructuring require-
2025 ments specifications for managing inconsistency and change: A
2026 case study," in *International Symposium on Requirements Engineer-*
2027 *ing*. IEEE, 1998, pp. 51–60.
- 2028 [125] B. Nuseibeh and S. Easterbrook, "Requirements engineering: A
2029 roadmap," in *Conference on The Future of Software Engineering*.
2030 ACM, 2000, p. 35–46.