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

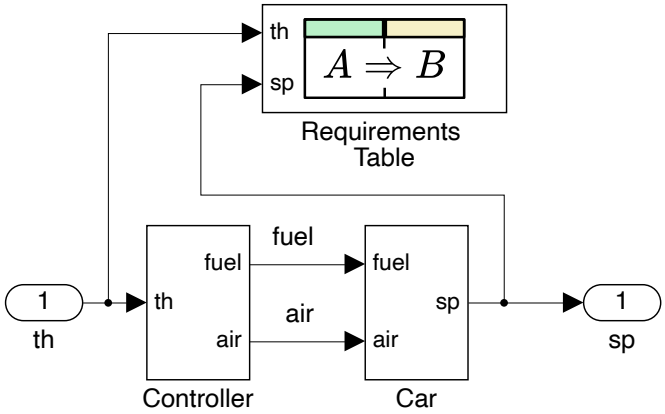
Claudio Menghi, Eugene Balai, Darren Valovcin, Christoph Sticksel, Akshay Rajhans

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. We overcome this limitation by considering Requirements Tables with both stateless and stateful requirements. 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. We also assessed the usefulness of THEANO in checking the consistency and completeness of 14 versions of a Requirements Table for a practical example from the automotive domain. Across these 14 versions, THEANO could effectively detect two inconsistent and five incomplete Requirements Tables reporting a problem (inconsistency or incompleteness) for 50% (7 out of 14) versions of the Requirements Table.

Index Terms—Requirements Tables, Tabular Requirements, Tabular Expressions, Completeness, Consistency, Requirements, Cyber-physical Systems

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].



(a) Simulink Model.

ID	Pre.	Post.
1	$\texttt{th} > 0$	$\texttt{sp} = \textbf{prev}(\texttt{sp}) + \texttt{th}$
2	$\textbf{prev}(\texttt{sp}) = 160$	$\texttt{sp} < 160$
3	$\texttt{th} < 0$	$\texttt{sp} < \textbf{prev}(\texttt{sp})$

(b) Requirements Table.

Fig. 1: Illustrative Example.

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

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. Sticksel, and A. Rajhans are with MathWorks, MA, USA - email: {ebalai, dvalovci, cstickse, arajhans}@mathworks.com

format further specifies that the *set* of three requirements are together satisfied.

Checking for the *completeness and consistency* of Requirements helps gaining confidence in their correctness. Developing effective techniques that support software engineers in checking the completeness and consistency of requirements is a well-established software engineering problem [20], [21], [22], [23], [24] pivotal for tabular requirements such as the ones expressed using Requirements Tables [25], [11], [8], [26]. 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 [27]. Analysis tools for tabular requirements specifications help software engineers check the consistency and completeness of requirements (e.g., [13], [28]). For example, Simulink supports identifying inconsistent and incomplete requirements [29], among other checks, under some syntactic conditions on the requirements.

This paper considers the problem of checking the completeness and consistency of a specific class of tabular requirement specifications containing flexible and stateful requirements and expressed using Requirements Tables. We introduce this type of tabular requirements and their challenges in the following.

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*. For example, the second requirement from the Requirements Table in Figure 1b is flexible since, when the precondition is satisfied ($\texttt{prev}(\texttt{sp}) = 160$), its postcondition ($\texttt{sp} < 160$) does not specify the exact value the speed variable ($\texttt{sp}$) should assume.

Stateful requirements use the previous value of a variable (a.k.a. "previous state") in the specification of a requirement. This enables specifying temporal behaviors of a system. For example, the first requirement from the Requirements Table in Figure 1b is a stateful requirement, since, when the precondition is satisfied ($\texttt{th} > 0$), its postcondition ($\texttt{sp} = \texttt{prev}(\texttt{sp}) + \texttt{th}$) increments the value of the variable ($\texttt{sp}$) from its previous value ($\texttt{prev}(\texttt{sp})$) by a value equal to the value of the throttle ($\texttt{th}$). That is, the value a variable assumes depends on the value assumed by a variable in the previous step (a.k.a. state).

In practice, requirements often contain at least some number of flexible and stateful requirements. 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. 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 [30]. Defining the syntax and semantics of Requirements Tables is a significant contribution as they rigorously specify what can be written by engineers (syntax) and its meaning (semantics).

(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. Our formal definition contributes by precisely and unambiguously defining the notions of completeness and consistency.

(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. These encodings contribute by providing the first procedure to check completeness and consistency Requirements Tables with flexible and stateful requirements.

(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 [31] (v4.5) component that translates the requirements into a logical formula executable from a Python script that uses the Z3Py [32] Python API to run the SMT solver Z3 [33]. We decided to use the SMT since it is a mature technology that can determine whether a mathematical formula is satisfiable and has been shown to be effective in solving similar problems (e.g., [34], [35], [36], [37], [38], [39]). We made our implementation publicly available [40]. Our implementation contributes by providing the first tool that extends the features from the Requirements Toolbox™ to check completeness and consistency of Requirements Tables containing flexible and stateful requirements.

(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 suffi-

ciently 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. Our evaluation contributes by providing the first benchmark that can be used to compare tools that can check the completeness and consistency of Requirements Tables containing flexible and stateful requirements, by assessing our solution and comparing our encodings.

We also assessed the usefulness of THEANO in developing a Requirements Table for a practical example from the automotive domain. We iteratively constructed a Requirements Table and used THEANO to check for the completeness and consistency of its requirements. Across the 14 versions of the Requirements Table we developed, THEANO could detect inconsistencies and confirm the consistency of respectively $\approx 14\%$ (2 out of 14) and $\approx 86\%$ (12 out of 14) versions of the Requirements Table. THEANO could find incompleteness and confirm the completeness of respectively $\approx 36\%$ (5 out of 14) and $\approx 64\%$ (9 out of 14) versions of the Requirements Table. In total, THEANO reported a problem (inconsistency or incompleteness) for 50% (7 out of 14) of the versions of the Requirements Table.

(vi) An extensive *discussion* on the practical implications of our results and threats to validity (Section 8). Our discussion contributes by providing thorough reflections on our results.

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 [41]. The model typically 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 (**`prev`** (`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 $\texttt{dom}(u)$ and $\texttt{dom}(y)$ to indicate the domain of an input $u \in \mathcal{U}$ and output $y \in \mathcal{Y}$ variable.

We define the notion of a trace. Figure 2 presents two examples of traces. A trace associates a value for every input $u \in \mathcal{U}$ or output $y \in \mathcal{Y}$ variable and position $\texttt{i} \in \mathbb{N}^0$.

Definition 1 (Trace). A *trace* π is a tuple $\langle \iota_{\mathcal{U}}, \iota_{\mathcal{Y}}, \iota_{\tau} \rangle$, where the input $\iota_{\mathcal{U}} : \mathbb{N}^0 \times \mathcal{U} \to \mathbb{D}$ and output $\iota_{\mathcal{Y}} : \mathbb{N}^0 \times \mathcal{Y} \to \mathbb{D}$

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

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.

Fig. 2: Two examples of a trace for the example from Figure 1.

interpretations respectively associate each input $u \in \mathcal{U}$ and output $y \in \mathcal{Y}$ variable and position $\mathtt{i} \in \mathbb{N}^0$ with a value $\iota_{\mathcal{U}}(\mathtt{i}, u)$ and $\iota_{\mathcal{Y}}(\mathtt{i}, y)$ from their domain $\mathtt{dom}(u)$ and $\mathtt{dom}(y)$. The timestamp interpretation $\iota_\tau : \mathbb{N}^0 \to \mathbb{R}$ associates each position $\mathtt{i}$ to a timestamp value. □

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, $\iota_\tau(3) = 0.6$s and $\iota_\tau(5) = 1.0$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, $\iota_{\mathcal{U}}(3, \mathtt{th}) = 5$ and $\iota_{\mathcal{Y}}(5, \mathtt{sp}) = 140.4$. 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 $\iota(\mathtt{i}, v)$ to indicate its interpretation in position $\mathtt{i}$, that is $\iota(\mathtt{i}, v) = \iota_{\mathcal{U}}(\mathtt{i}, v)$ when v is an input variable, and $\iota(\mathtt{i}, v) = \iota_{\mathcal{Y}}(\mathtt{i}, v)$ when v is an output variable. For example, for the trace in Figure 2a, the interpretation of the variable th at position 3 is $\iota(3, \mathtt{th}) = \iota_{\mathcal{U}}(3, \mathtt{th}) = 5$.

We consider two trace types: fixed and variable sample steps traces. A fixed sample step trace records the signal values regularly after a specific sample step T_s. For example, Figure 2a has a fixed sample step: The trace records are recorded every 0.2 time instants, as exemplified by the values assumed by the variable τ that collects the timestamp at which the signal values are recorded. Unlike fixed sample step traces, in variable sample step traces the sample step is variable. For example, Figure 2b has a variable sample time as exemplified by the values assumed by the variable τ.

Definition 2 (Fixed and Variable Step Traces). A trace π has a *fixed sampling time* $\mathrm{T_s}$ if for every index $\mathtt{i} \in \mathbb{N}^0, \iota_\tau(\mathtt{i}+1) - \iota_\tau(\mathtt{i}) = \mathrm{T_s}$. 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. □

The trace from Figure 2a has a fixed sampling time since for all index $\mathtt{i} \in \mathbb{N}^0, \iota_\tau(\mathtt{i}+1) - \iota_\tau(\mathtt{i}) = 0.2$. 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.

$$
\begin{array}{lll}
\mathtt{t} & ::= & c \mid v \mid \mathbf{prev}(v) \mid \mathtt{t}_1 \odot \mathtt{t}_2 \\
\mathtt{le} & ::= & \mathtt{t}_1 \oplus \mathtt{t}_2 \mid \neg \mathtt{le} \mid \mathbf{dur}(\mathtt{le}) \geq c \mid \mathtt{le}_1 \oslash \mathtt{le}_2 \\
\mathtt{pre} & ::= & \mathtt{le} \\
\mathtt{post} & ::= & \mathtt{le} \\
\mathtt{req} & ::= & \mathtt{pre} \Rightarrow \mathtt{post} \\
\mathtt{rt} & ::= & \mathtt{req} \mid \mathtt{req}, \mathtt{rt}
\end{array}
$$

$c \in \mathbb{R}, v \in \mathcal{V}, \odot \in \{+, -, *, /\}, \oplus \in \{>, <, \leq, \geq, =, \neq\}, \oslash \in \{\wedge, \vee, \Rightarrow\}$.

Fig. 3: Requirements Tables: Syntax.

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 $\mathtt{dom}(v)$ to indicate the domain of the values for variable v.

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 $\mathbf{prev}(u)$ indicating the previous value of the input variable u, or a combination $\mathtt{t}_1 \odot \mathtt{t}_2$ of two terms $\mathtt{t}_1$, $\mathtt{t}_2$ with an arithmetic operator $\odot \in \{+, -, *, /\}$.

A logical expression le is either the combination $\mathtt{t}_1 \oplus \mathtt{t}_2$ of two terms $\mathtt{t}_1$ and $\mathtt{t}_2$ with a relational operator $\oplus \in \{>, <, \leq, \geq, =, \neq\}$, the negation $\neg \mathtt{le}$ of a logical expression le, the duration term $\mathbf{dur}(\mathtt{le}) \geq c$ indicating that a logical expression le holds for at least c seconds, or the combination $\mathtt{le}_1 \oslash \mathtt{le}_2$ 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] ⇒

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.

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

3.2 Semantics

After introducing the syntax of Requirements Tables we have to define their semantics, i.e., what the different constructs that define the Requirements Table mean. We define a trace-based semantics of Requirements Tables, that is we define when a trace satisfies or violates a Requirement Table. 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).

Informally, a position (i) of a trace (π) satisfies (the requirements of) a Requirements Table (rt) if the values assigned to the signal variables in that position satisfy the specification from the Requirements Tables.

Definition 3 (Semantics). Let rt, $\pi = \langle \iota_{\mathcal{U}}, \iota_{\mathcal{Y}}, \iota_{\tau} \rangle$, and i ≥ 0 respectively be a Requirements Table, a trace, and a position. The satisfaction $\texttt{i}, \pi \models \texttt{rt}$ 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 **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 $\iota(\texttt{i}, v)$. For example, for Figure 2a, the interpretation of the variable th at position 3 is $\iota(3, \texttt{th}) = 5$. The interpretation of the term **prev**(v) at position i is the value $\iota(\texttt{i} - 1, v)$ assumed by the variable v in the previous position $\texttt{i} - 1$ 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 **prev**(th) of the variable th at position 3 is $\iota(2, \texttt{th}) = 10$. From Simulink R2023a, Requirements Tables require initial value to be specified for inputs and outputs when accessing their previous values at the beginning of simulation.[3] For simplicity, in this work, we assume that every variable is assigned to an initial value. The interpretation of $\texttt{t}_1 \odot \texttt{t}_2$ is obtained by applying the arithmetic operator $\odot$ to the interpretations of $\texttt{t}_1$ and $\texttt{t}_2$. For example, for Figure 2a, the interpretation of $\texttt{th} + \texttt{sp}$ 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.

The semantics of an expression of the form $\texttt{t}_1 \oplus \texttt{t}_2$ assesses whether the interpretation of the terms $\texttt{t}_1$ and $\texttt{t}_2$ at position i satisfy the relation specified by the relational operator $\oplus$. For example, for Figure 2a, the expression $\texttt{sp} < \texttt{th}$ does not hold at position 2 since the value 147 is not lower than 10. The semantics of an expression of the form $\neg$le specifies that the expression holds at position i if le does not hold at that position. For example, for Figure 2a, the expression $\neg(\texttt{sp} < \texttt{th})$ holds at position 2 since $\texttt{sp} < \texttt{th}$ does not hold. The semantics of the logical expression $\texttt{le}_1 \oslash \texttt{le}_2$ specifies that the expression holds at position i if the evaluation of the logical expressions $\texttt{le}_1$ and $\texttt{le}_2$ at position i satisfy the relation specified by the logical operator $\oslash$. For example, for Figure 2a, the logical expression $(\texttt{th} < 20) \wedge (\texttt{sp} < 150)$ holds at position 2 since both $\texttt{th} < 20$ and $\texttt{sp} < 150$ hold in that position. The semantics of $\texttt{pre} \Rightarrow \texttt{post}$ (for the case in which the duration column is not used) specifies that the expression holds at position i if the postcondition post holds when 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 $(\texttt{th} > 0)$ is satisfied, but the postcondition $(\texttt{sp} = \textbf{prev}(\texttt{sp}) + \texttt{th})$ 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 **dur**(le) $\geq c$ and requirements expressed using a duration column are defined: Fixed step and variable step semantics. These semantics respectively support fixed and variable step traces.

- *Fixed step semantics*. For the operator **dur**(le) $\geq c$, we assume that every constant c used within the duration operator is greater than the sample time $\mathrm{T_s}$ ($c \geq \mathrm{T_s}$). Additionally, the value $c_r = \lceil \frac{c}{\mathrm{T_s}} \rceil \cdot \mathrm{T_s}$ 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 $\mathrm{T_s}$. For example, consider the expression operator

3. https://www.mathworks.com/help/slrequirements/ug/modify-requirements-table-block-data.html.

ID	Pre.	Post.
1	$\mathbf{dur}(\mathtt{th} > 3) \geq 0.4$	$\mathtt{sp} = 1$
2	$\mathbf{dur}(\mathtt{th} > 3) < 0.4$	$\mathtt{sp} = 2$

(a) Example 1.

ID	Pre.	Post.
1	$\mathbf{dur}(\mathtt{th} > 3) > 0.4$	$\mathtt{sp} = 1$
2	$\mathbf{dur}(\mathtt{th} > 3) < 0.4$	$\mathtt{sp} = 2$

(b) Example 2.

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

(c) Example 3.

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

(d) Example 4.

Fig. 4: Examples of Requirements Tables.

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

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

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

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

(b) Fixed Step Semantics.

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

(c) Variable Step Semantics.

Fig. 5: Requirements Tables: Semantics.

$\mathbf{dur}(\mathtt{th} < 20) \geq 0.3$, the trace from Figure 2a, and the position $\mathtt{i} = 3$ with timestamp 0.6. For $\mathbf{dur}(\mathtt{th} < 20) \geq 0.3$ to hold in position 3, the expression $\mathtt{th} < 20$ 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 $\mathtt{th} < 20$ 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., $c_r = \lceil \frac{c}{\mathrm{T_s}} \rceil \cdot \mathrm{T_s} = \lceil \frac{0.3}{0.2} \rceil \cdot 0.2 = 2 \cdot 0.2 = 0.4$. 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 $\mathbf{dur}(\mathtt{th} < 20) \geq 0.3$ at position 3. Formally, the fixed step semantics (Figure 5b) specifies that the current timestamp $\iota_\tau(\mathtt{i})$ 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 $\iota_\tau(\mathtt{i})$ and $\iota_\tau(\mathtt{i}-k)$ is lower than c, the expression $\mathtt{le}$ is satisfied (i.e., $k, \pi \models \mathtt{le}$). For example, for Figure 5b, the expression $\mathbf{dur}(\mathtt{th} > 3) \geq 0.4$ holds at position 3 since for the positions 1, 2 and 3 (i.e., the positions for which $\iota_\tau(\mathtt{i}) - \iota_\tau(\mathtt{i}-k) \leq 0.4$ hold) the expression $\mathtt{th} > 3$ holds.

The fixed step semantics of a requirement expressed using a duration column uses the value $d_r = \lceil \frac{\mathtt{d}}{\mathrm{T_s}} \rceil \cdot \mathtt{d}$ and is defined if the duration $\mathtt{d}$ is greater than the sample time $\mathrm{T_s}$ ($\mathtt{d} \geq \mathrm{T_s}$). The fixed step semantic of a requirement expressed using a duration column requires the postcondition ($\mathtt{post}$) to hold in position $\mathtt{i}$ if (a) the current timestamp $\iota_\tau(\mathtt{i})$ 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 $\iota_\tau(\mathtt{i})$ and $\iota_\tau(\mathtt{i}-k)$ is lower than c, the precondition is satisfied (i.e., $k, \pi \models \mathtt{pre}$).

- *Variable step semantics*. The variable step semantics (Figure 5c) specifies that the expression holds at position $\mathtt{i}$ if there exists a position k lower than $\mathtt{i}$ such that the difference $\iota_\tau(\mathtt{i}) - \iota_\tau(k)$ of the timestamps associated with positions $\mathtt{i}$ and k is greater than or equal to the value c and that the logical expression $\mathtt{le}$ holds from k to $\mathtt{i}$. For example, for Figure 2b, the expression $\mathbf{dur}(\mathtt{th} > 3) \geq 1$ holds at position 3 since at positions 1, 2 and 3 the expression $\mathtt{th} > 3$ 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 semantics of a requirement expressed using a duration column requires the postcondition ($\mathtt{post}$) to hold in position $\mathtt{i}$ if there exists a position k lower than $\mathtt{i}$ such that the difference $\iota_\tau(\mathtt{i}) - \iota_\tau(k)$ of the timestamps associated with positions $\mathtt{i}$ and k is greater than or equal to the value c and that the precondition ($\mathtt{pre}$) holds from k to $\mathtt{i}$.

Informally, a trace (π) satisfies (the requirements of) a Requirements Table ($\mathtt{rt}$) if the values assigned to the signal variables satisfy the specification from the Requirements Tables for every trace position.

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

For example, the Requirements Table from Figure 4a is violated by the fixed step trace from Figure 2, since the precondition ($\mathbf{dur}(\texttt{th} > 3) \geq 0.4$) of the first requirement of the Requirements Table holds in position 3 of the trace, but its postcondition ($\texttt{sp} = 1$) 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.

Informally, a Requirements Table $\texttt{rt}$ is *complete* if for every output variable and position *at least* one precondition is "active", i.e., the behavior of the system is defined by at least one requirement.

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

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 $\mathbf{dur}(\texttt{th} > 3) \geq 0.4$ or $\mathbf{dur}(\texttt{th} > 3) < 0.4$ holds. The Requirements Table from Figure 4b is not complete since when $\mathbf{dur}(u > 3) = 0.4$ none of the preconditions are satisfied.

We show that checking the completeness of a Requirements Table is an undecidable problem (in general), i.e., a Turing machine that can always solve the problem in a finite amount of time does not exist.

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

Proof 1 (Proof Sketch). We consider the tenth Hilbert problem [42], 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 [43]) 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.

Informally, a Requirements Table $\texttt{rt}$ is *consistent* if for every input interpretation $\iota_{\mathcal{U}}$ it is possible to define (at least) an output $\iota_{\mathcal{Y}}$ interpretation that satisfies the requirements. This ensures that engineers can develop a software controller that produces one of the output interpretations that satisfy the requirements for that specific input.

Definition 6 (Consistency). A Requirements Table $\texttt{rt}$ 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., $\pi \models \texttt{rt}$. □

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

We show that checking the consistency of a Requirements Table is an undecidable problem (in general), i.e., a Turing machine that can always solve the problem in a finite amount of time does not exist.

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

Proof 2 (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 $\mathit{true} \Rightarrow A = 0$. 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.

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 attempts to check 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. Given that the task is computationally hard (and, as noted in previous section, generally undecidable), the success of our approach largely depends on the success of SatCheck component.

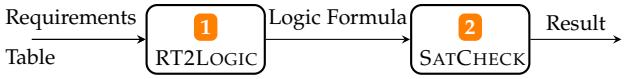


Fig. 6: THEANO: Approach.

We present two approaches to converting Requirements Tables into logic formulae. Both approaches follow the 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. The unbounded encoding checks the consistency and completeness of Requirements Tables by considering traces with an arbitrary number of trace records. Therefore, they can confirm the consistency and completeness of Requirements Tables and their inconsistency and incompleteness. However, we speculate that this encoding can have significant drawbacks in terms of efficiency. Our evaluation (Section 7) will assess if our speculation on the drawbacks of the unbounded encoding is confirmed. We also propose a bounded encoding to address the (potential) performance issues of the unbounded encoding. Unlike unbounded encoding, bounded encoding limits the number of trace records considered during the consistency and completeness checks. Therefore, this encoding can detect inconsistencies and incompleteness in the Requirements Tables but can not confirm their consistency and completeness. Our evaluation (Section 7) will assess if our speculation is correct and 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 Enconding of the Traces

We need to select appropriate structures to represent traces. We identified two possible alternatives: Arrays (i.e., data structures consisting of a collection of values associated with their positions within the array) or uninterpreted functions (i.e., functions with a name and signature but no definition). Z3 offers decision procedures for both alternatives. In the following, we introduce two encodings for these two data structures. Their effectiveness and efficiency are assessed in Section 7.

Encoding Based on Arrays (`Ar`). Our encoding creates one array $\overline{u}$ and $\overline{y}$ for each input $u \in \mathcal{U}$ and output $y \in \mathcal{Y}$ variable. We use the symbol $\overline{v}$ 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 $\overline{\mathcal{U}}$ and $\overline{\mathcal{Y}}$ 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 $\overline{\tau}$ that contains the timestamps corresponding to each trace position. The trace array $\overline{\pi}$ is the tuple $\langle \overline{\mathcal{U}}, \overline{\mathcal{Y}}, \overline{\tau} \rangle$ 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 [44] to represent the timestamps. An uninterpreted function [44] 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 function[4] $\overline{\tau} : \mathbb{N}^0 \rightarrow \mathbb{R}$ that contains the timestamps corresponding to each trace position. The trace function $\overline{\pi}$ is the tuple $\langle \overline{\mathcal{U}}, \overline{\mathcal{Y}}, \overline{\tau} \rangle$ 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 $\overline{v}[i]$ and $\overline{\tau}[i]$ to seamlessly indicate the values associated by the uninterpreted functions $\overline{v}$ and $\overline{\tau}$ to the value $\mathtt{i} \in \mathbb{N}^0$, or the value of the arrays $\overline{v}$ and $\overline{\tau}$ in position $\mathtt{i} \in \mathbb{N}^0$. 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 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.

TABLE 1: Encodings τ_m of the traces.

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

* $T_s > 0$

5.2 The Unbounded Encoding of Requirements Tables

The unbounded encoding of Requirements Tables is defined by the function $\mathfrak{h}$ 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 $\overline{v}[i]$. For **prev**(v) the translation returns $u[\mathtt{i}-1]$ if i is greater than 0, $u[\mathtt{i}]$ otherwise; for $\mathtt{t}_1 \odot \mathtt{t}_2$ it returns $\mathfrak{h}_i(\mathtt{t}_1) \odot \mathfrak{h}_i(\mathtt{t}_2)$; for $\mathtt{t}_1 \oplus \mathtt{t}_2$ it returns $\mathfrak{h}_i(\mathtt{t}_1) \oplus \mathfrak{h}_i(\mathtt{t}_2)$; for $\neg\mathtt{le}$ it returns $\neg\mathfrak{h}_i(\mathtt{le})$; for $\mathtt{le}_1 \oslash \mathtt{le}_2$ it returns $\mathfrak{h}_i(\mathtt{le}_1) \oslash \mathfrak{h}_i(\mathtt{le}_2)$; for `req` it returns $\mathfrak{h}_i(\mathtt{pre}) \Rightarrow \mathfrak{h}_i(\mathtt{post})$, and for $\mathfrak{h}_i(\mathtt{rt}, \mathtt{req})$ it returns $\mathfrak{h}_i(\mathtt{rt}) \wedge \mathfrak{h}_i(\mathtt{req})$.

The encoding of the operator **dur**$(\mathtt{le}) \geq c$ 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**$(\mathtt{le}) \geq c$, it returns a

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

$\mathfrak{h}_{i,\overline{\pi}}(c)$	$\equiv$	c
$\mathfrak{h}_{i,\overline{\pi}}(v)$	$\equiv$	$\overline{v}[i]$
$\mathfrak{h}_{i,\overline{\pi}}(\textbf{prev}(u))$	$\equiv$	$((\mathtt{i} > 0) \Rightarrow (u[\mathtt{i}-1])) \wedge ((\mathtt{i} = 0) \Rightarrow u[\mathtt{i}] = 0)$
$\mathfrak{h}_{i,\overline{\pi}}(\mathtt{t}_1 \odot \mathtt{t}_2)$	$\equiv$	$\mathfrak{h}_{i,\overline{\pi}}(\mathtt{t}_1) \odot \mathfrak{h}_{i,\overline{\pi}}(\mathtt{t}_2)$
$\mathfrak{h}_{i,\overline{\pi}}(\mathtt{t}_1 \oplus \mathtt{t}_2)$	$\equiv$	$\mathfrak{h}_{i,\overline{\pi}}(\mathtt{t}_1) \oplus \mathfrak{h}_{i,\overline{\pi}}(\mathtt{t}_2)$
$\mathfrak{h}_{i,\overline{\pi}}(\neg \mathtt{le})$	$\equiv$	$\neg \mathfrak{h}_{i,\overline{\pi}}(\mathtt{le})$
$\mathfrak{h}_{i,\overline{\pi}}(\mathtt{le}_1 \oslash \mathtt{le}_2)$	$\equiv$	$\mathfrak{h}_{i,\overline{\pi}}(\mathtt{le}_1) \oslash \mathfrak{h}_{i,\overline{\pi}}(\mathtt{le}_2)$
$\mathfrak{h}_{i,\overline{\pi}}(\mathtt{req})$	$\equiv$	$\mathfrak{h}_{i,\overline{\pi}}(\mathtt{pre}) \Rightarrow \mathfrak{h}_{i,\overline{\pi}}(\mathtt{post})$
$\mathfrak{h}_{i,\overline{\pi}}(\mathtt{rt}, \mathtt{req})$	$\equiv$	$\mathfrak{h}_{i,\overline{\pi}}(\mathtt{rt}) \wedge \mathfrak{h}_{i,\overline{\pi}}(\mathtt{req})$

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

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

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

(b) Fixed Step Semantics (Fs).

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

(c) Variable Step Semantics (Vs).

Fig. 7: Unbounded Encoding of Requirements Tables.

formula requiring (a) the timestamp $\overline{\tau}[i]$ at position $\mathtt{i}$ to be greater than c, (b) the value of the c to be greater than the sample time $\mathrm{T_s}$, (c) the formula $\mathfrak{h}_k(\mathtt{le})$ to hold for every position k within $[\mathtt{i} - \lceil \frac{c}{\mathrm{T_s}} \rceil, \mathtt{i}]$. Note that the definition of the translation for the operator $\mathfrak{h}_k(\mathtt{le})$ 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 $\textbf{dur}(\mathtt{le}) \geq c$, it returns a formula requiring the existence of an index j lower than i such that the difference $\overline{\tau}[i] - \overline{\tau}[j]$ 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 $j \leq k \leq \mathtt{i}$, the formula $\mathfrak{h}_k(\mathtt{le})$ 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 $\mathtt{rt}$ be a Requirements Table, the logic formula $\mathfrak{h}_{cp}(\mathtt{rt})$ to verify the completeness of the Requirements Table $\mathtt{rt}$ is

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

The formula specifies the existence of a trace ($\overline{\pi}$) and a position ($\mathtt{i}$) where for at least one output variable (y), all the conditions ($\mathtt{pre}$) of the corresponding preconditions ($Pre(\mathtt{rt}, y)$) do not hold ($\neg \mathfrak{h}_{i,\overline{\pi}}(\mathtt{pre})$). 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.

Our procedure to check completeness requires checking the satisfiability of the formula $\mathfrak{h}_{cp}(\mathtt{rt})$. If the formula $\mathfrak{h}_{cp}(\mathtt{rt})$ is satisfiable, the Requirements Table $\mathtt{rt}$ is incomplete, otherwise it is complete. Therefore, to prove the correctness of our procedure we have to prove the following theorem.

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

Proof 3 (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 an array (or uninterpreted function) and an integer that satisfies formula $\mathfrak{h}_{cp}(\mathtt{rt})$ corresponds to verifying the presence of an output variable $y \in \mathcal{Y}$, a trace $\pi \in \Pi$, and a position $\mathtt{i}$ that violates the conditions of Definition 5.

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 $\forall k, ((\mathtt{i} - 2 < k \leq \mathtt{i}) \Rightarrow \mathtt{th}[\mathtt{i}] > 3)$, but $\mathtt{i} - 2 = k \wedge \neg \mathtt{th}[\mathtt{i}] > 3$ 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 $k = \mathtt{i} - 2$ and $\neg \mathtt{th}[\mathtt{i}] > 3$ make the implication false since the left side of the implication is true and the right side of the implication is false, for the one reported in Line 3 $k = \mathtt{i} - 2$ and $\neg \mathtt{th}[\mathtt{i}] > 3$ make the implication false since the left side of the conjunction is false.

```
1 ∃π̄, ∃i, (∀j ≥ 0, τ̄[j] < τ̄[j + 1]∧
2 ¬(τ̄[i] ≥ 0.4 ∧ 0.4 ≥ 0.2 ∧ ∀k, ((i − 2 ≤ k ≤ i) ⇒ th[i] > 3))∧
3 ¬(τ̄[i] ≥ 0.4 ∧ 0.4 ≥ 0.2 ∧ ∃k, ((i − 2 < k ≤ i) ∧ ¬th[i] > 3)))
```

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

```
1 ∃th, ∃τ̄, (∀j ≥ 0, τ̄[j] < τ̄[j + 1]∧
2 ∀sp, (∃i, ¬(
3         (τ̄[i] ≥ 0.4 ∧ 0.4 ≥ 0.2∧
4             ∀k, (((i − 2 ≤ k ≤ i) ⇒ th[i] > 3) ⇒ sp[i] = 1))∨
5         (τ̄[i] ≥ 0.5 ∧ 0.5 ≥ 0.2∧
6 ∀k, (¬(((τ̄[i] − τ̄[i − k] ≤ 0.5) ⇒ th[i] > 3) ⇒ sp[i] = 2))))))
```

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

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

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

$$\exists\overline{\mathcal{U}}, \exists\overline{\tau}, (\tau_{mon} \wedge \forall\overline{\mathcal{Y}}, (\exists \texttt{i}, (\neg\mathfrak{h}_{\texttt{i},\overline{\pi}}(\texttt{rt}))))$$

with $\overline{\pi} = \langle\overline{\mathcal{U}}, \overline{\mathcal{Y}}, \overline{\tau}\rangle$. 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 $(\exists \texttt{i}, (\neg\mathfrak{h}_{\texttt{i},\overline{\pi}}(\texttt{rt})))$.

For example, Figure 8b presents the encoding for Requirements Table from Figure 4c with sample step 0.2. Lines 3, 4 contain the encoding for the requirement 1 from Figure 4c. Lines 5, 6 contain the encoding for the requirement 2 from Figure 4c.

Our procedure to check consistency requires checking the satisfiability of the formula $\mathfrak{h}_{cs}(\texttt{rt})$. If the formula $\mathfrak{h}_{cs}(\texttt{rt})$ is satisfiable, the Requirements Table rt is inconsistent, otherwise it is consistent. Therefore, to prove the correctness of our procedure we have to prove the following theorem.

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

Proof 4 (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 $\mathfrak{h}_{cs}(\texttt{rt})$ corresponds to verifying the presence of an input interpretation $\iota_{\mathcal{U}}$ that satisfies the conditions of Definition 6.

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

Our encoding generates formulae (e.g., the one from Figure 8a) within the AUFNIRA logic [45], i.e., closed formulas (i.e., logical formulae that do not contain variables that are not quantified) with free function and predicate symbols over a theory of arrays (used to represent signals) of integer index (used to access signal positions) and real value (representing signal values). The problem of deciding if an AUFNIRA formula is satisfiable is undecidable, i.e., no algorithm 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 $\textbf{dur}(\texttt{le}) \geq c$ 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 $\textbf{dur}(\texttt{le}) \geq c$ 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 $\textbf{dur}(\texttt{le}) \geq c$, the translation from Figure 9a forces the formula le to hold for every position n within $[\texttt{i} - \lceil\frac{c}{\mathrm{T_s}}\rceil, \texttt{i}]$ by creating a conjunction of formulae $\mathfrak{h}_{n\leftarrow k,\overline{\pi}}(\texttt{le})$ each representing the value of le 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 $\textbf{dur}(\texttt{le}) \geq c$, the disjunction operator from Figure 9b forces the existence of a position j with a timestamp $\overline{\tau}[j]$ ensuring that $\overline{\tau}[i] - \overline{\tau}[j] \geq c$, while the conjunction operator forces the formula le to hold in every position n in $[j, i]$ by creating a conjunction of formulae

$\mathfrak{h}_{n\leftarrow k,\overline{\pi}}(\texttt{le})$ each representing the value of `le` 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 $\mathfrak{h}_{bcp}(\texttt{rt})$ to verify the incompleteness of a Requirements Table `rt` is

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

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., $0.4 \geq 0.4$).

Our procedure to check incompleteness requires checking the satisfiability of the formula $\mathfrak{h}_{bcp}(\texttt{rt})$. If the formula $\mathfrak{h}_{bcp}(\texttt{rt})$ is satisfiable, the Requirements Table `rt` is incomplete. Therefore, to prove the correctness of our procedure we have to prove the following theorem.

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

Proof 5 (Proof Sketch). Since $\mathfrak{h}_{bcp}(\texttt{rt})$ is satisfiable for bound b, the formula $\mathfrak{h}_{cp}(\texttt{rt})$ 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 $\mathfrak{h}_{cp}(\texttt{rt})$ is satisfiable the Requirements Table `rt` is incomplete. Therefore, if the formula $\mathfrak{h}_{bcp}(\texttt{rt})$ is satisfiable, then the Requirements Table `rt` is incomplete.

The bounded encoding shows the incompleteness of the Requirements Table: by setting $\overline{\tau}[0] = 0$, $\overline{\tau}[1] = 0.4$, $\overline{\tau}[2] = 0.8$, $\overline{\texttt{th}}[0] > 3$, $\overline{\texttt{th}}[1] > 3$, and $\overline{\texttt{th}}[2] \leq 3$, the formula from Figure 10a is satisfiable. Since the formula is satisfiable, the Requirements Table is incomplete.

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

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

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 $b = 2$.

Our procedure to check inconsistency requires checking the satisfiability of the formula $\mathfrak{h}_{bcs}(\texttt{rt})$. If the formula $\mathfrak{h}_{bcs}(\texttt{rt})$ is satisfiable, the Requirements Table `rt` is inconsistent. Therefore, to prove the correctness of our procedure we have to prove the following theorem.

Theorem 6. Let b be a bound and `rt` be a Requirements Table, if the formula $\mathfrak{h}_{bcs}(\texttt{rt})$ is satisfiable, then the Requirements Table `rt` is inconsistent.

Proof 6 (Proof Sketch). Since $\mathfrak{h}_{bcs}(\texttt{rt})$ is satisfiable for bound b, the formula $\mathfrak{h}_{cs}(\texttt{rt})$ 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 $\mathfrak{h}_{cs}(\texttt{rt})$ is satisfiable the Requirements Table `rt` is inconsistent. Therefore, if the formula $\mathfrak{h}_{bcs}(\texttt{rt})$ is satisfiable, then the Requirements Table `rt` is inconsistent.

For example, the formula from Figure 10b is satisfiable since the interpretation for the input (`th`) and the timestamp ($\overline{\tau}$) variables ensuring the following conditions $\overline{\tau}[1] \geq 0.5$, and $0 < \overline{\tau}[0] - \overline{\tau}[1] < 0.5$, and $\texttt{th}[1] > 3$ is such that for every interpretation of the output variable (`sp`) either $\texttt{sp}[0] >= 1$ and $\texttt{sp}[1] = 1$, which makes the first condition of the disjunction satisfied, or $\texttt{sp}[0] = 2$ and $\texttt{sp}[1] = 2$ 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 $\mathfrak{h}_{bcp}(\texttt{rt})$ and $\mathfrak{h}_{bcs}(\texttt{rt})$ 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 model traces in the research literature (e.g., [39]). 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

$$\mathfrak{h}_{i,\overline{\pi}}(\textbf{dur}(\texttt{le}) \geq c) \equiv \overline{\tau}[i] \geq c \wedge c \geq \mathrm{T_S} \bigwedge_{k \in [\mathtt{i} - \lceil \frac{c}{\mathrm{T_S}} \rceil, \mathtt{i}]} \mathfrak{h}_{n \leftarrow k, \overline{\pi}}(\texttt{le})$$
$$\mathfrak{h}_{i,\overline{\pi}}(\texttt{pre[d]} \Rightarrow \texttt{post}) \equiv \overline{\tau}[i] \geq \mathtt{d} \wedge \mathtt{d} \geq \mathrm{T_S} \wedge ((\bigwedge_{k \in [\mathtt{i} - \lceil \frac{\mathtt{d}}{\mathrm{T_S}} \rceil, \mathtt{i}]} \mathfrak{h}_{n \leftarrow k, \overline{\pi}}(\texttt{pre})) \Rightarrow \mathfrak{h}_{n \leftarrow i, \overline{\pi}}(\texttt{post}))$$

(a) Fixed Step Semantics.

$$\mathfrak{h}_{i,\overline{\pi}}(\textbf{dur}(\texttt{le}) \geq c) \equiv \bigvee_{j \in [0,\mathtt{i}]} (\overline{\tau}[i] - \overline{\tau}[j] \geq c \wedge \bigwedge_{k \in [j,\mathtt{i}]} \mathfrak{h}_{n \leftarrow k, \overline{\pi}}(\texttt{le}))$$
$$\mathfrak{h}_{i,\overline{\pi}}(\texttt{pre[d]} \Rightarrow \texttt{post}) \equiv \bigvee_{j \in [0,\mathtt{i}]} (\overline{\tau}[i] - \overline{\tau}[j] \geq \mathtt{d} \wedge (\bigwedge_{k \in [j,\mathtt{i}]} \mathfrak{h}_{n \leftarrow k, \overline{\pi}}(\texttt{pre})) \Rightarrow \mathfrak{h}_{n \leftarrow i, \overline{\pi}}(\texttt{post}))$$

(b) Variable Step Semantics.

Fig. 9: Bounded Encoding.

$$\begin{array}{l}
1\ \exists \overline{\pi}, (\overline{\tau}[0] = 0 \wedge \overline{\tau}[1] = 0.4 \wedge \overline{\tau}[2] = 0.8 \wedge \\
2\ ((\neg(\overline{\tau}[0] \geq 0.4 \wedge \wedge \mathtt{th}[0] > 3 \wedge \mathtt{th}[1] > 3) \wedge \neg(\overline{\tau}[0] \geq 0.4 \wedge (\neg \mathtt{th}[0] > 3)))) \\
3\ \vee \\
4\ (\neg(\overline{\tau}[1] \geq 0.4 \wedge \wedge \mathtt{th}[0] > 3 \wedge \mathtt{th}[1] > 3 \wedge \mathtt{th}[2] > 3) \wedge \neg(\overline{\tau}[1] \geq 0.4 \wedge (\neg \mathtt{th}[0] > 3 \vee \neg \mathtt{th}[1] > 3))))
\end{array}$$

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

$$\begin{array}{l}
1\ \exists \overline{\pi}, (\overline{\tau}[0] = 0 \wedge \overline{\tau}[1] = 0.4 \wedge \overline{\tau}[2] = 0.8 \wedge \\
2\ ((\neg((\overline{\tau}[0] \geq 0.4 \wedge \mathtt{th}[0] > 3) \Rightarrow \mathtt{sp}[0] = 1) \vee \neg((\overline{\tau}[0] \geq 0.5 \wedge \mathtt{th}[0] > 3) \Rightarrow \mathtt{sp}[0] = 2)) \\
3\ \vee \\
4\ ((\neg((\overline{\tau}[1] \geq 0.4 \wedge \mathtt{th}[0] > 3 \wedge \mathtt{th}[1] > 3) \Rightarrow (\mathtt{sp}[0] = 1 \wedge \mathtt{sp}[1] = 1))) \vee (\neg((\overline{\tau}[1] \geq 0.5 \wedge \mathtt{th}[0] > 3 \wedge \mathtt{th}[1] > 3) \Rightarrow (\mathtt{sp}[0] = 2 \wedge \mathtt{sp}[1] = 2)))))
\end{array}$$

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

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

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

Fig. 11: Implementation of THEANO.

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

```
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
```

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

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 [31], a parser generator able to read grammars and generate a recognizer for the language. We selected antlr since it is a widely used tool [46] and a grammar for parsing MATLAB expressions was publicly available [47]. TEXT2Z3 implements the translation from Section 5 and generates a Python file that encodes the formulae in Z3 [33]. Z3 was chosen since it is an industry-strength awarded tool to check the satisfiability of logic-based formulae [48], [49]. Specifically, we used Z3Py [32], 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., [39]). Figure 13 presents an example of the python (.py) file produced by the TEXT2Z3 component for the file from Figure 12 to check completeness for the BeUfVs

```
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)
```

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

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 [33]. 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 [50] which is outdated, i.e., the last update of mparser was on 1 Sep 2011. In addition, this tool relies on libantlr3c [51], 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 (complete/incomplete 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.

We rely on the benchmark presented in Section 7.1 to answer these research questions (RQ1 and RQ2).

RQ3 *How useful is* THEANO *in supporting the development of a realistic Requirements Table from a practical example?* (Section 7.4)

This research question assesses the usefulness of THEANO in analyzing the consistency and completeness while developing a Requirements Table from a practical example.

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 [29]. 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. The baseline Requirements Tables were defined by considering the experience of the authors of this paper, which includes industrial experts with significant experience in the definition and analysis of Requirements Tables. 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 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

ID	Pre.	Post.
1	**isStartup**	y == 0;
2	¬ **isStartup** ∧ u > 0	y == **prev** (y) + 1;
3	¬ **isStartup** ∧ **prev** (y) == thr	y < thr;

(a) Baseline Table 1.

ID	Pre.	Post.
1	**isStartup**	y == 0;
2	¬ **isStartup**	y == **prev** (y) + 1;
3	¬ **isStartup** ∧ **prev** (y) == thr	y < thr;

(b) Baseline Table 2.

ID	Pre.	Post.
1	**isStartup**	y == 0;
2	¬ **isStartup** ∧ u == **prev** (u) + 1	y == **prev** (y) + 1;
3	¬ **isStartup** ∧ u != **prev** (u) + 1 ∧ y < thr	y < thr;

(c) Baseline Table 3.

ID	Pre.	Post.
1	**isStartup**	y == 0;
2	¬ **isStartup** ∧ u == **prev** (u) + 1	y == **prev** (y) + 1;
3	¬ **isStartup** ∧ u != **prev** (u) + 1	y < thr;

(d) Baseline Table 4.

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

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 **prev**$(y) = thr$ 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 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: Expected results for the completeness and consistency checks for the Requirements Tables generated from the baseline Tables from Figure 14.

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

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., [39]). 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 400 (40×10) 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 (400×4) 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 do not show significant differences in the effectiveness of the unbounded encodings based on arrays and uninterpreted functions and fixed and variable sample step semantics. They also 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

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.

	Completeness					Consistency				
Encoding	**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%	99.50%	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%

timeout.

The results from Table 2 do not show significant differences in the effectiveness of the bounded encodings based on arrays and uninterpreted functions and fixed and variable sample step semantics.

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 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.

Note that, when reducing the thresholds, we want to understand if THEANO could confirm the consistency of Requirements Tables for smaller threshold values. Our goal is not to analyze its scalability as the threshold value increases. Our results confirm that THEANO can verify the consistency of Requirements Tables for smaller threshold values. We do not have an explanation for the non-monotonic results of Table 5: This analysis is out of our scope. However, this is not the first time non-monotonic behaviors have been observed. For example, non-monotonic behaviors in the Z3 solver have also been observed in other works (e.g., when increasing the bound k used to encode Timed Automata [38]). These behaviors are likely caused by the non-deterministic behavior of the SMT technology and the strategy selected by the solver to solve the SMT problem which can differ across multiple runs.

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,

and 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. We analyzed the time required to compute those 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). Considering the limited time (less than two minutes) required by all the encodings to confirm the completeness of the Requirements Tables we did not report the time the different encodings require since it does not make any practical difference.

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 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.

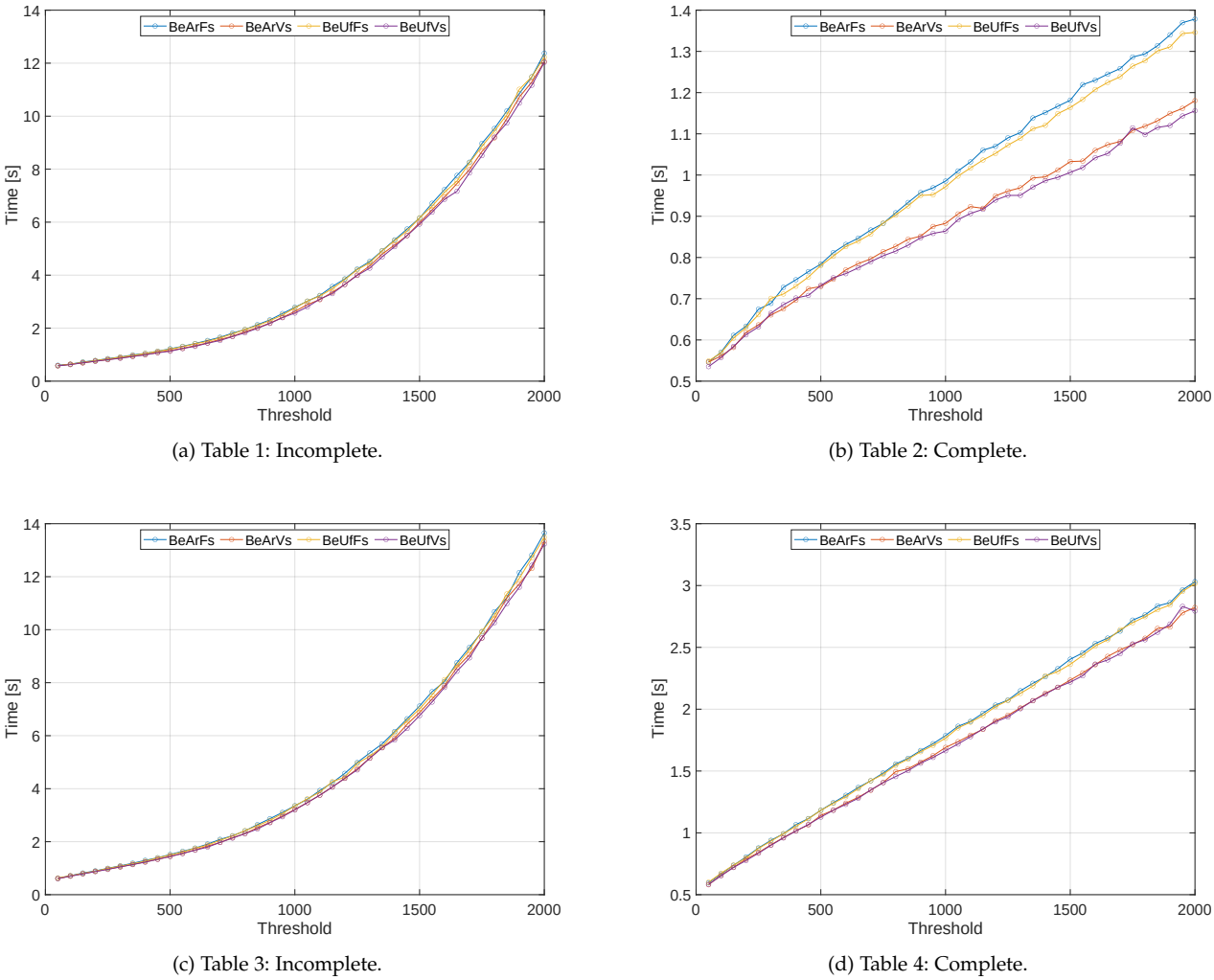
On average (across the different runs) when configured with the `BeUfFs`, `BeUfVs`, `BeArFs`, and `BeArVs` encodings, THEANO required 2.9 (*min*=0.5, *max*=16.3, *StdDev*=3.0), 2.8 (*min*=0.5, *max*=15.5, *StdDev*=2.9), 2.9 (*min*=0.5, *max*=16.8, *StdDev*=3.0), and 2.8 (*min*=0.5, *max*=16.0, *StdDev*=3.0) 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 ($\geq$0.71) 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 ($\geq$0.71) 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.

From our results, we did not notice any significant difference in the efficiency of the proposed technique related to the fixed and variable step semantics. Therefore, our procedure for checking completeness and consistency enables engineers to select (among the fixed and the variable step semantics) the more appropriate semantics for their problem since the completeness and consistency checks have similar efficiency.

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,



(a) Table 1: Incomplete.

(b) Table 2: Complete.

(c) Table 3: Incomplete.

(d) Table 4: Complete.

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

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).

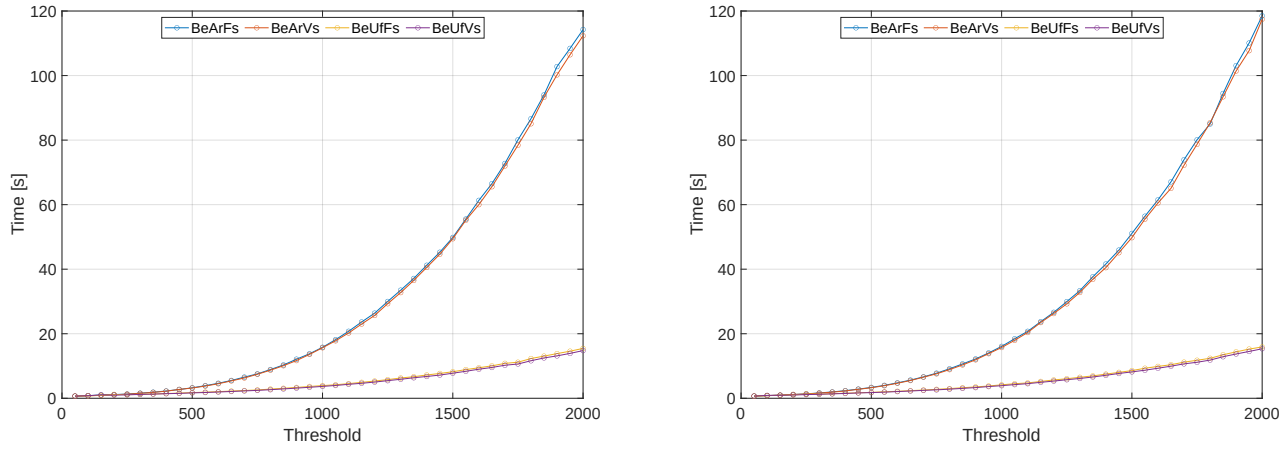
RQ2 - Efficiency

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.

7.4 Practical Application

We assessed the usefulness of THEANO in developing a Requirements Table for a practical example. We selected the Automatic Transmission Controller model (AT) [52] from the Applied Verification for Continuous and Hybrid Systems (ARCH): A yearly competition comparing state-of-the-art tools for testing and verifying hybrid systems. We considered the last report from the falsification category [53] containing the formalization of the AT requirements. We selected AT since it has the highest number of requirements (nine).[6] Therefore, AT is likely more interesting when its requirements are translated into a Requirements Table to assess the usefulness of the consistency and completeness checks. The inputs for the AT model are the throttle

6. We did not consider requirement AT6ab which is a simple conjunction of requirements AT6a, AT6b, and AT6c.



(a) Table 1: Inconsistent.

(b) Table 2: Inconsistent.

Fig. 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.

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

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

TABLE 7: Results of the consistency and completeness checks for the Requirements Tables of the AT example.

Reqirements Table Version	**v1**	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**	**v8**	**v9**	**v10**	**v11**	**v12**	**v13**	**v14**
Consistency Check Result	✓	✓*	✓*	✓	✓*	✓*	✓	✓	✗	✓*	✓*	✓*	✗	✓*
Completeness Check Result	✗	✗	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓

✓: THEANO confirmed the consistency/completeness of the Requirements Table.
✗: THEANO confirmed the inconsistency/incompleteness of the Requirements Table.
✓*: since THEANO timed out, the Requirements Table is likely to be consistent.

(`throttle`) and the brake (`brake`) signals. Its outputs are the gear (`g`), the rounds per minute (`rpm`) of the vehicle engine (ω), and its speed (`v`). We iteratively constructed a Requirements Table from this model and used THEANO to check for the completeness and consistency of its requirements. We selected the `BeUfFs` encoding since (a) the bounded encodings enable verifying the incompleteness of Requirements Tables in practical time, (b) encodings based on uninterpreted functions are more efficient than the ones based on arrays, and (c) the model is simulated by considering a fixed step size solver. We set a timeout of one hour for the consistency and completeness checks. We selected 33 as a value for the bound since it is the simulation time considered for the AT model. Even if we selected a bounded encoding, considering that the value for the bound matches the simulation time, we state that the Requirements Table is likely complete/consistent when the bounded encoding could not show its incompleteness/inconsistency.

In total, we developed 14 versions of the Requirements Table. Table 7 summarizes the results of the consistency and completeness of its requirements for the different versions of the Requirements Table we defined. Symbol ✓ indicates that THEANO confirmed the consistency/completeness of the Requirements Table. Symbol ✗ indicates that THEANO confirmed the inconsistency/incompleteness of the Requirements Table. The symbol ✓* associated with the consistency check result indicates that since THEANO timed out, the Requirements Table is likely to be consistent. The results are detailed in the following.

Version v1. We started by defining a Requirements Table with two input variables (`throttle` and `brake`) that can assume real values and three output variables (`rpm`, `w`, and `v`). The variables representing the rounds per minute and the vehicle speed can assume real values; the gear

assumes integer values. We added no requirement to the Requirements Tables. The Requirements Table is consistent since there are no contradictions among requirements. The Requirements Table is incomplete: No active precondition exists for all the output variables and positions.

Version v2. We added one requirement to the Requirements Table modeling the requirement AT1 from the ARCH report. AT1 requires the vehicle speed to be lower than 120 within the interval [0,20]. To model AT1, we used the variable (τ) representing the timestamp within the preconditions of the Requirements Table. Our Requirements Table has a precondition that specifies that the requirement should hold when the timestamp is within [0,20], and a postcondition that forces the speed (v) to be lower than 120. The consistency checker could not terminate within the timeout. Therefore, based on the results from RQ1, the Requirements Table is likely to be consistent. The Requirements Table is incomplete: the value assumed by the car speed is not specified outside the interval $[0, 20]$.

Version v3. We added a requirement specifying that the vehicle speed can assume any value when the timestamp is not within the interval $[0, 20]$. The Requirements Table is likely consistent since the consistency checker could not terminate within the timeout. The Requirements Table is incomplete: no active precondition exists for some position of the trace for the output variables `g` and `w`.

Version v4. To check the effectiveness of the change introduced in version v2, we temporarily removed the output variables `g` and `w` from the Requirements Table. By removing these variables, the consistency checker could confirm the consistency of the Requirements Table. The Requirements Table is complete.

Version v5. We reintroduced the `w` variable modeling the RPM. We added one requirement to the Requirements Table modeling requirement AT2 from the ARCH report. AT2 requires the RPM to be lower than 4750 within the interval [0,10]. The Requirements Table is likely consistent since the consistency checker could not terminate within the timeout. The Requirements Table is not complete.

Version v6. We added a requirement specifying that the RPM can assume any value when the timestamp is not within the interval [0, 10]. The Requirements Table is likely consistent since the consistency checker could not terminate within the timeout. The Requirements Table is complete.

Version v7. We reintroduced the `g` variable modeling the gear. We added one requirement to our Requirements Table modeling the requirement AT51 from the ARCH report. AT51 requires that within the time interval [0,30]s, if the vehicle changes the gear to 1 (from another gear), it shall remain in that gear for 2.5s. Modeling this requirement required using the operators **dur** and **prev**. Additionally, since a Requirements Table describes the relation between values assumed by input and output variables, we added an input variable to model the "monitored" gear (*gin*). The Requirements Table is consistent. The Requirements Table is incomplete since it does not specify the value of the output gear outside the time interval [0,30]s and within that interval when the vehicle does not change the gear to 1.

Version v8. We added one requirement specifying that the gear can assume any value in these cases. The Requirements Table is consistent and complete.

Version v9. We added the requirements AT52, AT53, and AT54. The requirements AT52, AT53, and AT54 follow the same pattern of requirement AT51 but consider different gear values (gear values 2, 3, 4). The Requirements Table is not consistent and complete. The fact that each requirement specifies that if the gear (*gin*) changes to that value, the output variable modeling the gear (*g*) should maintain that value for at least 2.5s leads to inconsistent behavior. For example, if the gear switches from 1 to 2 at time instant 1 and from 2 to 3 at time instant 2, there is one requirement specifying that the gear shall remain 2 until at least 3.5s and another specifying that it shall remain 3 until at least 4.5s. However, between 2 and 3.5 the variable gear can not assume simultaneously the values 2 and 3 leading to the inconsistency.

Version v10. We changed the preconditions of the requirements by requiring that after switching gear value, the variable *gin* shall maintain that value for at least 2.5s. The Requirements Table is likely consistent since the consistency checker could not terminate within the timeout. The Requirements Table is complete.

Version v11. We added one requirement to our Requirements Table modeling the requirement AT6a from the ARCH report. This requirement specifies that if the engine RPM is lower than 3000 within the time interval [0,30], the vehicle speed shall be lower than 35 within the time interval [0,4]. Modeling this requirement required using the **dur** operator. Since a Requirements Table describes the relation between values assumed by input and output variables, we added an input variable to model the value of the monitored RPM (ω_{in}). The Requirements Table is likely consistent since the consistency checker could not terminate within the timeout. The Requirements Table is complete.

Version v12. We added the requirements AT6b and AT6c. The requirements AT6b and AT6c follow the same pattern of requirement AT6a but consider different velocity values and time bounds. The Requirements Table is likely consistent since the consistency checker could not terminate within the timeout. The Requirements Table is complete.

Version v12 contains all the requirements of the ARCH competition.

Version v13. We added one requirement to the Requirements Table forcing the speed to increase by 10 every second. This requirement can not be modeled in STL since it can not relate the value of a signal with one of its previous values (e.g., to specify that $v = \textbf{prev}(v) + 10$). The Requirements Table is not consistent. Requirement AT1 forces the vehicle speed to be lower than 120 within the interval [0,20]. However, this requirement can not be satisfied if the speed increases by 10 every second. The Requirements Table is complete.

Version v14. We changed the requirement added in version v13 to force the speed to increase by "1" every second. The Requirements Table is likely consistent since the consistency checker could not terminate within the timeout. The Requirements Table is complete.

RQ3 - Usefulness

THEANO was useful in supporting the development of a Requirements Table from a practical example. It could detect inconsistencies and confirm the consistency of respectively $\approx 14\%$ (2 out of 14) and $\approx 86\%$ (12 out of 14) versions of the Requirements Table. THEANO could find incompleteness and confirm the completeness of respectively $\approx 36\%$ (5 out of 14) and $\approx 64\%$ (9 out of 14) versions of the Requirements Table. In total, THEANO reported a problem (inconsistency or incompleteness) for 50% (7 out of 14) of the versions of the Requirements Table.

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., $\mathbf{prev}(v)$, and $\mathbf{dur}(\mathtt{le}) \geq c$), the formula $\mathfrak{h}_{\mathtt{i},\overline{\pi}}(\mathtt{pre})$ does not depend on the value considered for the position $\mathtt{i}$ and do not contain any quantification operator ($\forall$ or $\exists$). Therefore, checking the satisfiability of $\mathfrak{h}_{cp}(\mathtt{rt})$ reduces to verifying the satisfiability of a set of algebraic equations ($=$) or inequations ($<$, $\leq$, $\geq$, $>$) 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 $\mathfrak{h}_{\mathtt{i},\overline{\pi}}(\mathtt{pre})$ does not depend on the value considered for the position $\mathtt{i}$, the formula $\mathfrak{h}_{cp}(\mathtt{rt})$ contains a first-order logic formula containing one universal quantifier ($\forall \overline{\mathcal{Y}}$). Despite checking the satisfiability of these formulae is undecidable in general, there are fragments of first-order logic for which the problem is decidable [54]. 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., [55], [56], [38], [39], [57], [58]), 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., [59], [60], [61]). 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 [62]). However, these toolboxes rely on alternative syntax (e.g., [25], [62]) and semantics of completeness and disjointedness (e.g., [25], [11], [8], [26]) that significantly differ from the ones considered in this work. From the syntax perspective, unlike this work, existing works (e.g., [25], [62]) 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 [27], differ from the semantics of completeness and disjointedness proposed in other works (e.g., [25], [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 $\texttt{rt}$ to be complete if for every output variable $y \in \mathcal{Y}$, trace $\pi \in \Pi$, and position $\texttt{i}$, there exists a precondition $\texttt{pre} \in Pre(\texttt{rt}, y)$ satisfied by the trace π at position $\texttt{i}$, 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 $\texttt{rt}$ to be 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 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 $\iota_{\mathcal{U}}$ there exists an output $\iota_{\mathcal{Y}}$ and a timestamp ι_{τ} interpretation, such that the corresponding trace π satisfies the Requirements Table.

Pang et al. [63] 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 [64], (b) proposed approaches that enable their usage for supporting rigorous development of critical systems [65], [66], (c) developed tools that generate tabular expressions representing requirements from textual requirements documents [67], and the IEC 61131-3 Function Block Diagrams [68], (d) designed tools that generate formal models from tabular expressions [13], and (e) described methods that automatically generate test cases from tabular expressions [69].

Finally, several works assessed the benefits of tabular requirement specification on industrial applications [70], [71], [72], [73], [74].

Formal Languages Support. Several formal and logic-based (e.g, [75], [76], [77], [39], [78], [79], [80], [81], [82], [83], [84], [85]), and domain-specific (e.g., [86], [87], [88], [89], [90], [91], [34], [92]) languages, discussed by recent surveys on the topic (e.g. [93], [94], [95], [96]), were proposed by the research community to specify and analyze software requirements. Automated analyses support engineers in several activities including refactoring requirements [97], simplifying requirements formalization [98], and analyzing the applicability of existing tools on industrial CPS [99], [100], and their consistency [101], [102], [103], [104]. 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., [38], [105]), verifying legal compliance (e.g., [106]) extracting topological proofs showing why a property holds on a system (e.g., [55], [56]), validate B and Event-B models (e.g., [35]), verifying safety of synchronous fault-tolerant algorithms (e.g., [107]), analyze Alloy specifications (e.g., [108], [109], [110], [108]), find models that satisfy temporal-logic specifications [111], [112], [113], [36] and trace-checking (e.g., [37], [39]). 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 [114], [115], [116], [117], [118], [119], [120] often relying on domain modeling (e.g., [23], [121]). These works are discussed and analyzed by many surveys and studies on the topic (e.g. [122], [123], [124], [125], [122], [126], [127], [128], [126], [129], [120], [22]). 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.

Data Availability

We made our implementation and benchmark publicly available for experiment replication [40].

Acknowledgments

The work of the first author is funded by the European Union - Next Generation EU. "Sustainable Mobility Center (Centro Nazionale per la Mobilità Sostenibile - CNMS)", M4C2 - Investment 1.4, Project Code CN_00000023, and by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

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 Wassyng, and Dr. Srinath Avadhanula for their feedback during the early stages 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, "Rtamt: 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." *Thesis (Ph. D.)–University of North Carolina at Chapel Hill*, 1990.

[10] S. Meyers and S. White, "Software requirements methodology and tool study for a6-e technology transfer," *Technical report, Grumman Aerospace Corp., Bethpage, NY*, 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. Wassyng, "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. Wassyng, "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. Wassyng, 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] S. Easterbrook and B. Nuseibeh, "Managing inconsistencies in an evolving specification," in *International Symposium on Requirements Engineering (RE)*. IEEE, 1995, pp. 48–55.

[21] S. Lauesen, *Software requirements: styles and techniques*. Pearson Education, 2002.

[22] B. Nuseibeh and S. Easterbrook, "Requirements engineering: A roadmap," in *Conference on The Future of Software Engineering*. ACM, 2000, p. 35–46.

[23] 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.

[24] D. Leffingwell and D. Widrig, *Managing software requirements: a unified approach*. Addison-Wesley Professional, 2000.

[25] 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.

[26] A. Wassyng and R. Janicki, "Tabular expressions in software engineering," in *Proceedings of ICSSEA*, vol. 3, 2003, pp. 1–46.

[27] MathWorks. (2023) Identify Inconsistent and Incomplete Formal Requirement Sets. [Online]. Available: https://www.mathworks.com/help/slrequirements/ug/check-requirements-table-block.html

[28] 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.

[29] (2023) Identify inconsistent and incomplete formal requirement sets. [Online]. Available: https://www.mathworks.com/help/slrequirements/ug/check-requirements-table-block.html

[30] (2023) Compare solvers. [Online]. Available: https://www.mathworks.com/help/simulink/ug/compare-solvers.html

[31] antlr. (2023) antlr. [Online]. Available: https://www.antlr.org/

[32] (2020) Z3. [Online]. Available: https://ericpony.github.io/z3py-tutorial/guide-examples.htm

[33] (2020) Z3. [Online]. Available: https://github.com/Z3Prover/z3

[34] P. Filipovikj, G. Rodriguez-Navas, M. Nyberg, and C. Seceleanu, "Automated SMT-based consistency checking of industrial critical requirements," *SIGAPP Applied Computing Review*, vol. 17, no. 4, pp. 15–28, 2018, ACM.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] (2023) Theano. [Online]. Available: https://github.com/foselab/Theano/

[41] (2023) Simulink is for model-based design. [Online]. Available: https://www.mathworks.com/products/simulink

[42] D. Hilbert, "Mathematical problems," in *Mathematics*. Chapman and Hall/CRC, 2019, pp. 273–278.

[43] L. J. Mordell, *Diophantine equations*. Academic press, 1969.

[44] 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.

[45] (2023) Smt-lib the satisfiability modulo theories library. [Online]. Available: https://smtlib.cs.uiowa.edu/logics.shtml

[46] Wikipedia. (2023) ANTLR. [Online]. Available: https://en.wikipedia.org/wiki/ANTLR

[47] Tom Everett. (2023) MATLAB Grammar. [Online]. Available: https://github.com/antlr/grammars-v4/blob/master/matlab/matlab.g4

[48] "ACM SIGPLAN - Programming Languages Software Award," http://www.sigplan.org/Awards/Software/, 2020.

[49] "ETAPS 2018 Test of Time Award," https://etaps.org/about/test-of-time-award/test-of-time-award-2018, 2020.

[50] David Wingate. (2023) mparser. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/32769-mparser

[51] antlr3. (2023) ANTLRv3 parsing library for C. [Online]. Available: https://www.antlr3.org/

[52] B. Hoxha, H. Abbas, and G. Fainekos, "Benchmarks for temporal logic requirements for automotive systems," in *ARCH14-15. International Workshop on Applied veRification for Continuous and Hybrid Systems*, ser. EPiC Series in Computing. EasyChair, 2015, pp. 25–30. [Online]. Available: https://easychair.org/publications/paper/4bfq

[53] C. Menghi, P. Arcaini, W. Baptista, G. Ernst, G. Fainekos, F. Formica, S. Gon, T. Khandait, A. Kundu, G. Pedrielli, J. Peltomäki, I. Porres, R. Ray, M. Waga, and Z. Zhang, "Arch-comp 2023 category report: Falsification," in *10th International Workshop on Applied Verification of Continuous and Hybrid Systems. ARCH23*, vol. 96, 2023, pp. 151–169.

[54] Wikipedia. (2023) Decidability (logic). [Online]. Available: https://en.wikipedia.org/wiki/Decidability_(logic)

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] C. Heitmeyer, "Software cost reduction," *Encyclopedia of software engineering*, vol. 2, pp. 1374–1380, 2002.

[63] L. Pang, C.-W. Wang, M. Lawford, A. Wassyng, J. Newell, V. Chow, and D. Tremaine, "Formal verification of real-time function blocks using pvs," *arXiv preprint arXiv:1506.03557*, 2015.

[64] M. Bialy, M. Lawford, V. Pantelic, and A. Wassyng, "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.

[65] N. K. Singh, M. Lawford, T. S. Maibaum, and A. Wassyng, "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.

[66] 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.

[67] 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.

[68] J. Newell, L. Pang, D. Tremaine, A. Wassyng, 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.

[69] 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.

[70] 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.

[71] 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.

[72] J. Newell, L. Pang, D. Tremaine, A. Wassyng, 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.

[73] 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.

[74] 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.

[75] 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.

[76] L. Brim, P. Dluhoš, D. Šafránek, and T. Vejpustek, "STL*: Extending signal temporal logic with signal-value freezing operator," *Information and computation*, vol. 236, pp. 52–67, 2014, Elsevier.

[77] 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.

[78] A. Bakhirkin, T. Ferrère, T. A. Henzinger, and D. Ničkovićl, "Keynote: The first-order logic of signals," in *International Conference on Embedded Software*, 2018, pp. 1–10.

[79] R. Alur and T. A. Henzinger, "Real-time logics: complexity and expressiveness," *Information and Computation*, vol. 104, no. 1, pp. 35–77, 1993, Elsevier.

[80] 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.

[81] A. Cimatti, M. Roveri, and S. Tonetta, "HRELTL: A temporal logic for hybrid systems," *Information and Computation*, vol. 245, pp. 54–71, 2015, Elsevier.

[82] 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.

[83] 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.

[84] F. Mallet, "Clock constraint specification language: specifying clock constraints with uml/marte," *Innovations in Systems and Software Engineering*, vol. 4, pp. 309–314, 2008.

[85] V. Langenfeld, D. Dietsch, B. Westphal, J. Hoenicke, and A. Post, "Scalable analysis of real-time requirements," in *International Requirements Engineering Conference (RE)*. IEEE, 2019, pp. 234–244.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] 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.

[97] M. Farrell, M. Luckcuck, O. Sheridan, and R. Monahan, "Towards refactoring fretish requirements," in *NASA Formal Methods Symposium*. Springer, 2022, pp. 272–279.

[98] 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.

[99] 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.

[100] 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.

[101] J. S. Becker, "Analyzing consistency of formal requirements," *Electronic Communications of the EASST*, vol. 76, 2019.

[102] 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.

[103] N. Mahmud, C. Seceleanu, 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.

[104] J. Bendík, "Consistency checking in requirements analysis," in *International Symposium on Software Testing and Analysis*. ACM, 2017, p. 408–411.

[105] 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.

[106] 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.

[107] 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.

[108] 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.

[109] 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.

[110] 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.

[111] 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.

[112] 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.

[113] 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.

[114] 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.
[115] 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.
[116] 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.
[117] 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.
[118] 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.
[119] 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.
[120] 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.
[121] 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.
[122] M. Kamalrudin and S. Sidek, "A review on software requirements validation and consistency management," *International journal of software engineering and its applications*, vol. 9, no. 10, pp. 39–58, 2015.
[123] I. Hadar, A. Zamansky, and D. M. Berry, "The inconsistency between theory and practice in managing inconsistency in requirements engineering," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3972–4005, 2019, Springer.
[124] R. Delima, K. Mustofa, A. K. Sari *et al.*, "Automatic requirements engineering: Activities, methods, tools, and domains–a systematic literature review," *Jurnal RESTI (Rekayasa Sistem dan Teknologi Informasi)*, vol. 7, no. 3, pp. 564–578, 2023.
[125] M. Kamalrudin, N. Mustafa, and S. Sidek, "A preliminary study: Challenges in capturing security requirements and consistency checking by requirement engineers," *Journal of Telecommunication, Electronic and Computer Engineering*, vol. 10, no. 1-7, pp. 5–9, 2018.
[126] D. Zowghi and V. Gervasi, "On the interplay between consistency, completeness, and correctness in requirements evolution," *Information and Software technology*, vol. 45, no. 14, pp. 993–1009, 2003.
[127] A. Chattopadhyay, G. Malla, N. Niu, T. Bhowmik, and J. Savolainen, "Completeness of natural language requirements: A comparative study of user stories and feature descriptions," in *International Conference on Information Reuse and Integration for Data Science*. IEEE, 2023, pp. 52–57.
[128] G. D. Hadad, C. S. Litvak, J. H. Doorn, and M. Ridao, "Dealing with completeness in requirements engineering," in *Encyclopedia of Information Science and Technology, Third Edition*. IGI Global, 2015, pp. 2854–2863.
[129] B. Nuseibeh, S. Easterbrook, and A. Russo, "Leveraging inconsistency in software development," *Computer*, vol. 33, no. 4, pp. 24–29, 2000.

Claudio Menghi is an Associate Professor at the "Dipartimento di Ingegneria gestionale, dell'informazione e della produzione", University of Bergamo (Italy) and an Adjunct Professor at the Department of Computing and Software, McMaster University (Canada). After receiving his PhD at Politecnico di Milano, he was a post-doctoral researcher at Chalmers | University of Gothenburg (Sweden), an Associate Researcher at the University of Luxembourg (Luxembourg), and an Assistant Professor at the Department of Computing and Software, McMaster University (Canada). His current research interests lie in software engineering, with a special interest in cyber-physical systems (CPS), and formal verification.

Eugene Balai is a Principal Software Engineer at MathWorks. He works on improving Stateflow semantics and architecture. He obtained his Ph.D in computer science from Texas Tech University, with a thesis focused on combining non-monotonic logic and probabilistic reasoning.

Christoph Sticksel is a Development Manager for Simulink Design Verifier. He worked on model checking of reactive systems in projects as a postdoctoral researcher at the University of Iowa and the University of Manchester. He obtained his Ph.D. in automated reasoning from the University of Manchester, UK.

Akshay Rajhans (S'08–M'13–SM'19) is a Principal Research Scientist and the Head of the Advanced Research & Technology Office at MathWorks. He obtained his PhD from Carnegie Mellon University and MS from University of Pennsylvania. His research interests include model-based design and formal analysis of cyber-physical systems. Earlier in his career he worked on software engineering in the Simulink Semantics group at MathWorks, and on research and development of embedded control systems at Cummins and Bosch.