Metric Interval Temporal Logic Specification Elicitation and Debugging

Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos
School of Computing, Informatics and Decision Systems
Arizona State University, Tempe, AZ, U.S.A.
Email: {adokhanc,bhoxha,fainekos}@asu.edu

***Abstract*—In general, system testing and verification should be conducted with respect to formal specifications. However, the development of formal specifications is a challenging and error prone task, even for experts. This is especially true when considering complex spatio-temporal requirements in real-time embedded systems, mixed-signal circuits, or more generally, software-controlled physical systems. In this work, we present a framework for the elicitation and debugging of formal specifications. The elicitation of formal specifications is handled through a graphical user interface. The debugging algorithm checks inconsistent and wrong specifications. Namely, it detects validity, redundancy and vacuity issues in formal specifications developed in a fragment of Metric Interval Temporal Logic (MITL). The algorithm informs system engineers on any issues in their specifications. This improves the specification elicitation process and, ultimately, the testing and verification process. Finally, we present experimental results on specifications that typically appear in Cyber Physical Systems (CPS) applications. Application of our specification debugging tool on user derived requirements shows that the aforementioned issues are common. Therefore, the algorithm can help developers to correct their specifications and avoid wasted effort on checking incorrect requirements.**

I. Introduction

In formal verification of Cyber-Physical Systems (CPS), a system is verified with respect to formal specifications. It has been shown that utilizing formal specifications can lead to improved testing and verification [16], [22], [31]. However, developing formal specifications using logics is a challenging and error prone task even for experts who have formal mathematical training. Therefore, in practice, system engineers usually define specifications in natural language. Natural language is convenient to use in many stages of system development, but its inherent ambiguity, inaccuracy and inconsistency make it unsuitable for use in defining specifications.

To assist in the elicitation of formal specifications, in [20], [21], we presented a graphical formalism and tool VISPEC that can be utilized by both expert and non-expert users. Namely, a user-developed graphical input is translated to a Metric Temporal Logic (MTL) formula. The formal specifications in MTL can be used for testing and verification with tools such as S-TALIRO [2] and Breach [14].

In [21], the tool was evaluated through a usability study which showed that both expert and non-expert users were able to use the tool to elicit formal specifications. The usability study results also indicated that in many cases the developed specifications were incorrect. Namely, the specifications contained logical inconsistencies or they were (partially) wrong[1]. This raised two questions. First, are these issues artifacts of the graphical user interface? Second, can we automatically detect and report issues with the requirements themselves?

We have created an on-line survey[2] to answer the first question. Namely, we conducted a usability study on Metric Interval Temporal Logic (MITL) by targeting experts in temporal logics. In our on-line survey, we tested how well formal method experts can translate natural requirements to MITL. That is, given a set of requirements in natural language, experts were asked to formalize the requirements in MITL. The study is ongoing but preliminary results indicate that even experts can make errors in their specifications. For example, for the natural language specification *"At some time in the first 30 seconds, the vehicle speed (v) will go over 100 and stay above 100 for 20 seconds"*, the specification $\varphi = \Diamond_{[0,30]}((v > 100) \Rightarrow \Box_{[0,20]}(v > 100))$ was provided. Here, $\Diamond_{[0,30]}$ stands for "*eventually within 30 time units*" and $\Box_{[0,20]}$ for "*always from 0 to 20 time units*".

However, specification φ is invalid. This is because, if at any point in time between 0 and 30 seconds the predicate $(v > 100)$ is false, then the specification evaluates to true. On the other hand, if the predicate $(v > 100)$ is true for all time between 0 and 30 seconds, then the subformula $\Box_{[0,20]}(v > 100)$ will be true at any time between 0 and 10 seconds. This means that the subformula $(v > 100) \Rightarrow \Box_{[0,20]}(v > 100)$ is true between 0 and 10 seconds. Thus, again, φ evaluates to true, which means that φ is a *tautology*! This implies that specification issues are not necessarily artifacts of the graphical user interface and that they can happen even for the people who are familiar with temporal logics.

This indicates that the specification elicitation can be a major issue in testing and verification since effort can be wasted in checking incorrect requirements, or even worse, the system can pass the incorrect requirements. Clearly, this can lead to a false sense of system correctness, which leads us to the second question: What can be done automatically to prevent specification errors in CPS?

In this work, we have developed a specification development framework that would enable the elicitation and debugging of specifications. The specification debugging algorithm identifies invalid and wrong specifications. Namely, it performs the following in order:

[1] In section V, we will define what we mean by wrong specifications.

[2] The on-line survey is available through: http://goo.gl/forms/YW0reiDtgi

1) Validity detection: the specification is unsatisfiable or a tautology.
2) Redundancy detection: the formula has redundant conjuncts.
3) Vacuity detection: some subformulas do not affect the satisfiability of the formula. This usually indicates some misunderstanding in the requirements.

Summary of Contributions:

1) We present a debugging algorithm for a fragment of MITL specifications.
2) We extend Linear Temporal Logic (LTL) [12] vacuity detection algorithms [10] to real-time specifications in MITL.
3) We present experimental results on specifications that typically appear in CPS specifications.

The above contributions solve the specification correctness problem for VISPEC [21] requirements. The user of VISPEC can benefit from our feed-back and fix any reported issues.

II. Related works

The challenge of developing formal specifications has been studied in the past. The most relevant works appear in [4] and [32]. In [4], the authors extend Message Sequence Charts and UML 2.0 Interaction Sequence Diagrams to propose a scenario based formalism called Property Sequence Chart (PSC). The formalism is mainly developed for specifications on concurrent systems. In [32], PSC is extended to Timed PSC which enables the addition of timing constructs to specifications.

Specification debugging can also be considered in areas such as system synthesis [30] and software verification [1]. In system synthesis, realizability is an important factor, which checks whether the system is implementable given the constraints (environment) and requirements (specification) [15], [24], [11], [30]. Specification debugging can also be considered with respect to the environment for robot motion planing. In [17], [23], the authors considered the problem where the original specification is unsatisfiable with the given environment and robot actions. Then, they relax the specification in order to render it satisfiable in the given domain.

One of the most powerful verification methods is model checking [12] where the model of the system is evaluated with respect to a specification. For example, let us consider model checking with respect to LTL formulas. It is possible that the model satisfies the specification but not in the intended way. This may hide actual problems in the model. These satisfactions are called *vacuous* satisfactions. Antecedent failure was the first problem that raised the vacuity as a serious issue in verification [5], [6]. For example, $\varphi = \Box_{[0,5]}(req \Rightarrow \Diamond_{[0,10]}ack)$ is interpreted as "if at any time within the first 5 seconds, a *request* happens, then from that moment on within the next 10 seconds, an *acknowledge* must happen". Here, φ can be vacuously satisfiable since it can be satisfied in all systems in which a *request* never happens. Vacuity can be addressed with respect to a model [3], [27], [19], [26] or without a model [18], [10]. A formula is vacuous when it can be simplified to a smaller equivalent formula. It has been proven in [18] that a specification φ is satisfied vacuously in all systems that satisfy it iff φ is equivalent to some mutations of it. In [10], they provide an algorithmic approach to detecting vacuity and redundancy in LTL specifications.

Our work extends [10] and it is applied to a fragment of MITL. We provide a new definition of vacuity with respect to Boolean or real-value signals. To the best of our knowledge, vacuity of real-time properties such as MITL has not been addressed yet. Although this problem is computationally hard, due to the small size of the formulas, in practice the computation problem is manageable.

III. Preliminaries

In this work, we take a general approach in modeling Cyber Physical Systems (CPS). In the following, $\mathbb{R}$ is the set of real numbers, $\mathbb{R}_+$ is the set of non-negative real numbers, $\mathbb{Q}$ is the set of rational numbers, $\mathbb{Q}_+$ is the set of non-negative rational numbers. Given two sets A and B, B^A is the set of all functions from A to B, i.e., for any $f \in B^A$ we have $f : A \to B$. We define 2^A to be the power set of set A. We fix $T \in \mathbb{R}_+$ to be the maximum time of a signal.

A. Metric Interval Temporal Logic

Metric Temporal Logic (MTL) was introduced in [25] in order to reason about the quantitative timing properties of boolean signals. Metric Interval Temporal Logic (MITL) is MTL where the timing constraints are not allowed to be singleton sets. In the rest of the paper, we restrict our focus to a fragment of MITL called Bounded-MITL(◇,□) where the only temporal operators allowed are *Eventually* (◇) and *Always* (□) operators with timing intervals. Formally, the syntax of Bounded-MITL(◇,□) is presented by the following grammar:

Definition 1 (Bounded-MITL(◇,□) syntax):

$$\phi \ ::= \ \top \mid \bot \mid a \mid \neg a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \Diamond_I \phi_1 \mid \Box_I \phi_1$$

where AP is the set of atomic propositions and $a \in AP$, $\top$ is True, $\bot$ is False. Also, I is a nonsingular interval over $\mathbb{Q}_+$ with defined end-points. The interval I is right-closed.

Definition 2 (Bounded-MITL(◇,□) semantics): Given a time trace $\mu : [0,T] \to 2^{AP}$ and $t, t' \in \mathbb{R}$, and an MITL formula ϕ, the satisfaction relation $(\mu, t) \models \phi$ is inductively defined:

$$(\mu,t) \models \top$$

$$(\mu,t) \models a \text{ iff } a \in \mu(t)$$

$$(\mu,t) \models \neg a \text{ iff } a \notin \mu(t)$$

$$(\mu,t) \models \varphi_1 \wedge \varphi_2 \text{ iff } (\mu,t) \models \varphi_1 \text{ and } (\mu,t) \models \varphi_2$$

$$(\mu,t) \models \varphi_1 \vee \varphi_2 \text{ iff } (\mu,t) \models \varphi_1 \text{ or } (\mu,t) \models \varphi_2$$

$$(\mu,t) \models \Diamond_I \varphi_1 \text{ iff } \exists t' \in (t+I) \cap [0,T] \text{ s.t } (\mu,t') \models \varphi_1.$$

$$(\mu,t) \models \Box_I \varphi_1 \text{ iff } \forall t' \in (t+I) \cap [0,T],\ (\mu,t') \models \varphi_1.$$

Where $(t + I)$ creates a new interval I' where if $I = [l, u]$ then $I' = [l + t, u + t]$.
A boolean signal μ satisfies a Bounded-MITL(◇,□) formula ϕ (denoted by $\mu \models \phi$), iff $(\mu, 0) \models \phi$.

The *Implication* ($\Rightarrow$) is defined as $\psi \Rightarrow \varphi \equiv \neg\psi \vee \varphi$, and also $\bot \equiv \neg\top$. In this paper, we assume that Bounded-MITL(◇,□)

formula is in Negation Normal Form (NNF) where the negation operation is only applied on atomic propositions. NNF is easily obtainable by applying DeMorgan's Law, i.e $\neg\Diamond_I\varphi \equiv \Box_I\neg\varphi$ and $\neg\Box_I\varphi \equiv \Diamond_I\neg\varphi$. For simplifying the presentation, when we mention MITL we mean Bounded-MITL($\Diamond$,$\Box$). Given MITL formulas φ and ψ, φ satisfies ψ, denoted by $\varphi \models \psi$ iff $\forall\mu.\mu \models \varphi \Rightarrow \mu \models \psi$.

B. Signal Temporal Logic

The logic and semantics can be extended to real-valued signals through Signal Temporal Logic (STL) [29].

Definition 3 (Signal Temporal Logic [29]): Let $s : [0, T] \rightarrow \mathbb{R}^m$ be a real-time signal, and $P = \{p_1, ..., p_n\}$ be a collection of predicates or boolean functions of the form $p_i : \mathbb{R}^m \rightarrow \mathbb{B}$ where $\mathbb{B} = \{\top, \bot\}$ is a boolean value.

We define the STL formula Φ_{STL} over predicates P using MITL formula Φ_{MITL} over the atomic propositions AP. The semantics of STL can be defined using MITL as follows:

1) Define a set of AP such that for each $p \in P$, there exist some $a_p \in AP$
2) For each signal s we define a μ such that $a_p \in \mu(t)$ iff $p(s(t)) = \top$
3) $\forall t\ (s, t) \vDash \Phi_{STL}$ iff $(\mu, t) \vDash \Phi_{MITL}$

C. Visual Specification Tool

The Visual Specification Tool (VISPEC) [21] enables the development of formal specifications for CPS. The graphical formalism enables reasoning on both timing and event sequence occurrence. Consider the specification $\phi_{cps} = \Box_{[0,30]}((speed > 100) \Rightarrow \Box_{[0,40]}(rpm > 4000))$. It states that whenever within the first 30 seconds, *vehicle speed* goes over 100, then from that moment on, the *engine speed (rpm)*, for the next 40 seconds, should always be above 4000. Here both the sequence and timing of the events are of critical importance. See Fig. 1 for the visual representation of ϕ_{cps}.

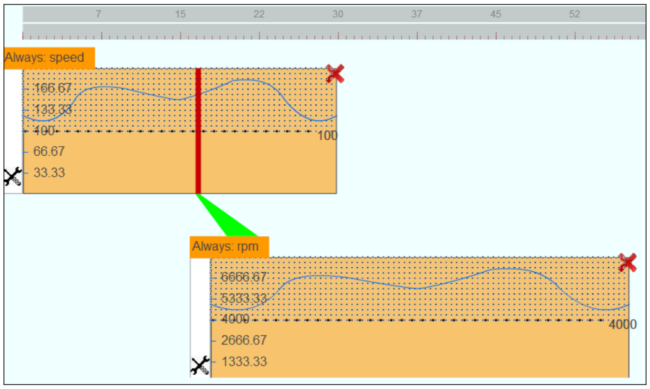

Fig. 1. Graphical representation of $\phi_{cps} = \Box_{[0,30]}((speed > 100) \Rightarrow \Box_{[0,40]}(rpm > 4000))$.

Users develop specifications using a visual formalism which can be translated to a Metric Temporal Logic formula. The set of specifications that can be generated from this graphical formalism is a proper subset of the set of MTL specifications. Formally, the following grammar produces the set of formulas that can be expressed by the proposed graphical formalism:

$$\begin{array}{lcl}
\mathrm{S} & \longrightarrow & \neg\mathrm{T} \mid \mathrm{T} \\
\mathrm{T} & \longrightarrow & \mathrm{A} \mid \mathrm{B} \mid \mathrm{C} \\
\mathrm{A} & \longrightarrow & \mathrm{P} \mid (\mathrm{P}\wedge\mathrm{A}) \mid (\mathrm{P}\Rightarrow\mathrm{A}) \\
\mathrm{B} & \longrightarrow & \Box_{\mathcal{I}}\mathrm{D} \mid \Diamond_{\mathcal{I}}\mathrm{D} \\
\mathrm{C} & \longrightarrow & \Box_{\mathcal{I}}\Diamond_{\mathcal{I}}\mathrm{D} \mid \Diamond_{\mathcal{I}}\Box_{\mathcal{I}}\mathrm{D} \\
\mathrm{D} & \longrightarrow & p \mid (p\Rightarrow\mathrm{A}) \mid (p\wedge\mathrm{A}) \mid (p\Rightarrow\mathrm{B}) \mid (p\wedge\mathrm{B}) \\
\mathrm{P} & \longrightarrow & p \mid \Box_{\mathcal{I}}p \mid \Diamond_{\mathcal{I}}p
\end{array}$$

where p is an atomic proposition. In the tool, the atomic propositions are automatically derived from the templates. For example the formula $\Box_{\mathcal{I}}\Diamond_{\mathcal{I}}p$ can be generated using the following parse tree $\mathrm{S}\longrightarrow\mathrm{T}\longrightarrow\mathrm{C}\longrightarrow \Box_{\mathcal{I}}\Diamond_{\mathcal{I}}\mathrm{D}\longrightarrow \Box_{\mathcal{I}}\Diamond_{\mathcal{I}}p$.

The graphical formalism was developed with the following goals: a) The user interface is easy to use, i.e, it does not have a high learning curve; b) The visual representation of the requirements is clear and unambiguous; c) There is a one-to-one mapping from the visual representation of the requirement and the corresponding requirement in MTL. The graphical formalism is mainly composed of the following: 1) Templates; 2) Relationships between templates.

Templates are used to define temporal logic operators, their timing intervals, and the expected signal shape. A template configuration wizard guides the user in the development process. The process is context dependent where each option selection leads to a potentially different set of options for the next step.

After the selection of the temporal operator, the user will define the timing bounds for it. For specifications with temporal operators such as *Eventually Always* ($\Diamond\Box$) and *Repeatedly Often and Finally* ($\Box\Diamond$), setting the timing bounds may be a challenging task. To clarify this issue, the tool provides a fill-in-the-blanks sentence format to the user. For example, if the operator *Eventually Always* is selected, the user will have to complete the following sentence with the timing bounds: "Eventually, between ____ and ____ seconds, the signal will become true, and from that point on, will stay true in the next ____ to ____ seconds". The set timing intervals are visualized with color shaded regions in the template.

The next step in the process is in defining whether the predicate will evaluate to true when the signal is above or below a set threshold. For example, for the *Always* ($\Box$) operator, a signal is selected that is either always above or below a specified threshold. Once either option is selected, a signal that fits the requirement is automatically generated and presented visually (See Fig. 1).

Relationships between templates enable the development of more complex specifications. The three main relationships between templates are the following: 1) Templates can be placed in a sequence, where the last template is only considered if the previous templates are evaluated to true. Formally, it enables the definition of an implication relationship between templates of the form $\phi \Rightarrow \psi$. 2) Templates can be grouped to establish a conjunction relationship of the form $\phi \wedge \psi$. This is indicated visually by a black box around the templates. 3) Finally, the relative timing relationship enables the definition of: a) Reactive response specifications of the form $\Box(\phi \Rightarrow M\psi)$; b) Non-strict sequencing specifications of the form $N(\phi \wedge M\psi)$, where N and M are temporal operators. This relationship is visually distinct in that the nested template is tabbed in relation to the main template.

The variety of templates and the connections between them allow users to express a wide variety of specifications as presented in Table I.

IV. Elicitation Framework for MITL

Our framework for elicitation of MITL specifications is presented in Fig. 2. Once a specification is developed using ViSpec, it is translated into STL. Then, we create the corresponding MITL formula from STL. Next, the MITL specification is analyzed by the debugging algorithm which returns an alert to the user in case the specification has inconsistency or correctness issues. The debugging process is explained in detail in the next section.

To enable the debugging of specifications, we must project the STL predicate expressions into atomic propositions with independent truth valuations. For example, consider the STL specification $\phi_{stl} = \Diamond_{[0,10]}((speed > 100) \wedge \Diamond_{[0,10]}(speed > 80))$. In this case, the subformula $\Diamond_{[0,10]}(speed > 80)$ does not affect the satisfaction of the specification. This indicates that there is an issue with the specification.

However, if we simply replace the predicate expressions $speed > 100$ and $speed > 80$ with the atomic propositions a and b, respectively, then the resulting MITL formula will be $\phi_{mitl} = \Diamond_{[0,10]}(a \wedge \Diamond_{[0,10]}b)$. Thus, we lose information about the intrinsic dependency between a and b and debugging will not find the issue. In order to enable the logical analysis of such formulas in our debugging process, we replace the original predicate expressions with atomic propositions with non-overlapping corresponding predicates (Boolean functions). For this example, the resulting MITL specification should be $\phi_{mitl} = \Diamond_{[0,10]}(q_1 \wedge \Diamond_{[0,10]}(q_1 \vee q_2))$ where q_1 corresponds to $speed > 100$ and q_2 corresponds to $100 \geq speed > 80$. The projection in the current implementation is conducted using a brute-force algorithm that runs through all the combinations of predicate expressions to find overlapping areas.

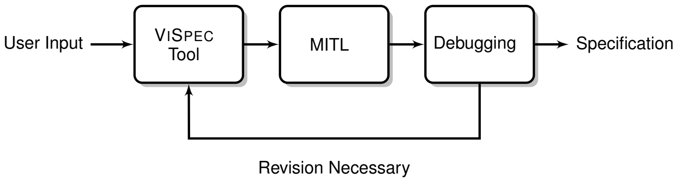

Fig. 2. Specification Elicitation Framework

V. Specification Debugging for MITL

Clearly, verifying a system with respect to incorrect specifications is pointless. Therefore, any inconsistencies or other issues with the specification should be resolved. In the following, we present algorithms that can detect inconsistency and correctness issues in specifications. This will help the user in the elicitation of correct specifications.

Our specification debugging process conducts the following checks in this order: 1) Validity, 2) Redundancy, and 3) Vacuity. In brief, validity checking determines whether the specification is satisfiable but not a tautology. Namely, if the specification is unsatisfiable no system can satisfy it and if it is a tautology every system can trivially satisfy it. For example, $p \vee \neg p$ is a tautology.

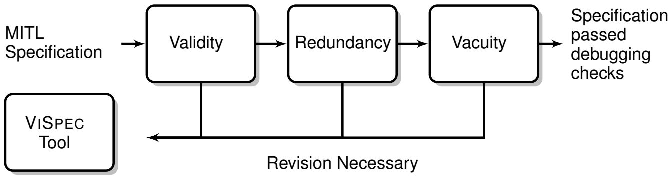

Fig. 3. Specification Debugging

Redundancy checking determines whether the specification has no redundant conjunct when the specification is a conjunction of MITL formulas. For example, in the specification $p \wedge \Box_{[0,10]}p$, the first conjunct is redundant. Sometimes redundancy is related to incomplete or erroneous requrements where the user may have wanted to specify something else. Therefore, the user should be notified.

Vacuity checking determines whether the specification has a subformula that does not have any affect on the satisfaction of the specification. For example $\varphi = p \vee \Diamond_{[0,10]}p$ is vacuous since the first occurrence of p does not have any affect on the satisfaction of φ.

Definition 4 (Wrong Specification): A specification which is redundant or vacuous is called *wrong*.

The reason that we choose the term “wrong” is that although this specification is logically valid, the specification in its current representation does not reflect the intention of the requirement in its natural language form. This is because part of the specification is over-shadowed with the other components.

The debugging process is presented in Fig. 3. First, given a specification, a validity check is conducted. If a formula does not pass a validity check then it means that there is a major problem in the specification and the formula is returned for revision. Therefore, redundancy and vacuity checks are not relevant at that point. Similarly, if the specification is redundant it means that it has a conjunct that does not have any affect on the satisfaction of the specification and we return the redundant conjunct for revision. Lastly, if the specification is vacuous it is returned with the issue for revision by the user.

A. Redundancy Checking

Recall that a specification has a redundancy issue if one of its conjuncts can be removed without affecting the models of the specification. Before we formally present what redundant requirements are, we have to introduce some notation. We consider specification Φ as a conjunction of MITL subformulas (φ_j):

$$\Phi = \bigwedge\nolimits_{j=1}^{k} \varphi_j \tag{1}$$

To simplify discussion, we will abuse notation and we will associate a conjunctive formula with the set of its conjuncts.

TABLE I. Classes of specifications expressible with the graphical formalism

Specification Class	Explanation
Safety	Specifications of the form $\Box\phi$ used to define specifications where ϕ should always be true.
Reachability	Specifications of the form $\Diamond\phi$ used to define specifications where ϕ should be true at least once in the future (or now).
Stabilization	Specifications of the form $\Diamond\Box\phi$ used to define specifications that, at least once, ϕ should be true and from that point on, stay true.
Oscillation	Specifications of the form $\Box\Diamond\phi$ used to define specifications that, it is always the case, that at some point in the future, ϕ repeatedly will become true.
Implication	Specifications of the form $\phi \Rightarrow \psi$ requires that ψ should hold when ϕ is true.
Reactive Response	Specifications of the form $\Box(\phi \Rightarrow M\psi)$, where M is temporal operator, used to define an implicative response between two specifications where the timing of M is relative to timing of $\Box$.
Conjunction	Specifications of the form $\phi \wedge \psi$ used to define the conjunction of two sub-specifications.
Non-strict Sequencing	Specifications of the form $N(\phi \wedge M\psi)$, where N and M are temporal operators, used to define a conjunction between two specifications where the timing of M is relative to timing of N.

That is:

$$\Phi = \{\varphi_j \mid j = 1, ..., k\} \tag{2}$$

Similarly, $\{\Phi\backslash\varphi_i\}$ represents the specification Φ where the conjunct φ_i is removed:

$$\{\Phi\backslash\varphi_i\} = \{\varphi_j \mid j = 1, ..., i-1, i+1, ..., k\} = \bigwedge_{j=1}^{i-1}\varphi_j \wedge \bigwedge_{j=i+1}^{k}\varphi_j \tag{3}$$

Whether $\{\Phi\backslash\varphi_i\}$ represents a set or a conjunctive formula will be clear from the context. Redundancy in specifications is fairly common in practice due to the incremental additive approach that system engineers take in the development of specifications. In the following, we consider the redundancy removal algorithm provided in [10] for LTL formulas and we extend it to support MITL formulas.

Definition 5 (Redundancy of Specification): A conjunct φ_i is redundant with respect to Φ if

$$\bigwedge_{\psi\in\{\Phi\backslash\varphi_i\}}\psi \quad \models \varphi_i$$

To reformulate, φ_i is redundant with respect to Φ if $\{\Phi\backslash\varphi_i\} \models \varphi_i$. For example, in $\Phi = \Diamond_{[0,10]}(p \wedge q) \wedge \Diamond_{[0,10]}p \wedge \Box_{[0,10]}q$, the conjunct $\Diamond_{[0,10]}(p \wedge q)$ is redundant with respect to $\Diamond_{[0,10]}p \wedge \Box_{[0,10]}q$ since $\Diamond_{[0,10]}p \wedge \Box_{[0,10]}q \models \Diamond_{[0,10]}(p \wedge q)$. In addition, $\Diamond_{[0,10]}p$ is redundant with respect to $\Diamond_{[0,10]}(p \wedge q) \wedge \Box_{[0,10]}q$ since $\Diamond_{[0,10]}(p \wedge q) \wedge \Box_{[0,10]}q \models \Diamond_{[0,10]}p$. This method can catch both the issues and report them to the user.

Algorithm 1 finds redundant subformulas, if they exist and it provides the list of subformulas that are redundant with respect to Φ as the feedback to the user. It should be noted that if a specification has nested subformulas in the conjunctive form, then redundancy checking can be used to find the redundant conjuncts. For example, $\varphi = \Diamond_{[0,10]}(p \wedge \Box_{[0,10]}p)$ can be checked by Algorithm 1 if $p \wedge \Box_{[0,10]}p$ is given as input to the algorithm instead of φ.

B. Vacuity Checking

Vacuity detection is used to ensure that all the subformulas of the specification contribute to the satisfaction of the specification. In other words, the vacuity check enables the detection of irrelevant subformulas in the specifications [10]. In the following, we provide the definition of MITL vacuity with respect to signal.

Definition 6 (MITL Vacuity with respect to signal): Given a signal $\mathcal{T}$ and an MITL formula φ. A subformula ψ of φ does not affect the satisfiability of φ with respect to $\mathcal{T}$ if and only if ψ can be replaced with any subformula θ without changing the satisfiability of φ on $\mathcal{T}$. A specification φ is satisfied vacuously by $\mathcal{T}$, denoted by $\mathcal{T} \models_V \varphi$, if there exists ψ which does not affect the satisfiability of φ on $\mathcal{T}$.

Algorithm 1 Redundancy Checking

```
Input: $\Phi$ ($MITL$ Specification)
Output: $RL_\varphi$ a list of redundant formulas
$RL_\varphi \leftarrow \emptyset$
for each formula $\varphi_i \in \Phi$ do
    if $(\Phi\backslash\varphi_i) \models \varphi_i$ then
        $RL_\varphi \leftarrow \varphi_i$
    end if
end for
```

In the following, we extend the framework presented in [10] to support MITL specifications. Let φ be a formula in NNF where only predicates can be in the negated form. A $literal$ is defined as a predicate or its negation. For formula φ the set of literals of φ is denoted by $literal(\varphi)$ and contains all the literals appearing in φ. For example if $\varphi = (\neg p \wedge q) \vee \Diamond_{[0,10]}p \vee \Box_{[0,10]}q$ then $literal(\varphi) = \{\neg p, q, p\}$. Literal occurrences, denoted by $litOccur(\varphi)$, is a multiset of literals appearing in some order in φ, e.g., by traversal of the parse tree. For the given example $litOccur(\varphi) = \{\neg p, q, p, q\}$. For each $l \in litOccur(\varphi)$ we create the mutation of φ by substituting the occurrence of l with $\bot$. We denote the mutated formula as $\varphi[l \leftarrow \bot]$.

Definition 7 ($MITL$ Vacuity w.r.t. literal occurrence): Given a signal $\mathcal{T}$ and an $MITL$ formula φ. Specification φ is vacuously satisfied by $\mathcal{T}$ if there exists a literal occurrence $l \in litOccur(\varphi)$ such that $\mathcal{T}$ satisfies the mutated formula $\varphi[l \leftarrow \bot]$. Formally, $\mathcal{T} \models_V \varphi$ if $\exists l \in litOccur(\varphi)$ s.t. $\mathcal{T} \models \varphi[l \leftarrow \bot]$.

Theorem 1 ($MITL$ Vacuity with respect to Specification): Assume that the specification Φ is a conjunction of MITL formulas. If $\exists \varphi_i \in \Phi$ and $\exists l \in litOccur(\varphi_i)$, such that $\Phi \models \varphi_i[l \leftarrow \bot]$, then Φ satisfies φ_i vacuously ($\Phi \models_V \varphi_i$).

The proof is straightforward modification of the proofs given in [10], [27]. We have added the proof in Appendix (Section IX-A) for completeness. When we do not have the conjunction in the specification ($\Phi = \varphi$), we check the vacuity of the formula with respect to itself. In other words, we check whether the specification satisfies its mutation ($\varphi \models \varphi[l \leftarrow \bot]$ or $\varphi \models_V \varphi$). Algorithm 2 finds the vacuous subformulas of the specification similar to [10].

Algorithm 2 Vacuity Checking

```
Input: $\Phi$ ($MITL$ Specification)
Output: $VL_\varphi$ a list of vacuous formulas
$VL_\varphi \leftarrow \emptyset$
for each formula $\varphi_i \in \Phi$ do
    for each $l \in litOccur(\varphi_i)$ do
        if $\Phi \models \varphi_i[l \leftarrow \bot]$ then
            $VL_\varphi \leftarrow \varphi_i[l \leftarrow \bot]$
        end if
    end for
end for
```

VI. Experimental Analysis

All the 3-level correctness analysis of MITL specifications need satisfiability checking as the underlying tool [9]. In validity checking we simply check whether the specification and its negation are satisfiable. In general, in order to check whether $\varphi \models \psi$, we should check whether $\varphi \implies \psi$ is a tautology, that is $\forall \mu, \mu \models \varphi \implies \psi$. This can be verified by checking whether $\neg(\varphi \implies \psi)$ is unsatisfiable.

Recall that $\varphi \implies \psi$ is equivalent to $\neg\varphi \vee \psi$. So we have to check whether $\varphi \wedge \neg\psi$ is unsatisfiable to conclude that $\varphi \models \psi$. We use the above reasoning for redundancy checking as well as for vacuity checking. For redundancy checking, $\{\Phi \backslash \varphi_i\} \wedge \neg\varphi_i$ should be unsatisfiable, in order to reason that $\{\Phi \backslash \varphi_i\} \models \varphi_i$. For vacuity checking, $\Phi \wedge \neg(\varphi_i[l \leftarrow \bot])$ should be unsatisfiable, in order to prove that $\Phi \models \varphi_i[l \leftarrow \bot]$.

A. MITL Satisfiability

As mentioned earlier, we can check all evaluations of a specification using a satisfiability checker. In order to check whether an MITL formula is satisfiable we use two publicly available tools: qtlsolver[3] and zot[4]. The qtlsolver that we used, translates MITL formulas into CLTL-over-clocks [7], [9]. Constraint LTL (CLTL) is an extension of LTL where predicates are allowed to be the assertions on the values of non-Boolean variables [13]. That is, in CLTL, we are allowed to define predicates using relational operators for variable over domains like $\mathbb{N}$ and $\mathbb{Z}$. Although satisfiability of CLTL in general is not decidable, some variant of it is decidable [13].

CLTLoc (CLTL-over-clocks) is a variant of CLTL where the clock variables are the only arithmetic variables that are considered in the atomic constraints. It has been proved in [8] that CLTLoc is equivalent to timed automata [12]. Moreover, it can be polynomially reduced to decidable Satisfiable Modulo Theories which are solvable by many SMT solvers such as Z3[5]. The satisfiability of CLTLoc is PSPACE-complete [9] and the translation from MITL to CLTLoc in the worst case can be exponential [7]. One additional restriction over the MITL formulas is that the lower bound and upper bound for the intervals of MITL formulas should be integer in order to use the qtlsolver [7]. Therefore, we expect the values to be integer when we analyse MITL formulas.

B. Specification Debugging Results

We utilize the debugging algorithm on a set of specifications developed as part of a usability study for the evaluation of the VISPEC tool [21]. The usability study was conducted on two groups:

1) Non-expert users: These are users who declared that they have little to no experience in working with requirements. The non-expert cohort consists of twenty subjects from the academic community at Arizona State University. Most of the subjects have an engineering background.
2) Expert users: These are users who declared that they have experience working with system requirements. Note that they do not necessarily have experience in writing requirements using formal logics. The expert subject cohort was comprised of ten subjects from the industry in the Phoenix area.

Each subject received a task list to complete. The task list contained ten tasks related to automotive system specifications. Each task asked the subject to formalize a natural language specification through VISPEC and generate an STL specification. The task list is presented in Table II. Note that the specifications were preprocessed and transformed from the original STL formulas to MITL in order to run the debugging algorithm. For example, specification ϕ_3 in Table III originally in STL was $\phi_{3_{STL}} = \Diamond_{[0,40]}(((speed > 80) \Rightarrow \Diamond_{[0,20]}(rpm > 4000)) \wedge \Box_{[0,30]}(speed > 100))$. The STL predicate expressions $(speed > 80), (rpm > 4000), (speed > 100)$ are mapped into atomic propositions with non-overlapping predicates (Boolean functions) p_1, p_2, p_3. The predicates p_1, p_2, p_3 correspond to the following STL representations: $p_1 \equiv speed > 100$, $p_2 \equiv rpm > 4000$, and $p_3 \equiv 100 \geq speed > 80$.

In Table III, we present common issues with the developed specifications that our debugging algorithm would have detected and alerted each subject if the tool were available at the time of the study. Note that validity, redundancy and vacuity issues are present in the specifications listed. It should be noted that for specification ϕ_3, although finding the error takes a significant amount of time, our algorithm can be used off-line.

In Fig. 4, we present the runtime overhead of the three stage debugging algorithm over specifications collected in the usability study. In the first stage, 87 specifications go through validity checking. Five specifications fail the test and therefore they are immediately returned to the user. As a result, 82 specifications go through redundancy checking, where 9 fail the test. Lastly, 73 specifications go through vacuity checking where 5 specification have vacuity issue. The rest 68 specifications passed the tests. Note that in the figure, two outlier data points are omitted from the vacuity sub-figure for presentation purposes. The two cases were timed at 39,618sec and 17,421sec. In both cases, the runtime overhead was mainly because the zot software took hours to determine that the modified specification is unsatisfiable (both specifications where vacuous). The overall runtime of ϕ_3 in Table III is 39,645sec which includes the runtime of validity

[3] qtlsolver: A solver for checking satisfiability of Quantitative / Metric Interval Temporal Logic (MITL/QTL) over Reals. Available from https://code.google.com/p/qtlsolver/

[4] The zot bounded model/satisfiability checker. Available from https://code.google.com/p/zot/

[5] Microsoft Research, Z3: An efficient SMT solver. Available from http://research.microsoft.com/en-us/um/redmond/projects/z3/

and redundancy checking. The runtime overhead of vacuity checking of ϕ_3 (39,618sec) can be reduced by half because in vacuity checking we run MITL satisfiability checking for all literal occurrences. In particular, ϕ_3 has four literal occurrences where for two cases the zot took more than 19,500sec to determine that the modified specification is unsatisfiable. We can provide an option for early detection: as soon as an issue is found (just one unsatisfiable detection) the software should return the result which in ϕ_3 case can lead to half of the computation time of the original vacuity detection.

The blue circles in Fig. 4 represent the timing performance in each test categorized by literal occurrence and number of temporal operators. The red asterisks represent the mean values and the dashed line is the linear interpolation between them. In general, we observe an increase on the average computation time as the literal occurrence and number of temporal operators increases. Ideally, the performance analysis should be conducted over a large set of artificially generated benchmarks, i.e., specification formulas. However, developing such benchmarks is a challenging problem on itself and thus further research is required. The experimental results were extracted from an Intel Xeon X5647 (2.993GHz) machine with 12 GB RAM.

VII. Discussion

In the previous section, we mentioned that MITL satisfiability problem is a computationally hard problem. However, we know that LTL satisfiability is in practice solvable faster than MITL satisfiability [28]. We consider how we can use the satisfiability of LTL formulas to decide about the satisfiability of MITL formulas. Consider the following fragments of MITL and LTL in NNF:
MITL($\Box$): $\varphi ::= \top \mid \bot \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Box_I \varphi_1$
MITL($\Diamond$): $\varphi ::= \top \mid \bot \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Diamond_I \varphi_1$
LTL($\Box$): $\varphi ::= \top \mid \bot \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Box \varphi_1$
LTL($\Diamond$): $\varphi ::= \top \mid \bot \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Diamond \varphi_1$

In the Appendix (Section IX-B), we prove that the satisfaction of a formula $\phi_M \in$ MITL($\Diamond$) in NNF is related to the satisfaction of an LTL version of ϕ_M called $\phi_L \in$ LTL($\Diamond$) where ϕ_L is identical to ϕ_M except that the every interval I in ϕ_M is removed. For example, if $\phi_M = \Diamond_{[0,10]}(p \wedge q) \wedge \Diamond_{[0,10]} p$ then $\phi_L = \Diamond(p \wedge q) \wedge \Diamond p$. In essence, if ϕ_M is satisfiable, then ϕ_L is also satisfiable. Therefore, if ϕ_L is unsatisfiable then ϕ_M is also unsatisfiable.

For $\Box$ operator, satisfiability is dual of $\Diamond$. Assume $\phi'_M \in$ MITL($\Box$) contains only $\Box$ operator and $\phi'_L \in$ LTL($\Box$) is the LTL version of ϕ'_M. If ϕ'_L is satisfiable, ϕ'_M will also be satisfiable.

Based on the above discussion, if the specification ϕ_M that we intend to test/debug belongs to either category (Fragment) of MITL($\Diamond$) or MITL($\Box$), then we can check the satisfiability of its LTL version (ϕ_L) and decide accordingly:

- If $\phi_L \in$ LTL($\Diamond$) is unsatisfiable, then ϕ_M is unsatisfiable.

- If $\phi_L \in$ LTL($\Box$) is satisfiable, then ϕ_M is satisfiable.

In these two cases, we do not need to run MITL satisfiability. As a result LTL satisfiability checking is useful for validity testing. For redundancy check it may also be useful. For example, if we have a formula $\phi = \Diamond_{[0,10]} p \wedge \Box_{[0,20]} p$ we should check the satisfiability of $\phi' = \Box_{[0,10]} \neg p \wedge \Box_{[0,20]} p$ and $\phi'' = \Diamond_{[0,10]} p \wedge \Diamond_{[0,20]} \neg p$ for redundancy. Although in ϕ the original formula does not belong to either MITL($\Diamond$) or MITL($\Box$), its modified NNF version will fit in these categories and we may benefit by the LTL satisfiability for ϕ' and ϕ''. For vacuity checking, in rare occasions we may be able to use LTL satisfiability if after manipulating/simplifying the original specification and creating the NNF version, we can categorize the resulting formula into MITL($\Diamond$) or MITL($\Box$) fragments. We have currently not run any experiments using LTL satisfiability, but we plan to run feasibility studies in the future.

VIII. Conclusion and Future Work

We have presented a specification elicitation and debugging framework that helps expert and non-expert users to produce correct formal specifications. The debugging algorithm enables the detection of logical inconsistencies in MITL specifications. Our algorithm improves the elicitation process by providing feedback to the users on validity, redundancy and vacuity issues. The specification elicitation and debugging framework will be integrated in the ViSpec tool to simplify MITL specification development for verification of CPS. As future work, we will consider vacuity detection with respect to signals and systems. This will enable improved analysis since some issues can only be detected when considering both system and specification.

Acknowledgment

This work was partially supported by NSF awards IIP-1454143, CNS-1116136 and CNS-1350420.

References

[1] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus. Debugging temporal specifications with concept analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 182–195, 2003.

[2] Y. S. R. Annapureddy, C. Liu, G. E. 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*, volume 6605 of *LNCS*, pages 254–257. Springer, 2011.

[3] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Y. Vardi. Enhanced vacuity detection in linear temporal logic. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, pages 368–380, 2003.

[4] M. Autili, P. Inverardi, and P. Pelliccione. Graphical scenarios for specifying temporal properties: an automated approach. *Automated Software Engineering*, 14(3):293–340, 2007.

[5] D. L. Beatty and R. E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *DAC*, pages 596–602, 1994.

[6] S. Ben-David, D. Fisman, and S. Ruah. Temporal antecedent failure: Refining vacuity. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, pages 492–506, 2007.

[7] M. Bersani, M. Rossi, and P. San Pietro. A tool for deciding the satisfiability of continuous-time metric temporal logic. *Acta Informatica*, pages 1–36, 2015.

[8] M. M. Bersani, M. Rossi, and P. S. Pietro. A logical characterization of timed (non-)regular languages. In *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I*, pages 75–86, 2014.

TABLE II. TASK LIST WITH AUTOMOTIVE SYSTEM SPECIFICATIONS PRESENTED IN NATURAL LANGUAGE

Task	Natural Language Specification
1. Safety	In the first 40 seconds, vehicle speed should always be less than 160.
2. Reachability	In the first 30 seconds, vehicle speed should go over 120.
3. Stabilization	At some point in time in the first 30 seconds, vehicle speed will go over 100 and stay above for 20 seconds.
4. Oscillation	At every point in time in the first 40 seconds, vehicle speed will go over 100 in the next 10 seconds.
5. Oscillation	It is not the case that, for up to 40 seconds, the vehicle speed will go over 100 in every 10 second period.
6. Implication	If, within 40 seconds, vehicle speed is above 100 then within 30 seconds from time 0, engine speed should be over 3000.
7. Reactive Response	If, at some point in time in the first 40 seconds, vehicle speed goes over 80 then from that point on, for the next 30 seconds, engine speed should be over 4000.
8. Conjunction	In the first 40 seconds, vehicle speed should be less than 100 and engine speed should be under 4000.
9. Non-strict sequencing	At some point in time in the first 40 seconds, vehicle speed should go over 80 and then from that point on, for the next 30 seconds, engine speed should be over 4000.
10. Long sequence	If, at some point in time in the first 40 seconds, vehicle speed goes over 80 then from that point on, if within the next 20 seconds the engine speed goes over 4000, then, for the next 30 seconds, the vehicle speed should be over 100.

TABLE III. INCORRECT SPECIFICATIONS FROM THE USABILITY STUDY IN [21], ERROR REPORTED TO THE USER BY THE DEBUGGING ALGORITHM, AND ALGORITHM RUNTIME. FORMULAS HAVE BEEN TRANSLATED FROM STL TO MITL.

ϕ	Task (#) from Table II	MITL Specification	Reporting the errors	Runtime (Sec.)
ϕ_1	Stabilization (3)	$\Diamond_{[0,30]} p_1 \wedge \Diamond_{[0,20]} p_1$	$\Diamond_{[0,30]} p_1$ is redundant	14
ϕ_2	Stabilization (3)	$\Diamond_{[0,30]} (p_1 \Rightarrow \Box_{[0,20]} p_1)$	φ is a tautology	7
ϕ_3	Long sequence (10)	$\Diamond_{[0,40]} (((p_1 \vee p_3) \Rightarrow \Diamond_{[0,20]} p_2) \wedge \Box_{[0,30]} p_1)$	φ is vacuous: $\varphi \models \varphi[p_3 \leftarrow \perp]$	39645
ϕ_4	Oscillation (4)	$\Box_{[0,40]} p_1 \wedge \Box_{[0,40]} \Diamond_{[0,10]} p_1$	$\Box_{[0,40]} \Diamond_{[0,10]} p_1$ is redundant	29
ϕ_5	Long sequence (10)	$\Diamond_{[0,40]} (p_1 \vee p_3) \wedge \Diamond_{[0,40]} p_2 \wedge \Diamond_{[0,40]} \Box_{[0,30]} p_1$	$\Diamond_{[0,40]} (p_1 \vee p_3)$ is redundant	126

[9] M. M. Bersani, M. Rossi, and P. San Pietro. Deciding the satisfiability of mitl specifications. In Fourth International Symposium on *Games, Automata, Logics and Formal Verification,*, volume 119 of *EPTCS*, pages 64–78. Open Publishing Association, 2013.

[10] H. Chockler and O. Strichman. Before and after vacuity. *Form. Methods Syst. Des.*, 34(1):37–58, Feb. 2009.

[11] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings*, pages 52–67, 2008.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999.

[13] S. Demri and D. D'Souza. An automata-theoretic approach to constraint LTL. *Inf. Comput.*, 205(3):380–415, 2007.

[14] A. Donze. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Computer Aided Verification*, volume 6174 of *LNCS*, pages 167–170. Springer, 2010.

[15] R. Ehlers and V. Raman. Low-effort specification debugging and analysis. In *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014.*, pages 117–133, 2014.

[16] G. Fainekos, S. Sankaranarayanan, K. Ueda, and H. Yazarel. Verification of automotive control applications using s-taliro. In *Proceedings of the American Control Conference*, 2012.

[17] G. E. Fainekos. Revising temporal logic specifications for motion planning. In *IEEE International Conference on Robotics and Automation, ICRA 2011, Shanghai, China, 9-13 May 2011*, pages 40–45, 2011.

[18] D. Fisman, O. Kupferman, S. Sheinvald-Faragy, and M. Y. Vardi. A framework for inherent vacuity. In *Hardware and Software: Verification and Testing, 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27-30, 2008. Proceedings*, pages 7–22, 2008.

[19] A. Gurfinkel and M. Chechik. How vacuous is vacuous? In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 451–466, 2004.

[20] B. Hoxha, H. Bach, H. Abbas, A. Dokhanchi, Y. Kobayashi, and G. Fainekos. Towards formal specification visualization for testing and monitoring of cyber-physical systems. In *Int. Workshop on Design and Implementation of Formal Tools and Systems*. October 2014.

[21] B. Hoxha, N. Mavridis, and G. Fainekos. VISPEC: a graphical tool for easy elicitation of MTL requirements. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Hamburg, Germany*, September 2015.

[22] X. Jin, A. Donzé, J. Deshmukh, and S. A. Seshia. Mining requirements from closed-loop control models. In *Proceedings of the International Conference on Hybrid Systems: Computation and Control (HSCC)*, April 2013.

[23] K. Kim, G. E. Fainekos, and S. Sankaranarayanan. On the revision problem of specification automata. In *IEEE International Conference on Robotics and Automation, ICRA 2012, 14-18 May, 2012, St. Paul, Minnesota, USA*, pages 5171–5176, 2012.

[24] R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *STTT*, 15(5-6):563–583, 2013.

[25] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[26] O. Kupferman, W. Li, and S. A. Seshia. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, FMCAD '08, pages 25:1–25:9, Piscataway, NJ, USA, 2008. IEEE Press.

[27] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *STTT*, 4(2):224–233, 2003.

[28] J. Li, L. Zhang, G. Pu, M. Y. Vardi, and J. He. LTL satisfiability checking revisited. In *2013 20th International Symposium on Temporal Representation and Reasoning, Pensacola, FL, USA, September 26-28, 2013*, pages 91–98, 2013.

[29] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Proceedings of FORMATS-FTRTFT*, volume 3253 of *LNCS*, pages 152–166, 2004.

[30] V. Raman and H. Kress-Gazit. Analyzing unsynthesizable specifications for high-level robot behavior using ltlmop. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 663–668, 2011.

[31] H. Yang, B. Hoxha, and G. Fainekos. Querying parametric temporal logic properties on embedded systems. In *Testing Software and Systems*, pages 136–151. Springer, 2012.

[32] P. Zhang, B. Li, and L. Grunske. Timed property sequence chart. *Journal of Systems and Software*, 83(3):371–390, 2010.

IX. APPENDIX

A. Proof of Theorem 1

Proof: In order to show that φ_i is satisfied vacuously with respect to Φ, we must show that if $\Phi \models \varphi_i[l \leftarrow \perp]$, then the mutated specification is equivalent to the original specification. In other words, we should show that if $\Phi \models \varphi_i[l \leftarrow \perp]$, then

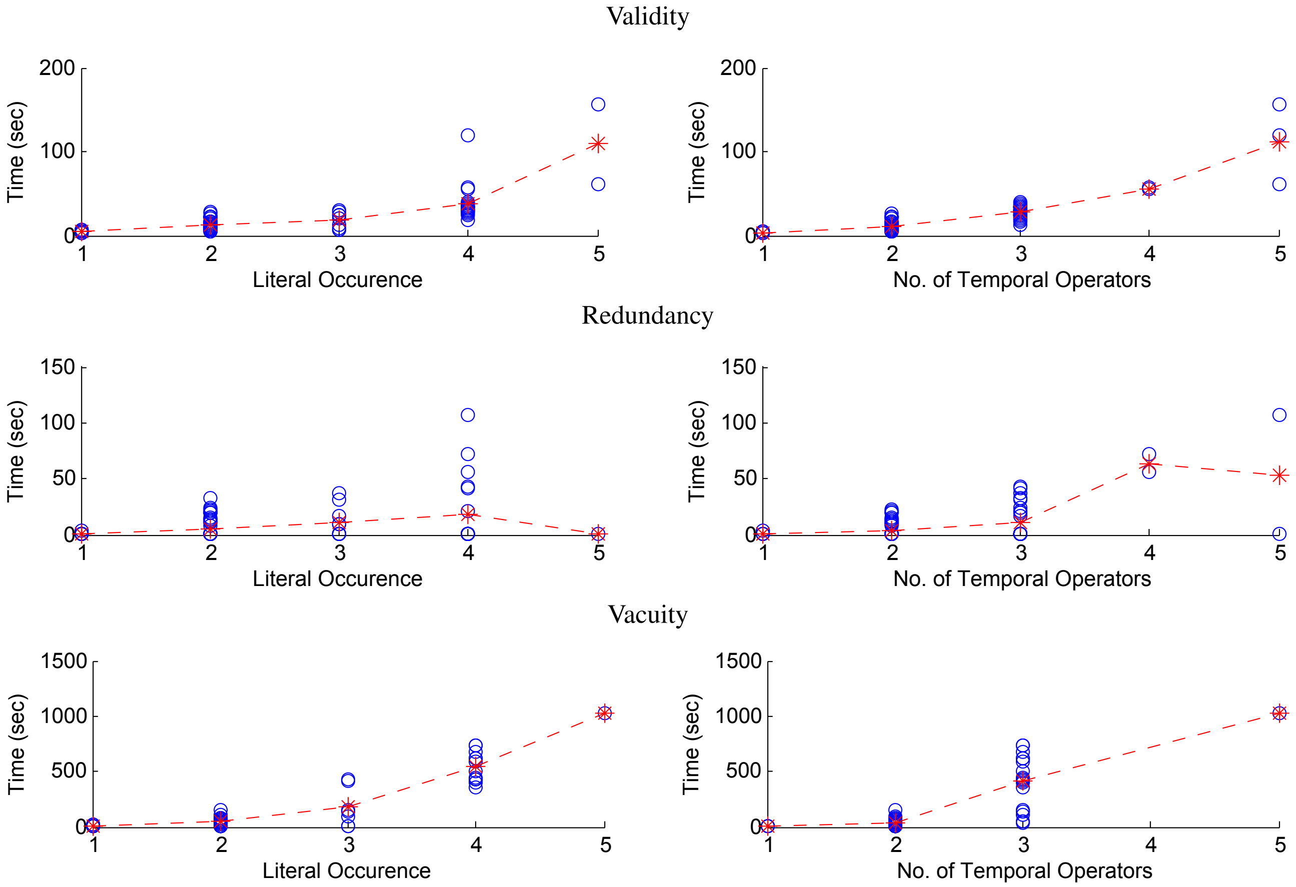

Fig. 4. Runtime overhead of the three stages of the debugging algorithm over user-submitted specifications. Timing results are presented over literal occurrence and number of temporal operators.

$(\{\Phi\backslash\varphi_i\} \cup \varphi_i[l \leftarrow\bot]) \equiv \Phi$. If the mutated specification is equivalent to the original specification, then the original specification is vacuously satisfiable in any system. That is, the specification is *inherently vacuous* [18], [10]. We already know that if $\Phi \models \varphi_i[l \leftarrow\bot]$, then $\Phi \implies \varphi_i[l \leftarrow\bot]$ and trivially $\Phi \implies \varphi_i[l \leftarrow\bot] \cup \{\Phi\backslash\varphi_i\}$. Now we just need to prove the other direction. We need to prove that when φ_i is in the negation normal form, then $\varphi_i[l \leftarrow\bot] \implies \varphi_i$. Since we replace only one specific literal occurrence of φ with $\bot$, the rest of the formula remains the same. Therefore, it should be noted that $\varphi_i[l \leftarrow\bot]$ does not modify any $l' \in litOccur(\varphi_i)$ where $l' \neq l$.
Base: Case $\varphi_i = l$ or $\varphi_i = l' \neq l$
We know that $\bot \implies l$ and $l' \implies l'$. Therefore $\varphi_i[l \leftarrow\bot] \implies \varphi_i$.
Induction Hypothesis: $\forall\varphi_j, \varphi_j[l \leftarrow\bot] \implies \varphi_j$
Induction Step: We will separate the case into unary and binary operators.
Before providing the cases we should review the *positively monotonic* operators [27]. According to MITL semantics, $f \in \{\Box_I, \Diamond_I\}$ and $g \in \{\wedge, \vee\}$ are positively monotonic, i.e. for every MITL formulas φ_1 and φ_2 in NNF with $\varphi_1 \implies \varphi_2$, we have $f(\varphi_1) \implies f(\varphi_2)$. Also, for all MITL formulas φ' in NNF, we have $g(\varphi_1, \varphi') \implies g(\varphi_2, \varphi')$ and $g(\varphi', \varphi_1) \implies g(\varphi', \varphi_2)$.
Case 1: $\varphi_i = f(\varphi_j)$ where $f \in \{\Box_I, \Diamond_I\}$. Since f is positively monotonic, we have that $\varphi_j[l \leftarrow\bot] \implies \varphi_j$ implies $f(\varphi_j[l \leftarrow\bot]) \implies f(\varphi_j)$. Thus,
$f(\varphi_j)[l \leftarrow\bot] = f(\varphi_j[l \leftarrow\bot]) \implies f(\varphi_j) = \varphi_i$. As a result $\varphi_i[l \leftarrow\bot] \implies \varphi_i$.
Case 2: $\varphi_i = g(\varphi_{j_1}, \varphi_{j_2})$ where $g \in \{\wedge, \vee\}$ Since g is positively monotonic, we have that $\varphi_{j_1}[l \leftarrow\bot] \implies \varphi_{j_1}$, and $\varphi_{j_2}[l \leftarrow\bot] \implies \varphi_{j2}$ implies
$g(\varphi_{j_1}[l \leftarrow\bot], \varphi_{j_2}[l \leftarrow\bot]) \implies g(\varphi_{j_1}, \varphi_{j_2})$. Thus, $g(\varphi_{j_1}, \varphi_{j_2})[l \leftarrow\bot] = g(\varphi_{j_1}[l \leftarrow\bot], \varphi_{j_2}[l \leftarrow\bot]) \implies g(\varphi_{j_1}, \varphi_{j_2}) = \varphi_i$. As a result $\varphi_i[l \leftarrow\bot] \implies \varphi_i$.

Since $\varphi_i[l \leftarrow\bot] \implies \varphi_i$ we can have:
$\{\Phi\backslash\varphi_i\} \cup \varphi_i[l \leftarrow\bot] \implies \{\Phi\backslash\varphi_i\} \cup \varphi_i$ which is equivalent to
$\{\Phi\backslash\varphi_i\} \cup \varphi_i[l \leftarrow\bot] \implies \Phi$ ■

B. Proofs of MITL fragments

We consider two MITL(◇,□) fragments, denoted MITL(□), and MITL(◇). In this proof we assume that all formulas are in NNF. We also consider LTL(◇,□) as the set of LTL formulas (with continuous semantics) that contains only ◇ and □ as temporal operators. In the following we provide the continuous semantics of LTL(◇,□) over traces with bounded duration. Semantics of LTL(◇,□) over bounded time traces can be defined as follows:

Definition 8 (LTL(◇,□) continuous semantics): Given a timed trace $\mu : [0, T] \to 2^{AP}$ and $t, t' \in \mathbb{R}$, and an LTL(◇,□) formula ϕ, the satisfaction relation $(\mu, t) \models \phi$ for temporal operators is inductively defined:

$$(\mu, t) \models \Diamond\phi_1 \text{ iff } \exists t' \in [t, T] \text{ s.t } (\mu, t') \models \phi_1.$$

$$(\mu, t) \models \Box\phi_1 \text{ iff } \forall t' \in [t, T],\ (\mu, t') \models \phi_1.$$

We will consider two LTL($\Diamond$,$\Box$) fragments denoted LTL($\Box$), and LTL($\Diamond$). The syntax of MITL and LTL fragments are as presented in Section VII.

We define the operator $[\phi]_{LTL}$ which can be applied to any MITL($\Diamond$,$\Box$) formula and removes its interval constraints to create a new formula in LTL($\Diamond$,$\Box$). For example if $\phi = \Diamond_{[0,10]}(p \wedge q) \wedge \Diamond_{[0,10]}p \wedge \Box_{[0,10]}q$, then $[\phi]_{LTL} = \Diamond(p \wedge q) \wedge \Diamond p \wedge \Box q$. As a result, for any $\phi \in$ MITL($\Diamond, \Box$) there exists a $\psi \in$ LTL($\Diamond, \Box$) where $\psi = [\phi]_{LTL}$. For each MITL($\Diamond, \Box$) formula ϕ, the language of ϕ denoted $L(\phi)$ is the set of all timed traces that satisfy ϕ: $\mu \models \phi$ iff $\mu \in L(\phi)$. Similarly, for any $\psi \in$ LTL($\Diamond, \Box$), the language of ψ denoted $L(\psi)$ is the set of all timed traces that satisfy ψ: $\mu' \models \psi$ iff $\mu' \in L(\psi)$.

Based on set theory, it is trivial to prove that $A \subseteq B$ and $C \subseteq D$ implies $A \cup C \subseteq B \cup D$ and $A \cap C \subseteq B \cap D$.

Theorem 2: For any formula $\varphi \in$ MITL($\Diamond$), and $t \in [0, T]$ we have $L_t(\varphi) \subseteq L_t([\varphi]_{LTL})$ where $L_t(\varphi) = \{\mu \mid (\mu, t) \models \varphi\}$. In other words for all timed trace μ we have $(\mu, t) \models \varphi$ implies $(\mu, t) \models [\varphi]_{LTL}$.

Proof: We use structural induction to prove that $L_t(\varphi) \subseteq L_t([\varphi]_{LTL})$
Base: if $\varphi = \top, \bot, p, \neg p$, then $[\varphi]_{LTL} = \varphi$ and $L_t(\varphi) \subseteq L_t([\varphi]_{LTL})$
Induction Hypothesis: We assume that there exist $\varphi_1, \varphi_2 \in$ MITL($\Diamond$) where for all $t \in [0, T]$, $L_t(\varphi_1) \subseteq L_t([\varphi_1]_{LTL})$ and $L_t(\varphi_2) \subseteq L_t([\varphi_2]_{LTL})$

Case 1: For Binary operators $\wedge, \vee$ we can use the union and intersection properties. In essence, for all formulas φ_1, φ_2 we have $L_t(\varphi_1 \vee \varphi_2) = L_t(\varphi_1) \cup L_t(\varphi_2)$ and $L_t(\varphi_1 \wedge \varphi_2) = L_t(\varphi_1) \cap L_t(\varphi_2)$. According to the IH $L_t(\varphi_1) \subseteq L_t([\varphi_1]_{LTL})$ and $L_t(\varphi_2) \subseteq L_t([\varphi_2]_{LTL})$; therefore, $L_t(\varphi_1) \cap L_t(\varphi_2) \subseteq L_t([\varphi_1]_{LTL}) \cap L_t([\varphi_2]_{LTL})$ and $L_t(\varphi_1) \cup L_t(\varphi_2) \subseteq L_t([\varphi_1]_{LTL}) \cup L_t([\varphi_2]_{LTL})$. As a result, $L_t(\varphi_1 \wedge \varphi_2) \subseteq L_t([\varphi_1]_{LTL} \wedge [\varphi_2]_{LTL}) = L_t([\varphi_1 \wedge \varphi_2]_{LTL})$, and $L_t(\varphi_1 \vee \varphi_2) \subseteq L_t([\varphi_1]_{LTL} \vee [\varphi_2]_{LTL}) = L_t([\varphi_1 \vee \varphi_2]_{LTL})$.

Case 2: For the temporal operator $\Diamond$, we need to compare the semantics of MITL($\Diamond$) and LTL($\Diamond$). Recall that

$$(\mu, t) \models \Diamond_I\varphi_1 \text{ iff } \exists t' \in (t + I) \cap [0, T] \text{ s.t } (\mu, t') \models \varphi_1.$$

$$(\mu, t) \models \Diamond\varphi_1 \text{ iff } \exists t' \in [t, T] \text{ s.t } (\mu, t') \models \varphi_1.$$

Recall that $t'' \in (t + I) \cap [0, T]$ implies $t'' \in [t, T]$ since the left bound of I is nonnegative.

According to the semantics, $\forall \mu.(\mu, t) \models \Diamond_I\varphi_1$ implies
$\exists t' \in (t + I) \cap [0, T]$ s.t $(\mu, t') \models \varphi_1$ implies
$\exists t' \in (t + I) \cap [0, T]$ s.t $(\mu, t') \models [\varphi_1]_{LTL}$ according to IH ($L_{t'}(\varphi_1) \subseteq L_{t'}([\varphi_1]_{LTL})$).
If $\exists t' \in (t + I) \cap [0, T]$ s.t $(\mu, t') \models [\varphi_1]_{LTL}$ then
$\exists t' \in [t, T]$ s.t $(\mu, t') \models [\varphi_1]_{LTL}$ since $t' \in (t + I) \cap [0, T]$ implies $t' \in [t, T]$.

Moreover, $(\mu, t') \models [\varphi_1]_{LTL}$ implies that $(\mu, t) \models \Diamond[\varphi_1]_{LTL} \equiv [\Diamond\varphi_1]_{LTL}$.
As a result, $\forall \mu.\ (\mu, t) \models \Diamond_I\varphi_1 \implies (\mu, t) \models [\Diamond\varphi_1]_{LTL}$ so $L_t(\Diamond_I\varphi_1) \subseteq L_t([\Diamond\varphi_1]_{LTL})$. ■

If $\varphi \in$ MITL($\Diamond$) then $\overline{L_t([\varphi]_{LTL})} \subseteq \overline{L_t(\varphi)}$ (immediate from set theory). Thus, for all timed traces μ, $\mu \not\models [\varphi]_{LTL}$ implies that $\mu \not\models \varphi$.

Corollary 1: For any $\varphi \in$ MITL($\Diamond$), if $[\varphi]_{LTL} \in$ LTL($\Diamond$) is unsatisfiable, then φ is unsatisfiable.

Theorem 3: For any formula $\varphi \in$ MITL($\Box$), and $t \in [0, T]$, we have $L_t([\varphi]_{LTL}) \subseteq L_t(\varphi)$, where $L_t(\varphi) = \{\mu | (\mu, t) \models \varphi\}$. In other words $\forall \mu(\mu, t) \models [\varphi]_{LTL} \implies (\mu, t) \models \varphi$

Proof: Similar to Theorem 2, we can apply structural induction for the proof of Theorem 3. ■