This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3085911, IEEE Access

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Digital Object Identifier

SMT-Based Consistency Checking of Configuration-Based Components Specifications

LAURA PANDOLFO[1],(Member, IEEE), LUCA PULINA[1],(Member, IEEE), and SIMONE VUOTTO[1]
[1]Intelligent system DEsign and Applications (IDEA) Lab, University of Sassari, via Muroni 23A, 07100 Sassari, Italy
(e-mail: lpandolfo@uniss.it, lpulina@uniss.it, svuotto@uniss.it)
Corresponding author: Laura Pandolfo (e-mail: lpandolfo@uniss.it).

ABSTRACT Cyber-Physical Systems (CPSs) are engineered systems that are built from, and depend upon, the seamless integration of computational algorithms and physical components. CPSs are widely used in many safety-critical domains, making it crucial to ensure that they operate safely without causing harm to people and the environment. Therefore, their design should be robust enough to deal with unexpected conditions and flexible to answer to the high scalability and complexity of systems. Nowadays, it is well-established that formal verification has a great potential in reinforcing safety of critical systems, but nevertheless its application in the development of industrial products may still be a challenging activity. In this paper, we describe an approach based on Satisfiability Modulo Theories (SMT) to formally verify, at the design stage, the consistency of the system design – expressed in a given domain-specific language, called QRML, which is specifically designed for CPSs – with respect to some given property constraints, with the purpose to reduce inconsistencies during the system development process. To this end, we propose an SMT-based approach for checking the consistency of configuration based-components specifications and we report the results of the experimental analysis using three different state-of-the-art SMT solvers. The main goal of the experimental analysis is to test the scalability of the selected SMT solvers and thus to determine which SMT solver is the best in checking the satisfiability of the properties.

INDEX TERMS Design verification, Application of formal methods, Satisfiability Modulo Theories

I. INTRODUCTION

Cyber-Physical Systems (CPSs) are real-time embedded systems in which the software controllers continuously interact with physical environments, possibly with humans in the loop. These systems are often distributed with sensors and actuators, which monitor and control physical processes, usually with feedback loops where physical processes affect computations and vice-versa [1]. Recently, CPSs are gathering momentum and attracting massive attention from the research communities and large investment from industry [2]. The emerging applications of CPSs can be found in a number of large-scale and safety-critical domains, making it crucial to ensure that they operate safely without causing harm to people and the environment. Application areas include healthcare, automotive, manufacturing, industry automation, and critical infrastructure such as, electric power, energy, and water resources, so CPSs design should be robust enough to deal with unexpected conditions and, at the same time, flexible to answer to the high scalability and complexity of systems. Due to the critical nature of their applications and the tight time-to-market constraints, the verification of the CPSs design becomes an important issue in order to ensure the correctness of these systems.

Since the 90's, formal methods have been exploited to improve complex automation systems (see, e.g., [3]), and currently they are often introduced in the verification of CPSs' applications. Nowadays, it is well-established that the usage of formal methods has a great potential in reinforcing safety in the design of (critical) CPSs – see, e.g., the results obtained in the context of the CERBERO EU H2020 project [4], [5] about the application of formal methods in the design of CPSs [6]–[8].

In this paper, we describe an approach based on Satisfiability Modulo Theories (SMT) [9] to formally verify, at the design stage, the consistency of the system design – expressed in a given Domain Specific Language (DSL) – with respect to some given property constraints, with the purpose to reduce inconsistencies during the system development process.

Such research has been developed in the context of the ECSEL project entitled "From the cloud to the edge smart IntegraTion and OPtimisation Technologies for highly efficient Image and VIdeo processing Systems" (`FitOptiVis` [10]) [11], which involves 30 partners from industry and academia. The main objective of `FitOptiVis` is to develop an integral approach for smart integration of image and video processing pipelines for CPSs covering a reference architecture, supported by low-power, high-performance, smart devices, and by methods and tools for combined design-time and run-time multi-objective optimisation within system and environment constraints. Thus, a DSL for the design of CPSs, namely the Quality and Resource Management [12] (QRML) has been developed by the authors of [13], [14]. It is based on an interface-modeling framework, which eases the dynamic reconfiguration and multi-objective optimization of component-based systems for quality and resource management purposes.

Checking the consistency of configuration-based components specifications at design-time is an important task, in order to formally ensure their correctness and satisfaction, to avoid manual review which is time-consuming and error-prone. Moreover, it is also crucial for reducing time-to-market window in industrial applications use cases within the project. In order to cope with this task, we present our SMT-based approach implemented into a tool able to check the consistency of configuration based-components design expressed in QRML with the purpose to formally check by means of an SMT solver whether the configurations guarantee to satisfy all the properties.

The core of our approach is based on an SMT encoding, where system components expressed in QRML are translated into an instance of a satisfiability problem. In order to evaluate the effectiveness of the proposed SMT approach, we have developed an automated generator of DSL specifications and employed three different state-of-the-art SMT solvers to check the satisfiability of the translated SMT properties. The purpose of the experimental analysis here reported is mainly to test the scalability of the selected SMT solvers and thus to determine which SMT solver is the best in checking the satisfiability of the properties. As it will be shown later in the paper, we demonstrate the effectiveness of the proposed SMT -based approach to verify configuration-based components design of various sizes within a reasonable time.

The rest of the paper is organized as follows. Section II provides some background of SMT as well as an overview of QRML. In Section III we briefly report about the related work. The process of consistency checking is presented in Section IV. In Section V we provide details about the instances and SMT solvers involved in the experimental analysis, which results and related discussion are reported in Section VI. We conclude the paper in Section VII with some final remarks.

II. BACKGROUND

In this section, we introduce some concepts and terminology that will be used in the rest of the paper. In particular, we provide a big-picture overview of SMT, followed by a description of the key aspects of the formal interface-modeling framework for Quality and Resource Management as well as the language that derives from it.

A. SATISFIABILITY MODULO THEORIES

SMT is the problem of deciding the satisfiability of a first-order formula with respect to some decidable theory $\mathcal{T}$, while an SMT instance is a formula in first-order logic where some function and predicate symbols have additional interpretations.

Given a first-order formula ϕ in a decidable background theory $\mathcal{T}$, SMT problem consists in deciding whether there exists a model, namely an assignment to the free variables in ϕ, that satisfies ϕ. SMT generalizes the Boolean satisfiability problem (SAT) by adding background theories such as the theory of real numbers, the theory of integers, and the theories of data structures. For example, a formula can contain clauses like $p \vee q \vee (x + 2 \leq y) \vee (x > y + z)$, where p and q are Boolean variables and x, y and z are integer variables. Predicates over non-Boolean variables, such as linear integer inequalities, are evaluated according to the rules of a background theory. In this respect, there exist several theories of practical interests, such as the quantifier-free linear integer arithmetic (QF_LIA), where atoms are linear inequalities over integer variables, the quantifier-free non-linear integer arithmetic (QF_NIA), where atoms are polynomial inequalities over integer variables, and the quantifier-free linear real arithmetic (QF_LRA), which is similar to QF_LIA but with real variables.

The current general library for SMT is the Satisfiability Modulo Theories Library (SMT-LIB) [15], which provides standard descriptions of background theories used in SMT systems, as well as collecting and making available a large library of benchmarks for SMT solvers. An SMT solver is a decision procedure which solves the satisfiability problem, which is the problem – given a propositional formula – to determine whether it is satisfiable or not. Given an unsatisfiable SAT formula φ, a subset of clauses φ_C (i.e. $\varphi_C \subseteq \varphi$) whose conjunction is still unsatisfiable is called an *unsatisfiable core* of the original formula. Modern SAT solvers can be instructed to generate an unsatisfiable core [16]. Current state-of-the-art SMT solvers use the so-called *lazy approach*, which consists on the integration of a SAT solver and a $\mathcal{T}$-solver, i.e., a decision procedure for the given theory $\mathcal{T}$. In order to decide the satisfiability of an input formula ϕ, the SAT solver enumerates the truth assignments to the Boolean abstraction of ϕ, while the $\mathcal{T}$ solver is invoked when the SAT solver finds a satisfying assignment for the Boolean abstraction in order to check whether the current Boolean

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3085911, IEEE Access

assignment is consistent in the theory. If the conjunction is satisfiable, then a satisfying solution (*model*) is found for the input formula ϕ. Otherwise, the $\mathcal{T}$-solver returns an explanation for the conflict which identifies a reason for the unsatisfiability. Then, the conflict explanation is learned by the SAT solver in order to prune the search until either a theory-consistent Boolean assignment is found, or no more Boolean satisfying assignments exist. For a comprehensive background on SMT, please refer to [9], [17].

B. THE QUALITY AND RESOURCE MANAGEMENT LANGUAGE

In [13], the authors present a formal interface-modeling framework for quality and resource management that provides an abstract description of hardware and software components in terms of resources, quality indicators and working configurations. From the Quality Resource Management (QRM) framework derives the related language which aims to support the configuration component-based design of CPSs. In general, a DSL is a language designed to be useful for describing a limited set of tasks in a specific domain, in contrast to general-purpose languages that are supposed to be useful for more generic tasks in crossing multiple application domains [18]. DSLs usually have a concrete syntax and an implicit or explicit semantics.

In the following we report some of the definitions given in [13] that will be useful for the understanding of the subsequent sections.

A configuration c is a set of parameters that capture the configurable working points of the component. In particular, the parameters are represented by an *input*, an *output*, a *required budget*, a *provided budget*, and a *quality*.

Definition 1 (Configuration Space): A configuration space $\mathcal{S}$ is the Cartesian product $Q_1 \times Q_2 \times \cdots \times Q_n$ of a finite number of partially-ordered sets (posets). A poset is a set Q with a partial-order relation $\preceq_Q$. A configuration c is an element of a configuration space $c \in \mathcal{S}$. We define $C \subseteq \mathcal{S}$ to be the set of possible configurations for a given component.

Definition 2 (Free Product): Let S_1 and S_2 be configuration spaces, let $C_1 \subseteq S_1$ and let $C_2 \subseteq S_2$. The free product of C_1 and C_2 is the Cartesian product $C_1 \times C_2$ in the configuration space $S_1 \times S_2$.

Definition 3 (QRM Interface): The QRM interface of a component is a set of configurations from a six-dimensional configuration space $\mathcal{S} \subseteq Q_i \times Q_o \times Q_r \times Q_p \times Q_q \times Q_x$ where:

- Q_i models the inputs
- Q_o models the outputs
- Q_r models the required budget
- Q_p models the provided budget
- Q_q models the quality
- Q_x models the parameters

In the framework, input and required budget specifications capture the requirements of the component, while output, provided budget and quality capture their promises.

The difference between qualities and parameters is that the former is used by the quality and resource manager for optimization, the latter is used by external actors, e.g., the user, to control the selection of subsets of configurations. In the following, we consider qualities and parameters to be integer values and, when not otherwise specified, we use the name *properties* to refer to both of them. Moreover, with a slight abuse of notation, we use C (with $C \subseteq \mathcal{S}$) to refer to the component defined by that configuration space. Finally, the elements of Q_i and Q_o are typed *channel* objects and the elements of Q_r and Q_p are typed *budget* objects. *Channels* model data that the components require in input or provide in output, while *budgets* model resources or services that the component provides or requires. For example, a camera can have a power source as required budget and a video stream as output channel. The distinction of *channels* and *budgets* also makes it impossible to connect the output of a component with a required budget and vice versa. A new type is defined with a unique name, a list of properties and a set of constraints on such properties. When we refer to a generic element, either channel or budget, we use the term *interface element*.

In addition, the framework describes three different kinds of components composition: free, horizontal and vertical.

Definition 4 (Free Composition): Given two components C_1 and C_2, the free composition is computed by first applying the free product $C_1 \times C_2$ and then by applying a grouping derivation for each of the six dimensions of the QRM interface. A grouping derivation puts two posets Q_a and Q_b in a single new multi-dimensional poset $Q_{a,b}$ where $(p_a, p_b) \preceq_{Q_{a,b}} (q_a, q_b)$ iff $p_a \preceq_{Q_a} q_a$ and $p_b \preceq_{Q_b} q_b$. The result is a new component $C_{comp} \subseteq Q_i \times Q_o \times Q_r \times Q_p \times Q_q \times Q_x$ composed of all the possible combinations of the configurations in C_1 and C_2.

Using Definition 4, we can construct a component C made by many sub-components. We define $subcomp(C)$ to be a set of such sub-components.

Definition 5 (Horizontal Composition): The horizontal composition of two components C_1 and C_2 is similar to the free composition, but the set of configurations is constrained to the ones for which the output of the former component matches with the input of the latter. Moreover, the resulting component does not provide the output and does not require the satisfied input anymore.

Definition 6 (Vertical Composition): The vertical composition of two components C_1 and C_2 is similar to the free composition, but the set of configurations is constrained to the ones for which the provided budget of the former component matches with the required budget of the latter. Moreover, the resulting component does not provide the budget and does not require the satisfied budget anymore.

Figure 1 shows an example of a system expressed using QRML [14], a domain-specific language that implements the QRM interface. Looking at the figure, we can see the definition of component `SmartCamera`, composed of the components `Camera` and `CPU` connected together, and it

```
system S {
 component SmartCamera sm;
}

 component SmartCamera {
  configuration low_frequency {
  component Camera camera;
  component CPU cpu;
  quality power;
  camera.comp runs on cpu.comp;
  cpu.comp.frequency = 100;
  camera.framerate = 30;
  power = 10;
  }

  configuration high_frequency {
  component Camera camera;
  component CPU cpu;
  quality power;
  camera.comp runs on cpu.comp;
  cpu.comp.frequency = 200;
  camera.framerate = 60;
  power = 100;
  }
 }

 component Camera {
  outputs VideoStream video;
  requires ComputationalCapability comp;
  property framerate;
 }

 component CPU {
  supports ComputationalCapability comp;
 }

 budget ComputationalCapability {
  property frequency;
 }

 channel VideoStream { }
```

FIGURE 1. Example of system expressed in QRML.

is available in two configurations: `low_frequency` and `high_frequency`. In the first configuration, the framerate of the camera is set to 30 and the frequency of the computational capability provided by the `CPU` is 100, while in the second configurations the values are 60 and 200, respectively. Moreover, `SmartCamera` makes available also the `quality power` that can be used at runtime to choose which configuration to run. Notice also that `Camera` and `CPU` do not have the `configuration` because they only implement a single default configuration. The specification also defines a budget (`ComputationalCapability`) and a channel (`VideoStream`). Finally, at the system level, only an instance of `SmartCamera` is instantiated, but the same component declaration can be instantiated multiple times with different values.

To make a parallel with the QRM framework described before, given a configuration $c = (Q_i, Q_o, Q_r, Q_p, Q_q, Q_x)$ let us suppose to have three possible configurations for the `CPU` component

$$c_1 = (\bot, \bot, \bot, CC(100), \bot, \bot)$$
$$c_2 = (\bot, \bot, \bot, CC(150), \bot, \bot)$$
$$c_3 = (\bot, \bot, \bot, CC(200), \bot, \bot)$$

and two possible configurations for the `Camera`

$$c_4 = (\bot, VS(30), CC(100), \bot, \bot, \bot)$$
$$c_5 = (\bot, VS(60), CC(200), \bot, \bot, \bot)$$

where $\bot$ is the void poset, while `CC` and `VS` are shorthands for the `ComputationalCapability` budget and the `VideoStream` channel, respectively. For example, `VS`(30) is a `VideoStream` channel with a framerate of 30. The free composition of the two components is simply:

$$c_{1,4} = (\bot, VS(30), CC(100), CC(100), \bot, \bot)$$
$$c_{1,5} = (\bot, VS(60), CC(200), CC(100), \bot, \bot)$$
$$c_{2,4} = (\bot, VS(30), CC(100), CC(150), \bot, \bot)$$
$$c_{2,5} = (\bot, VS(60), CC(200), CC(150), \bot, \bot)$$
$$c_{3,4} = (\bot, VS(30), CC(100), CC(200), \bot, \bot)$$
$$c_{3,5} = (\bot, VS(60), CC(200), CC(200), \bot, \bot).$$

However, if we apply a vertical composition, as in the example in Figure 1, not all of these configurations are feasible. Moreover, we have to remove the required and provided budgets, because they cancel each other out. Therefore, the result of the vertical composition is:

$$c_{1,4} = (\bot, VS(30), \bot, \bot, \bot)$$
$$c_{3,5} = (\bot, VS(60), \bot, \bot, \bot).$$

III. RELATED WORK

In this section, we report on the most relevant and recent contributions in the field. Considering that our approach has the QRM framework as a starting point, the purpose of this section is not to compare our approach to others but rather to present some important related works in order to increase awareness about the context in which our contribution can be collocated in the scientific literature.

Several approaches that investigate ways to provide early fault detection when developing safety-critical industrial systems using DSLs have been proposed in the scientific literature. For instance, in [19], the authors have extended a DSL tailored to model heterogeneous robots swarm with a denotational semantics that supports both automatic and semi-automatic verification in the form of model checking and theorem proving. Several works in this field have often used logic solver approaches, namely mapping a model generation problem into a logic problem, which is usually solved by SMT or SAT solvers. Some techniques that validate a wide range of properties related to the semantics of the DSL have been proposed in [20], [21], where the authors translated the DSL instances and properties into SMT problems, and then the model is analysed using the SMT solver Z3. If the property does not hold, delta debugging is used to identify the rules in the DSL instance that contribute to the failure.

Complete frameworks with standalone specification languages have been presented in [22] and in [23], which use the SMT solver Z3 and the SAT solver Sat4j [24], respectively. One more complete tool is presented in [25], where the authors describe Clafer, a class modeling language with first-class support for feature modeling. The language is supported by tools for model analysis, which is performed by translating the models into the input language of the underlying back-end solvers Alloy [23], Z3, and Choco 3 [26]. Finally, another interesting work is presented in [27], where the authors propose an automated validation framework to formally check the specification of DSLs in the avionics domain by using Z3 and Alloy as back-end reasoners.

IV. CONSISTENCY CHECKING

The goal of this section is to describe the process of checking the consistency of a given specification. The consistency checking aims to automatically verifying if each component of the system can be instantiated in at least one configuration, and if there exists a feasible assignment for each property satisfying all constraints. To achieve this goal, we encode a system described in QRML into an instance of a satisfiability problem in Quantifier Free Linear Arithmetic over Integers (QF_LIA in [15]).

As illustrated in Section II, a component C is defined by the QRM interface, namely a set of configurations in which such component can be instantiated. As shown in the example reported in Figure 1, in QRML a configuration is described by a set of statements defining:

- the elements composing the interface of the component: inputs, outputs, required budgets, provided budgets, quality and properties;
- the sub-components $subcomp(C)$ that are part of C and their connections;
- a set of arithmetic constraints that limit the possible values of properties of the component, its sub-components and the elements composing its interface. We call $exp(c)$ the list of such constraints.

The first step of the process consists in sorting the components in topological order, considering the sub-components as dependencies. In the example in Figure 1 a possible topological order could be [`CPU`, `Camera`, `SmartCamera`, `S`]. If the topological sort fails, it means that there is a circular dependency and the process stops.

If there are no circular dependencies, each component is built in the given order, namely all its dependencies are resolved and replaced by their definitions. If an element is not defined or has a wrong type (e.g., a channel is defined as required budget), an error is reported.

If the building step succeeds, the resulting model is a tree composed of components, configurations, channels, budgets, qualities and properties. We define a function $id(.)$ which assigns an identifier to each element of the tree, which will be useful during the encoding stage.

The next step is the binding of the components interface in order to achieve a horizontal or vertical composition. For the horizontal composition, we consider the inputs $i \in Q_i$ and outputs $o \in Q_o$ involved in the composition and we constraint their id (and the ids of their properties) to be the same, i.e., $id(i) = id(o)$. In this way the constraints applied to them actually refer to the same object. If there is a type mismatched an error is returned. In a similar way, the same is done for the vertical composition with budgets $r \in Q_r$ and $p \in Q_p$. Figure 2 shows the result of the whole process considering the example in Figure 1 as input.

Given the tree obtained by the previous steps, the encoding procedure works as follows. First, for each node of the tree a new variable is created, using the id as variable name: properties and qualities are defined as integer variables, while the remaining ones are defined as Boolean variables. The latter variables are used to indicate if a particular component, configuration or interface element is instantiated or not.

Secondly, starting from the root component C, we build the following assertion:

$$id(C) \rightarrow \bigvee_{c \in C} id(c) \tag{1}$$

and for each configuration $c \in C$:

$$id(c) \rightarrow \Big(\bigwedge_{e \in exp(c)} e \Big) \wedge \Big(\bigwedge_{x \in subcomp(C)} id(x) \Big) \tag{2}$$

Assertion (1) states that if a component is instantiated, then at least one of the configurations must be true. In practice, only one of them should be true at a given time, but since we are only interested in checking if *at least one* configuration is feasible, the disjunctive clause is sufficient. Assertion (2) states that if a particular configuration c is selected, then the constraints $exp(c)$ declared in c must hold and the subcomponents $subcomp(C)$ defined into the configuration must exist.

Regarding the horizontal composition, let consider a constraint that connects an output o, identified by $id(o)$, with an input i identified by $id(i)$. We have already seen that having $id(o) = id(i) = id^*$ forces the encoding to apply the constraints on both i and o, on the same object, as the semantics of the language implies. However, the binding between i and o also says something else, that is not enforced yet: for a connection between i and o to hold, i and o must exist in some configuration. For example, if we want to connect the output o of a sub-component C_1, and this component can exist in three possible configurations, but only one of them expose the output o, we want to restrict the configuration space of C_1 to only that specific configuration. Moreover, we can extend this idea to sub-sub-components and so on, involving more levels of the tree described before. Therefore, defining $input(c)$ and $output(c)$ as the sets of all possible inputs and outputs, respectively, that are part of the sub-tree originating from c and defining $path(c, n)$ as the set containing the configurations crossed to reach the node n of the same sub-tree, we have the Assertion (3) for each

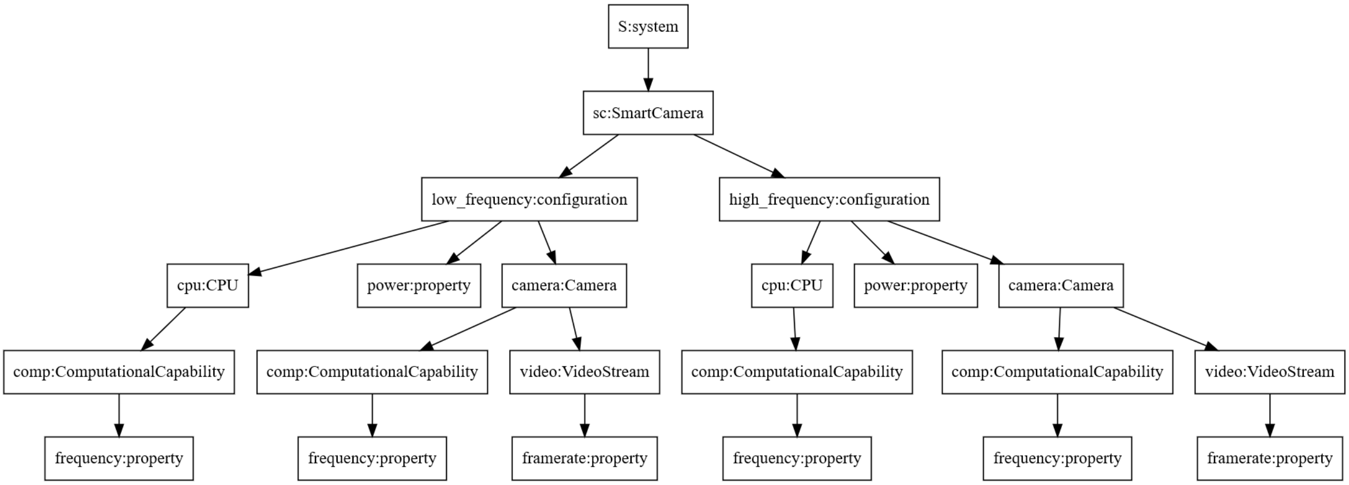

FIGURE 2. Tree Structure of the specification.

configuration c and each horizontal composition between i and o, such that $id(i) = id^*$ and $id(o) = id^*$.

$$id(c) \rightarrow \Big(\bigvee_{\substack{i \in input(c) \\ id(i)=id^*}} \bigwedge_{c_i \in path(c,i)} id(c_i) \Big) \wedge \Big(\bigvee_{\substack{o \in output(c) \\ id(i)=id^*}} \bigwedge_{c_o \in path(c,o)} id(c_o) \Big) \qquad (3)$$

The vertical composition works in a similar way, but we have required budgets instead of inputs, and provided budgets rather than outputs.

We recursively build assertions (1), (2) and (3) for each sub-component $x \in subcomp(C)$. The procedure terminates because we previously checked that there is no circular dependency. Finally, we have to assert that the root $id(C)$ must be true, otherwise the formula is trivially satisfied by setting all boolean variables to false.

Following the steps described above, the specification shown in Figure 1 is translated into the set of SMT constraints depicted in Figure 3.

The tool implementing the translation here described is available for download at [34].

```
(set-logic QF_LIA)

(declare-const S.sc.camera.framerate Int)
(declare-const S.sc.power Int)
(declare-const _b4.frequency Int)
(declare-const S Bool)
(declare-const S.sc Bool)
(declare-const S.sc.low_frequency Bool)
(declare-const S.sc.high_frequency Bool)
(declare-const S.sc.cpu Bool)
(declare-const S.sc.camera Bool)

(assert  (=> S S.sc))

(assert  (=> S.sc (or S.sc.low_frequency
    S.sc.high_frequency)))

(assert  (=> S.sc.low_frequency
    (and (= _b4.frequency 100)
    (and (= S.sc.camera.framerate 30)
    (and (= S.sc.power 10)
    (and S.sc.cpu S.sc.camera))))))

    (assert  (=> S.sc.high_frequency
    (and (= _b4.frequency 200)
    (and (= S.sc.camera.framerate 60)
    (and (= S.sc.power 100)
    (and S.sc.cpu S.sc.camera))))))

(assert S)

(check-sat)
```

FIGURE 3. SMT encoding example.

V. INSTANCES AND SOLVERS

In order to evaluate the scalability of a set of SMT solvers, we have implemented an automated generator (available at [34]) of QRML specifications and employed three state-of-the-art SMT solvers to check the satisfiability of the translated SMT constraints.

The generator can produce a specification according to a large series of parameters, such:

- components: the number of components C to define;
- interface_elements: the number of channels (Q_i, Q_o) or budgets (Q_r, Q_p) to define (randomly picked according to a uniform distribution);
- properties: the number of properties to define for each component, channel and budget;
- configs: the number of configurations $c \in \mathcal{S}$ per component;
- depth (δ): the depth of component/sub-component hierarchy;

- subcomps: the number of sub-components per component ($|subcomp(C)|$), except for components at depth 0, which do not have any further dependency;
- expressions: the number of expressions (randomly generated, considering a set of arithmetic operators and the set of accessible properties) for each configuration;
- interface_elements_per_component: the number of channels and/or budgets composing the interface of each component. The type and direction (i.e., if they are required or provided) is chosen randomly;
- connection_rate: the rate of generated feasible connections between compatible inputs and outputs or required and provided budgets.

The generator works as follows: first, it generates the specified number of interface elements, randomly choosing the type (channel or budget) and assigning unique names to them. For each interface element, a number properties of properties is generated, assigning a name to each of them. It starts generating $\lceil \frac{components}{\delta} \rceil$ components with $\delta = 1$, which do not have any subcomponent, then $\lfloor \frac{components}{\delta} \rfloor$ components with $\delta = 2$ and so on, so that the depths of components are evenly distributed from 1 to δ. For example, if the generation starts with components = 7 and δ = 3, the generator will produce 3 components with $\delta = 1$, 2 components with $\delta = 2$ and 2 components with $\delta = 3$. For components with $\delta > 1$ the subcomponents are computed as follows: 1 component is randomly chosen in the set of components with $(\delta - 1)$, while all the other (subcomp - 1) subcomponents are randomly chosen from the components with depth ranging from 1 to $(\delta - 1)$. Moreover, for each component C:

- a number properties of properties is generated;
- a number interface_elements_per_component of interface elements are randomly selected from the set of budgets and channels previously generated;
- a number config of configurations is generated.

Notice that the subcomponents and properties are defined in all components configurations, while the interface elements are evenly distributed among the defined configurations. At this point, for each configuration $c_i \in C$ the generator:

- generates a number expressions of expressions, randomly combining some constraints, arithmetic operators and accessible properties (i.e., the properties of the component, its subcomponents and of the defined interface elements);
- computes all possible feasible connections between accessible interface elements (e.g. if a subcomponent C_1 provides a budget of type X and another subcomponent C_2 requires the same type of budget, the connection is considered feasible) and a feasible connection is randomly selected with probability connection_rate. If a connection is selected, a connection constraint is defined in the configuration.

The procedure is repeated for each component and finally, at the system level, one of the components with maximum depth is instantiated, so that the specification is feasible only if there exists at least a configuration for the selected component and its subcomponents that satisfy all the generated constraints. Finally, the QRML specification is automatically translated into SMT using the encoding described in the previous section.

For our analysis, we generate 5 benchmarks for each combination of the following parameters values: components $\in \{8, 16, 32, 64, 128\}$, configs $\in \{2, 4, 8, 16, 32, 64, 128\}$, depth $\in \{2, 4, 8\}$ and properties $\in \{0, 2, 4, 8\}$. We keep the value of some parameters fixed: interface_elements = components, subcomps = 5, expressions = 2, interface_elements_per_component = 3, and connection_rate = 0.5. In such settings, we therefore generate 2100 different benchmarks.

The three SMT solvers involved in our experimentation were selected from among participants to the QF_LIA division in the Single Query Track of the SMT Competition 2019 [28], namely Z3 (version 4.8.8) [29], CVC4 (version 1.7) [30] and SMTInterpol (version 2.5) [31]. The brief characteristics of these systems are listed below.

- Z3 is state-of-the-art SMT solver developed and maintained by Microsoft Research, which is focused at solving problems arising in software analysis and verification. It can be used to check the satisfiability of logical formulas over one or more theories. Z3 provides a compelling match for verification components and software analysis since several similar software constructs map directly into its supported theories.
- CVC4 is an efficient open-source automatic theorem prover for SMT problems. It can be used to prove the validity (or, dually, the satisfiability) of first-order formulas in a large number of built-in logical theories and their combination, including rational and integer linear arithmetic, arrays, bitvectors and a subset of non-linear arithmetic. CVC4 is intended to be an open and extensible SMT engine, and it can be used as a stand-alone tool or as a library, with essentially no limit on its use for research or commercial purposes.
- SMTInterpol is an SMT solver written in Java which supports the quantifier-free combination of the theories of uninterpreted functions, linear arithmetic over integers and reals, and arrays. Furthermore, SMTInterpol can produce models, proofs, unsatisfiable cores, and interpolants.

VI. EXPERIMENTAL ANALYSIS

In this section, we present the results of the experiments involving solvers and instances presented in Section V. All the experiments here reported ran on a workstation equipped with an Intel Xeon E31245 @ 3.30GHz CPU and 32GB RAM running Lubuntu 18.10 64bits. For all the experiments, we granted a time limit of 600 CPU seconds (10 minutes) and a memory limit of 30GBs. The source code and data used in the experiments are available at [34].

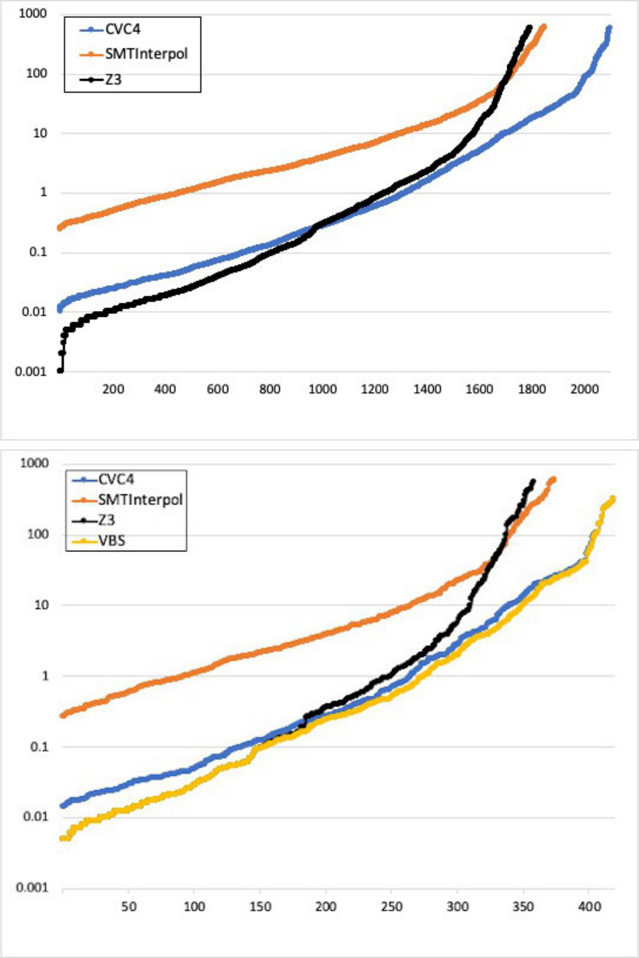

FIGURE 4. Performance of SMT solvers involved in the experiments on the whole dataset (top) and considering the median time value computed on the set of samples for each instance (bottom). Figures are organized as follows: in the x-axis is depicted the total amount of instance, while in the y-axis the CPU time in seconds.

Our first experiment aims to test the scalability of selected SMT solvers on generated instances. The results of such experiments are reported in Figure 4 (top). Looking at the figure, we can see that CVC4 outperforms SMTInterpol and Z3, solving 2098 instances (out of 2100), while the remaining solvers were able to solve 1849 and 1795 instances, respectively. It is worth to notice that no discrepancies have been reported in the satisfiability result returned by the solvers.

Despite the fact that CVC4 is able to solve almost all the instances in the whole dataset, looking in detail at the results, we can see that there are parameters settings for which CVC4 did not report the best CPU time. In order to investigate this point, we compute a dataset considering the *median* performance for each instance, i.e., for each pair solver/instance, we consider the median value obtained from the 5 random generated samples of the given instance. Notice that the median is computed separately for each solver, so the reported performance can be related to a different sample; this process can help us to make our analysis more robust.

Regarding the performance of the involved solvers in this dataset, we report that CVC4 is again the best solver; it was able to solve all the instances (420) in 5332.15 CPU seconds (s). The picture does not change with respect to the previous one, because the second best solver was SMTInterpol, which was able to solve 375 instances in 12456.92 s, while Z3 tops to 359 solved instances in 7117.58 s. Their performance is depicted in the cactus plot in Figure 4 (bottom).

Looking in details at the results, we can report a picture similar to the one obtained considering the whole set of instances: CVC4 is the best solver in general, but is not the best one for each instance. This consideration lets us introduce our second experiment, that consists in the analysis of the Virtual Best Solver (VBS), i.e., considering a problem instance, the oracle that always fares the best among available solvers. Looking again at the bottom of Figure 4, we can see that VBS solves the same amount of instances solved by CVC4 but spending 5014.30 CPU seconds. This different amount of time is due to the fact that CVC4 contributed to the VBS with 220 (out of 420) instances, while the second contributor was Z3 with 200 instances. Notice that, despite SMTInterpol solved 16 instances more than Z3, it did not contribute to the composition of the VBS.

The mapping for each instance between its parameters – in terms of feature described in Section V – and the best solver can help us to understand which is the best solver for a given setting. In our last experiment we investigate this point, and we compare the structure of the encodings related to the pool of instances in which CVC4 was the best solver against the pool in which Z3 was the best one. In order to refine our analysis, we discard all instances solved in less than 1 s and the ones for which the CPU time difference was less than 5%. At the end, we obtained a pool of 173 instances; for 139 of them the best solver was CVC4.

Concerning the analysis of the encoding related to this pool of instances, we report that we did not obtain a clear picture considering one feature at time. This is the motivation for which we introduce in our analysis a Machine Learning classifier, namely J48, the WEKA [32] implementation of the C4.5 [33] decision tree algorithm. We employed this algorithm (with WEKA default configuration) for data mining purposes, setting up a multinomial classification problem, which is structured as follows. Given a set of patterns, i.e., input vectors $X = \{\underline{x_1}, \ldots, \underline{x_k}\}$ with $\underline{x_i} \in \mathbb{R}^n$, and a corresponding set of labels, i.e., output values $Y \in \{1, \ldots, m\}$, where Y is composed of values representing the m classes of the multinomial classification problem, in our modeling, the n features are the parameters described in Section V, while the m classes are m SMT solvers (m = 2, namely CVC4 and Z3). Given a set of patterns X and a corresponding set of labels Y, the task of a multinomial classifier c is to construct c from X and Y so that when we are given some $\underline{x}^\star \in X$ we should ensure that $c(\underline{x}^\star)$ is equal to $f(\underline{x}^\star)$.

Considering the dataset composed as described before, we report that the model obtained after a run of J48 shows that CVC4 is the best choice when the total amount of components is less or equal to 64, but excluding the cases in which the number of properties are equal to 0 and the total amount of configurations is greater than 16; in these cases, the best choice is Z3. Z3 is also the best choice when the total number of components is greater than 64 and the total amount of configurations in smaller or equal to 4.

VII. CONCLUSION

In this paper, we have proposed an SMT-based approach for checking the consistency of configuration based-components design expressed in QRML in order to reduce faults and risks during the development process. Such an approach has been implemented into a tool which is available at [34]. Furthermore, to evaluate the scalability of the proposed approach, we developed an automated generator of DSL specifications and employed three different state-of-the-art SMT solvers to check the satisfiability of the encoded SMT properties. As we have shown in the experimental analysis, we demonstrated the effectiveness of the proposed SMT-based approach to verify configuration-based components design of various sizes within a reasonable time.

ACKNOWLEDGMENT

This work has partially received funding from the Electronic Component Systems for European Leadership (ECSEL) Joint Undertaking under grant agreement no 783162 (FitOptiVis) [11]. The research of Laura Pandolfo and Luca Pulina has been also partially funded by the Sardinian Regional Project PROSSIMO (POR FESR 2014/20-ASSE I).

REFERENCES

[1] E. A. Lee, "CPS foundations," in *Design Automation Conference*. IEEE, 2010, pp. 737–742.

[2] A. S. Vincentelli, "Let's get physical: Adding physical dimensions to cyber systems," in *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2015, pp. 1–2.

[3] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[4] M. Masin, F. Palumbo, H. Myrhaug, J. de Oliveira Filho, M. Pastena, M. Pelcat, L. Raffo, F. Regazzoni, A. Sanchez, A. Toffetti *et al.*, "Cross-layer design of reconfigurable cyber-physical systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 740–745.

[5] F. Palumbo, T. Fanni, C. Sau, L. Pulina, L. Raffo, M. Masin, E. Shindin, P. S. de Rojas, K. Desnos, M. Pelcat *et al.*, "CERBERO: Cross-layer model-based framework for multi-objective design of reconfigurable systems in uncertain hybrid environments: Invited paper: CERBERO teams from UniSS, UniCA, IBM Research, TASE, INSA-rennes, UPM, USI, Abinsula, AmbieSense, TNO, S&T, CRF," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, 2019, pp. 320–325.

[6] M. Narizzano, L. Pulina, A. Tacchella, and S. Vuotto, "Consistency of property specification patterns with boolean and constrained numerical signals," in *NASA Formal Methods Symposium*. Springer, 2018, pp. 383–398.

[7] ——, "Property specification patterns at work: Verification and inconsistency explanation," *Innovations in Systems and Software Engineering*, vol. 15, no. 3-4, pp. 307–323, 2019.

[8] S. Vuotto, "Automated requirements-based testing of black-box reactive systems," in *NASA Formal Methods: 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11–15, 2020, Proceedings*, vol. 12229. Springer Nature, 2020, p. 153.

[9] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[10] https://fitoptivis.eu, From the cloud to the edge smart IntegraTion and OPtimisation Technologies for highly efficient Image and VIdeo processing Systems (FitOptiVis).

[11] Z. Al-Ars, T. Basten, A. de Beer, M. Geilen, D. Goswami, P. Jääskeläinen, J. Kadlec, M. M. de Alejandro, F. Palumbo, G. Peeren *et al.*, "The FitOptiVis ECSEL project: highly efficient distributed embedded image/video processing in cyber-physical systems," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, 2019, pp. 333–338.

[12] https://qrml.org, Quality and Resource Management Language (QRML).

[13] M. Hendriks, M. Geilen, K. Goossens, R. de Jong, and T. Basten, "Interface modeling for quality and resource management," *arXiv preprint arXiv:2002.08181*, 2020.

[14] FitOptiVis, "D2.1: Component models, abstractions, virtualization and methods, v1.0," 2019. [Online]. Available: https://fitoptivis.eu/wp-content/uploads/2019/09/FitOpTiVis-D2.1.pdf

[15] C. Barrett, A. Stump, C. Tinelli *et al.*, "The SMT-LIB standard: Version 2.0," in *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, vol. 13, 2010, p. 14.

[16] L. Zhang and S. Malik, "Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications," in *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 2003, pp. 880–885.

[17] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[18] F. Jouault and J. Bézivin, "KM3: a DSL for metamodel specification," in *International Conference on Formal Methods for Open Object-Based Distributed Systems*. Springer, 2006, pp. 171–185.

[19] A. Cavalcanti, A. Miyazawa, A. Sampaio, W. Li, P. Ribeiro, and J. Timmis, "Modelling and verification for swarm robotics," in *International Conference on Integrated Formal Methods*. Springer, 2018, pp. 1–19.

[20] S. Keshishzadeh, A. J. Mooij, and M. R. Mousavi, "Early fault detection in DSLs using SMT solving and automated debugging," in *International Conference on Software Engineering and Formal Methods*. Springer, 2013, pp. 182–196.

[21] J. Hooman, "Industrial application of formal models generated from domain specific languages," in *Theory and Practice of Formal Methods*. Springer, 2016, pp. 277–293.

[22] E. K. Jackson, T. Levendovszky, and D. Balasubramanian, "Reasoning about metamodeling with formal specifications and automatic proofs," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 653–667.

[23] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[24] D. Le Berre and A. Parrain, "The Sat4j library, release 2.2," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 2-3, pp. 59–64, 2010.

[25] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski, "Clafer: unifying class and feature modeling," *Software & Systems Modeling*, vol. 15, no. 3, pp. 811–845, 2016.

[26] N. Jussien, G. Rochart, and X. Lorca, "Choco: an open source java constraint programming library," 2008.

[27] O. Semeráth, Á. Barta, Á. Horváth, Z. Szatmári, and D. Varró, "Formal validation of domain-specific languages with derived features and well-formedness constraints," *Software & Systems Modeling*, vol. 16, no. 2, pp. 357–392, 2017.

[28] L. Hadarean, A. Hyvärinen, A. Niemetz, and G. Reger, "SMT-comp 2019," 2019.

[29] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[30] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, Jul. 2011, pp. 171–177, snowbird, Utah.

[31] J. Christ, J. Hoenicke, and A. Nutz, "SMTInterpol: An interpolating SMT solver," in *International SPIN Workshop on Model Checking of Software*. Springer, 2012, pp. 248–254.

[32] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[33] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[34] Tools and benchmarks related to the experiments are available at https://gitlab.sagelab.it/sage/fitoptivis-smt-benchmarks.

LAURA PANDOLFO is Assistant Professor at the University of Sassari. In 2017, Laura obtained her PhD from the Department of Informatics, Bioengineering, Robotics and System Engineering (DIBRIS) at the University of Genoa. Her research interests include Semantic Web and Linked Data, Knowledge Representation, Ontology Design. Complementing her research activity, she has served as a reviewer of international conference and peer-reviewed journals, such as IJCAI and Semantic Web Journal, and in the organizing committee of different international conferences.

LUCA PULINA is Associate Professor of Computer Science. He received a Ph.D. in Computer Engineering and Robotics from the University of Genoa, Italy, in 2009. His research activities focus on Automated Reasoning, Formal Verification and Knowledge Representation. He served in the organizing and technical program committee of several international conferences. He has (co-)authored more than eighty publications in peer-reviewed journals, international conferences, and workshops. He was involved as principal investigator of a regional project, while he participated in European and national projects.

SIMONE VUOTTO is research assistant at the University of Sassari and PhD student at the University of Genoa. He graduated in Computer Engineering from the University of Genoa in 2016. He worked as a Software Engineer in the travel industry for over one year and for an high-tech Italian startup. His research activities focus on Formal Methods and Requirements Engineering.