Component-Based Synthesis of Embedded Systems Using Satisfiability Modulo Theories

STEFFEN PETER and TONY GIVARGIS, University of California, Irvine

Constraint programming solvers, such as Satisfiability Modulo Theory (SMT) solvers, are capable tools in finding preferable configurations for embedded systems from large design spaces. However, constructing SMT constraint programs is not trivial, in particular for complex systems that exhibit multiple viewpoints and models. In this article we propose CoDeL: a component-based description language that allows system designers to express components as reusable building blocks of the system with their parameterizable properties, models, and interconnectivity. Systems are synthesized by allocating, connecting, and parameterizing the components to satisfy the requirements of an application. We present an algorithm that transforms component-based design spaces, expressible in CoDeL, to an SMT program, which, solved by state-of-the-art SMT solvers, determines the satisfiability of the synthesis problem, and delivers a correct-by-construction system configuration. Evaluation results for use cases in the domain of scheduling and mapping of distributed real-time processes confirm, first, the performance gain of SMT compared to traditional design space exploration approaches, second, the usability gains by expressing design problems in CoDeL, and third, the capability of the CoDeL/SMT approach to support the design of embedded systems.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General; C.3 [**Special-Purpose and Application-based Systems**]: *Real-time and embedded systems*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic; J.6 [**Computer-Aided Engineering**]: *Computer-Aided Design (CAD)*

General Terms: Design

Additional Key Words and Phrases: Embedded systems, components, satisfiability modulo theory, design space exploration, systems specification methodology, modeling

ACM Reference Format:
Steffen Peter and Tony Givargis. 2015. Component-based synthesis of embedded systems using satisfiability modulo theories. ACM Trans. Des. Autom. Electron. Syst. 20, 4, Article 49 (September 2015), 27 pages.
DOI: http://dx.doi.org/10.1145/2746235

1. INTRODUCTION

The task of identifying suitable configurations from large design spaces has been a challenging problem since the early days of embedded system design. The problem continues to grow in scope in today's systems, as they grow in complexity and are embedded into large-scale cyber and physical applications. The extended range of viewpoints and models that have to be considered pose complex non-trivial trade-offs, dependencies and constraints, which call for powerful tools and strategies to explore the design spaces, searching for valid system designs.

A plethora of approaches have been proposed to explore the design spaces of embedded and cyber-physical systems [Gries 2004; Neema et al. 2014]. While many

This work was supported in part by the National Science Foundation under NSF grant number 1136146.
Author's address: Steffen Peter, email: st.peter@uci.edu.

DOI: http://dx.doi.org/10.1145/2746235

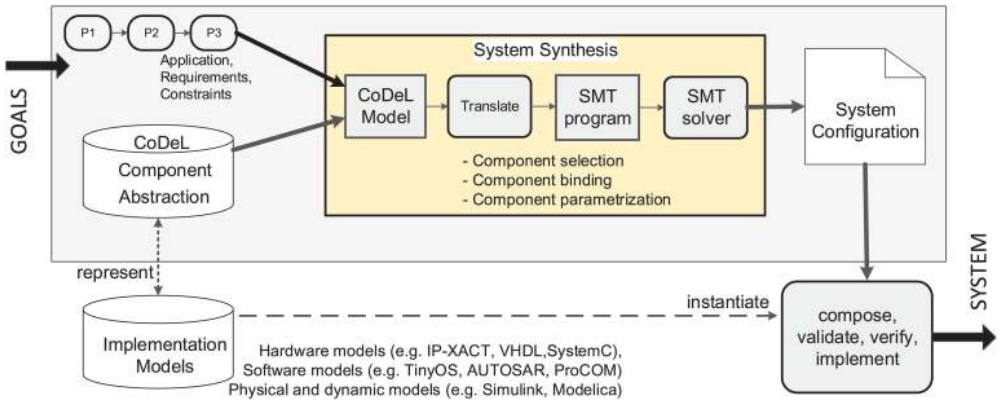


Fig. 1. Overview of the CoDeL/SMT-based system synthesis. The repository of components and the application requirements, expressed in the CoDeL language, are translated to an SMT program that can be solved with existing SMT solvers. The result is a system configuration that can be further analyzed and implemented.

exploration tools apply optimized problem-specific search algorithms, several tools have also harnessed the benefits of well-researched information theoretic solvers, such as Satisfiability solvers (SAT) [Haubelt and Feldmann 2003], Satisfiability Modulo Theory (SMT) [De Moura and Bjørner 2011], or Integer Linear programming (ILP) [Lukasiewycz et al. 2008]. In this article we focus on the application of SMT solvers. SMT efficiently extends predicate-logic-based SAT solvers with powerful background algebras, which include standard integer or floating point arithmetic, but also extended mathematical models to describe, for instance, execution times [Henry et al. 2014], real-time calculus [Kumar et al. 2013], or control quality [Aminifar et al. 2013]. SMT solvers answer the stated problems with outstanding performance, without the need for the design tool developer to implement a solver or a custom design space search algorithm. However, constructing SMT programs is non-trivial and requires in-depth knowledge of the mathematical abstraction as well as insight in the workings of the solvers. After all, an SMT program is still a nonstructured set of mostly predicate logic, which limits the understandability, reusability, and extensibility of the program. Therefore, the challenge for the system designer shifts from the actual solving of the design problem to the description (the construction) of the design space as an SMT program.

To cope with the challenge of constructing SMT programs that solve design challenges in the embedded system domain, this article proposes an algorithm to translate design spaces, expressed in a generic high-level component-based description language, into SMT programs. We present:

(1) CoDeL, a component-based description language to express applications, requirements, and building blocks, with their properties, constraints, and connectivity;
(2) synthesis rules that describe how systems are composed from the CoDeL building blocks, by supporting the concepts of component allocation, component connections (binding), and component parameterization;
(3) a CoDeL to SMT transformation algorithm, which translates the space of systems resulting from the CoDeL building blocks to an SMT program.

As a result, designers and tool developers may utilize the SMT performance for component-based systems that were expressed without the SMT formulation in mind. As illustrated in Figure 1, we first express the application requirements and the available set of components with their properties in CoDeL, then translate CoDeL into an

SMT program, which can be solved by SMT solvers to obtain a valid system configuration. The system configuration is used to instantiate implementation models, and to validate, verify, and implement the system. If the detailed validation step discovers failures, standard embedded system design techniques, such as backtracking the design process, refinement of models, or adding constraints, may be applied to find correct design points [Gajski et al. 2009].

Since CoDeL applies generic component-based concepts, such as components, interfaces, properties, and constraints, the proposed system synthesis and SMT transformation can represent the attributes of a wide range of existing implementation languages for embedded systems, describing hardware (e.g. IP-XACT (IEEE P1685) [Berman 2006], SystemC [Vachoux et al. 2003] or VHDL), software (e.g. TinyOS [Levis et al. 2005], C, or ProCom [Sentilles et al. 2008]), as well as physical interfaces and dynamic systems described in Simulink [2013] or OPENMODELICA.[1] Those languages already utilize the concept of modules or components and are regularly exposed to constraint-driven allocation, binding, and parameterization problems. Expressed in CoDeL and following our proposed translation algorithm, those design challenges can be answered fully, benefiting from the outstanding performance of SMT solvers.

We implemented CoDeL into a prototypical tool that allows its users to express the CoDeL components with their properties, and link the components to their implementation model. The tool further applies the CoDeL/SMT transformation algorithm and invokes the SMT solver Z3 [De Moura and Bjørner 2008] to solve the program and find one suitable system solution that satisfies the provided system constraints. We demonstrate the effectiveness of CoDeL/SMT to express, encode, and find suitable configurations for a range of examples including a distributed sensor and actuator control system. Our experiments confirmed the superior performance of the CoDeL/SMT approach while the use of CoDeL provides improved usability, reusability, and extensibility of applied components and models.

The rest of this article is structured as follows. After an overview of related work in Section 2, we introduce CoDeL and its composition rules in Section 3. In Section 4 we present the SMT encoding scheme in detail. Results for our prototype tool and a set of design examples are presented in Section 5. The article concludes with a discussion of the current results and an outlook for future work.

2. RELATED WORK

The exploration of complex design spaces has been a topic of great interest in the design community for decades, resulting in a variety of overview papers on this matter [Sangiovanni-Vincentelli and Martin 2001; Gries 2004], capable languages [Feljan et al. 2009], and tools. Tools such as the BIP (behavior, interaction, priority) framework [Bourgos et al. 2011], the Octopus toolset [Trcka et al. 2011], or SystemCoDesigner [Keinert et al. 2009] evaluate design choices for software and hardware early in the development, apply analytic models, and facilitate the composition and verification of systems of components. However, the applied methods and languages either provide only limited support for automated synthesis, or are specifically tailored to address a certain viewpoint. Our work is intended to be more general so that it can serve a variety of existing component frameworks as an abstract component meta-model in which common design space exploration problems can be addressed. In this regard, the work in our article relates generally to the platform based design (PBD) of embedded systems [Sangiovanni-Vincentelli and Martin 2001]. The goal of PBD is to select and compose a subset of available components (the configuration), from the design space of

[1] http://www.openmodelica.org.

possible platform configurations. A good and implementable configuration satisfies the system constraints as well as all component-imposed assumptions.

The correctness of a configuration can be validated in great detail by formal verification approaches [De Alfaro and Henzinger 2001a], which capture temporal aspects of compositions, or validated in complex (co-)simulations [Mühleis et al. 2011]. Such detailed approaches are essential for good system design, but should be executed after a configuration has been found. Due to exploration time requirements, such approaches do not seem suitable for the exploration of large design spaces, as intended in this article. Options to capture the correctness of a system on a higher level of abstraction are interface algebras [De Alfaro and Henzinger 2001b] or design contracts [Damm et al. 2005]. Interface algebras provide a clear semantics on how a system can be composed, and facilitate a formal assessment of the composability and compatibility of the structure of a system. Design contracts extend the idea of interface theories and pose a powerful tool to evaluate the promise-assumption relation between components. The use of design contracts has been demonstrated in the automotive industries [Damm et al. 2005], for cyber-physical systems [Derler et al. 2013], and airplane power systems [Nuzzo et al. 2014]. As we will discuss in the next section, CoDeL applies interface algebras and a general version of design contracts to express the structure and viewpoints of components and system.

The exploration of complex design spaces and searching for suitable system configurations is frequently fostered by satisfiability (SAT) solvers [Haubelt and Feldmann 2003], Integer Linear Programming (ILP) [Lukasiewycz et al. 2008] and SMTs [De Moura and Bjørner 2011; Reimann et al. 2010, 2011; Liu et al. 2011; Aminifar et al. 2013]. ILPs have been applied for a variety of design space explorations and optimizations [Lukasiewycz et al. 2008] for energy-efficient processor mapping of distributed applications [Gunes and Givargis 2014] or to configure avionic power systems [Nuzzo et al. 2014]. We considered ILP as an alternative for the SMT approach in this article. However, the strict requirements for linearity as part of the ILP rules do not allow the expression of even simple multiplications (e.g. for the computation of energy=power $\times$ time). ILP modulo theory (IMT) [Manolios and Papavasileiou 2013] extends ILP with a powerful background theory to solve complex problems, and has already been proposed to optimize resource planning and synthesis of industrial designs [Hang et al. 2011]. While IMT, with its optimization capabilities, is an interesting development, as of today no ready-to-use IMT solvers are available.

Due to their great performance in solving propositional problems, SAT techniques have been proposed for verification [Schmidt et al. 2013], synthesis [Haubelt and Feldmann 2003] and analysis of software systems and electronical systems. For example, Haubelt and Feldmann [2003] presented a mapping of the system synthesis binding problem to Boolean equations that could be solved by SAT techniques. However, the need to describe SAT programs entirely as conjunctive normal form (CNF) renders SAT less expressive than ILP and significantly complicates the encoding.

SMTs [De Moura and Bjørner 2011] address the shortcomings of SAT solvers with the addition of powerful background theories to solve equations expressed as standard algebra. Advanced background theories reason about the behavior of protocols to manage shared resources [Liu et al. 2011; Kumar et al. 2013] or execution times of real-time systems [Henry et al. 2014; Kumar et al. 2013]. As of today many solvers such as Z3 [De Moura and Bjørner 2008] or OpenSMT [Bruttomesso et al. 2010], as well as a standardized description language, SMT-LIB [Barrett et al. 2013] are available. SMT has been discussed for a variety of synthesis and analysis applications in the embedded systems domain. For instance, Reimann et al. [2011] described a platform-based synthesis of real-time systems with a highly optimized timing analysis. Liu et al. [2011] applied SMT to solve mapping and scheduling of tasks on multi-processor systems.

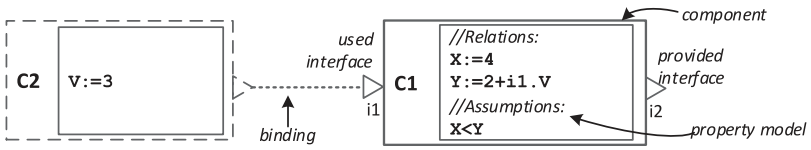


Fig. 2. Core elements of the component model. Component C1 has one used interface ($i1$) and one provided interface ($i2$). Properties of C1 are described by two relations and one assumption. Connecting component (C2) using the $i1$-interfaces, sets the values of variables in scope of C1 (i1.V=V of C2) so that the assumption (X<Y) can be satisfied.

Kumar et al. [2013] presented an SMT solver that applies a real-time calculus as background theory. The real-time calculus computes arrival curves for constrained resources, expressed in a very specific encoding scheme for this single use case. The background theories presented in those works are important for future use of SMT and can be applied in the property model of CoDeL. However in all those works, the actual SMT program is still an unstructured set of assertions, which limits their applicability and reusability.

To improve and simplify the programming of ILP and SMT solvers, several works proposed alternative programming styles and languages. For ILPs, languages such as AMPL (A Mathematical Programming Language) [Fourer et al. 1989] or the more expressive GAMS (General Algebraic Modeling System) [Bussieck and Meeraus 2004] help to express the program as a set of well-defined mathematical expressions. Other approaches such as Sheard [2012] and Agarwal and Karkare [2013] advocate for the expression of SMT problems as part of regular programming languages, which provide a classic programming interface instead of assertion-based definitions of the SMT-LIB. Such languages indeed help to improve the accessibility of ILP and SMT, while added support to formulate the problems is still limited. For embedded systems design, we need a translation from the component-based design space to the mathematical and assertion-based expressions as provided by SMT, ILP, or GAMS. For this purpose we describe the generic component description language CoDeL in the next section, and show the translation of the component model into an SMT program in Section 4.

3. COMPONENT-BASED SYSTEM MODEL

In this section we introduce the CoDeL component and system model and its graphical notation. We further discuss the resulting design space, which supports design variabilities component selection, component binding, and parameterization.

3.1. Component Model

For the underlying data structure of CoDeL, we apply a lightweight component model. Components are the building blocks of the system. Each component represents a characteristic functional behavior, which may be the behavior of a software module, a hardware module, or a model of the environment. A component c is represented by the triplet $c = (M, I, \mathbb{P})$, where M represents meta information, I the interfaces, and $\mathbb{P}$ the property model of c. M contains descriptive data about c, including its name, possible links to implementation, and simulation models for the component.

Figure 2 shows a graphical representation of two components, C1 and C2. Each component is represented as a box with a name (C1 and C2), a property model (the box in the component), and the interfaces (triangles). In the following we introduce the interface model and the properties model, using components C1 and C2 from Figure 2.

3.1.1. Interfaces and Bindings. Components provide their functions, services and properties to other components via interfaces. Reciprocally, interfaces are used to connect to

other interfaces in order to use their functionality. The set of interfaces of a component is described with the tuple $I = (I_P, I_U)$, where I_P is the set of interfaces provided by the component, and I_U is the set of used (needed) interfaces. In Figure 2, C1 uses the interface named *i1* and provides the interface *i2*. In the graphical notation of CoDeL, the used interfaces are illustrated as triangles that point toward the component, the provided interfaces are triangles pointing away from its component.

A *binding* $\beta_{i,j}$ is a connection between two interfaces $i \in I_P$ and $j \in I_U$, while $i \cong j$. The compatibility operator $\cong$ means that i and j are the same type and i provides a compatible interface to the interface used by j. The compatibility operator is in accordance with interface algebras [De Alfaro and Henzinger 2001b] and can be computed offline based on extended functional and state-based reasoning. The set of all possible bindings of a design space is noted as B throughout the article, while B_S notates the set of bindings active in the current system. A solid line between two components shows an active connection between the two interfaces, while a dashed line illustrates the possibility of a connection. Figure 2, shows a possible binding between *i1* and the output interface provided by C2.

3.1.2. Properties. The property model $\mathbb{P}$ is a declarative model of the properties of the components. A property describes a characteristic or quality of the component. Properties comprise all aspects of a system that can be expressed by data types, including all sorts of attributes, to describe the system, its components, its requirements, or its environment. The declarations, as part of the property model, contain relations and assumptions, so that we can express the property model as a triplet $\mathbb{P} = (P, R, A)$, with properties P, relations R, and assumptions A. In the graphical notation, the property model is shown as a separate text box within the component.

A *Property* $p \in P$ is a typed variable. p can be of a basic type such as Boolean, string, integer, or floating point number, but also an extended data type such as a set, vector, or a scheduling arrival curve [Kumar et al. 2013]. The types are not freely programmable, but defined by the available set of background algebras in the underlying solver.

Relations R assign the result of an operation over a set of input properties or constants to a property. The operations can be standard arithmetic, logic, and conditional operations, but also complex background-theory-specific operations, such as set-theoretic operations, or the aggregation and comparison of scheduling arrival curves. As a small example, component C1 in Figure 2 contains two relations on integer data types: $X := 4$ and $Y := 2 + i1.V$. The first relation assigns the value 4 to the property X. The second relation assigns the sum of 2 and the variable $i1.V$ to the property Y. The use of $i1.V$ in C1 already indicates that properties can be defined in the context of the composition. By binding C2 to the interface *i1* of C1, the properties of C2 become available in the scope of C1. Forwarding of variables via bindings facilitates reasoning about component properties in the context of the system composition.

Assumptions are similar to relations, as they compute a function over a set of input properties. However, the result of an assumption is a Boolean, which is not assigned to a variable, but it is evaluated to determine the correctness of the system. The overall goal of the system synthesis is that all assumptions of a system are satisfied, in other words they evaluate to true. In Figure 2, C1 contains one assumption, which is $X < Y$. Property X is defined as 4, while Y has to be evaluated in context of the binding. Due to the binding, $i1.V$ is defined by C2 as 3, so that Y is 5, which resolves the assumption to be satisfied.

In addition to standard algebraic terms, which can be directly processed by a solver, CoDeL supports macro operations and complex interpreted functions. Macro operations, such as `each(interface):R`, do not extend the expressiveness but reduce the complexity of $\mathbb{P}$ by applying a relation (R) or assertion, to a set of properties or

interfaces (i). Macros are, for instance, utilized in the scheduling example shown later in Figure 5.

3.2. Design Space and System Synthesis

The system synthesis relates to the composition of a system implementation (S) from a predefined repository of available components ($\mathbb{C}$). The system implementation contains a set of selected components $C_S \subseteq \mathbb{C}$, the set of bindings B_S between the interfaces of the selected components, and the parameterization of the properties of C_S. The design knobs in this design space are selection, binding, and parameterization of components.

Component Selection addresses the selection of the subset C_S from the set of available components $\mathbb{C}$. Therefore, the selection step decides whether or not an available component is used as part of the composed system.

Binding addresses the selection of active bindings $B_S \subseteq B$ from the set of possible connections between components (B). The selection is influenced by a range of structural (each $j \in I_U$ must be connected not more than once; adjacent components C_i and C_j must be $\in C_S$), and overfunctional constraints, since each active binding propagates properties to adjacent components. A common use case in the embedded system design that involves binding is the mapping of processes to resources such as memory or processing units (PUs). Such an example is discussed in detail in Section 5.3. We discuss bindings in detail in Section 4.2.

Parametrization concerns the definition of variable properties ($p \in P$) of the property model of C_S. Such variable system properties are, for instance, the update frequency of a protocol, which directly influences the energy consumption of the system, but also the quality of service. Another example for the parameterization issue in embedded system design is the planning of periodic real-time schedules. The example shown in Section 5.2 assigns start times of a schedule in order to satisfy the timing constraints.

Parametrization made for one component may propagate through the system components (C_S) via the active bindings (B_S). Therefore, to solve the resulting design problems, capable solvers are needed that in addition to the propositional logic for components and bindings, have the ability to process a set of equations that express inequalities and equalities over a set of variables. SMT solvers contain such a solver as part of their background theory, which makes them a promising tool for the exploration of design spaces that contain parameterization problems.

3.3. Expressiveness and Application of CoDeL

In this section we discuss the abilities and limitations of CoDeL to express structural and over-functional attributes of the embedded system. We further discuss the general application of CoDeL. As introduced earlier in this article, CoDeL is a generic component description language with a very condensed set of semantic features. CoDeL represents hardware and software components as well as physical interfaces and systems, described in different implementation languages. The main purpose of CoDeL, as presented in this article, is to define a suitable abstraction between implementation model languages and the capabilities of formal solvers. Therefore such a language must:

(1) capture the structure of the design space and its components;
(2) allow the expression and evaluation of important over-functional properties;
(3) provide a good usability and reusable concept;
(4) be evaluated efficiently and be able to be encoded in solver languages.

While the last item will be addressed under SMT encoding in the next section, we discuss CoDeL's support for the first three items in the following paragraphs.

3.3.1. Expression of Structure. With the ability to model components (blocks, modules), interfaces (ports), and bindings (connections), CoDeL represents the standard concepts of today's design tools that describe the structure of systems. In our experiments components were modeled originating from design languages and tools like C, TinyOS, SystemC, VHDL, and Simulink. Most of the languages already support a well-defined component and interface model, or (as for C) are suitable to add meta-information to express components (or modules). Therefore, CoDeL supports basic stateless interface algebras as described in De Alfaro and Henzinger [2001b] to validate the structural correctness of a composed system, including the validity of composition of two components ($C1 \| C2$), and connection of two interfaces ($i1 \| i2$). To evaluate the compatibility and composability of interfaces and components in greater detail, CoDeL implements a functional variant of design contracts [Damm et al. 2005]. Design contracts assume that two components, C_1 and C_2, with their contract data as part of the information $M_i = (P_i, A_i)$ can be composed ($C1 \| C2$) if $A_1 \subseteq P_2$ and $A_2 \subseteq P_1$, meaning that all assumptions of a component are delivered by properties promised by the peer. A system S of components is free of conflicts if the aggregated set of assumptions is satisfied by the aggregated set of proposed properties.

$$\bigcup_{component\ c \in S} A_c \subseteq \bigcup_{component\ c \in S} P_c.$$

Since each assumption can be expressed as function $f_a : P \rightarrow true$ iff $a \in P$, a system, expressed in CoDeL is free of conflicts if

$$\forall a \in A_s : f_a(p^*_{ina}) = true,$$

where $p^*_{ina} \subseteq P$ is the subset of relevant system properties to evaluate f_a. In theory, this formalism allows us to analyze the compatibility of complex dynamic interface descriptions like the interface automata proposed by De Alfaro and Henzinger [2001a]. However, modeling and analyzing system details on this low level of granularity is not the primary concern of CoDeL, since such details would violate our requirement (4), that is, efficient evaluation. Instead, we can evaluate the compatibility of interfaces ($i1 \cong i2$) offline and store the information as part of the component model.

3.3.2. Expression of OverFunctional Attributes. The assumption-based evaluation of designs is not only suitable to assert the constraints of the structure, but also to reason about over-functional attributes of the system. In the examples in Section 5 we show how CoDeL can model timing, schedulability, shared resources, and control quality. In the context of design contracts, high-level system analysis has been successfully applied for systems ranging from automotive [Damm et al. 2005], avionic [Nuzzo et al. 2014], to general cyber-physical system systems [Derler et al. 2013]. Other static high-level analysis approaches, for example, spreadsheet analysis, have been successfully applied to model industrial systems [Trcka et al. 2011], general worst-case execution times [Henry et al. 2014; Lednicki et al. 2013], power consumption [Gunes and Givargis 2014], system security [Peter et al. 2008], and scheduling [Zhang and Burns 2009; Zhu et al. 2012]. This enumeration is not conclusive but indicates the suitability of equation-based approaches to estimate and analyze over-functional properties of systems in practice.

Within CoDeL the following approaches are supported to model and evaluate static and dynamic properties of systems.

—*Static resolution or approximation of dynamic behavior*. This is one standard approach for high-level abstraction to assess system properties in related work, as listed in the preceding. The idea is to express system properties of interest as equations, which can be efficiently evaluated in many tools. The disadvantage is a lack

of accuracy in the models that may result in an over- or under-approximation of properties.

—*Enumeration of all possible states, as proposed by [Nuzzo et al. 2014].* The system is correct if no conflict can be found for any state. While theoretically very powerful, state-explosion renders this approach infeasible for most cases.

—*Design of a viewpoint-specific algebra (VSA).* VSAs facilitate equation-based computation with complex data types, but require an algebra-specific background computation in the solver. As one example, Kumar et al. [2013] showed how a real-time calculus can be integrated as a background theory in an SMT, where property variables express scheduling arrival curves $(a_1, a_2, \ldots)$ that can be aggregated $(a_S = \sum_i a_i)$. Resource constraints are expressed as a service curve (s_C), and constraint, with the assumption $a_S \leq s_C$ in CoDeL. Other examples for VSA are composable security [Martin et al. 2014] or execution times [Lednicki et al. 2013].

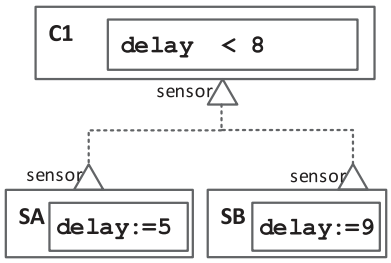
—*Invoking complex executable models.* Executable models (e.g. in Simulink) or implementation models, can be invoked to evaluate complex properties. For example, our evaluation tool, discussed in Section 5.1, can invoke complex Simulink simulations as black boxes, for instance to assess the quality of a parameterizable control system. The result is accurate, but usually expensive in terms of computation time.

While all four approaches are supported by CoDeL and in current SMT solvers, the static models are preferable, due to their performance advantages. However finding and expressing good static models for over-functional properties of systems is not trivial, and therefore needs the creativity of knowledgeable domain experts, as well as a tool infrastructure that fosters reusability of models.

3.3.3. Application of CoDeL. The application of CoDeL is separated into two phases: a setup phase of the component repository and the actual design synthesis phase.

The setup phase is ideally supported by component information that is already available as part of the implementation model. This is generally possible for structural information, but also over-functional properties, such as WCET and memory consumption. Most over-functional models, however, require domain experts to be designed correctly. For example, we need control experts to express models for complex dynamic behaviors, battery experts to model batteries used by the systems, software engineers to model properties of their software modules, and hardware designers for hardware components. Important in this context is reusability. Once described, even by third-party domain-experts, the models should be reusable and extensible within the system and for future systems. In Section 5.4.1 we discuss this issue for a model to assess the control quality. From an organization perspective, reusability requires a management structure to develop, store, and describe models and components, and an extended component semantics that goes beyond the generic component model discussed in this article. The organization challenge is not the focus of this article but requires additional studies in future work.

In the actual design phase, CoDeL supports manual design by analyzing preconfigured configurations, or automatic search of the design space for suitable configurations. For a model instance with allocated components (C_S), bindings (B_S), and parameterization (P_S), the configuration can be evaluated with an evaluation function $eval(S(C_S, B_S, P_S) \rightarrow \{true, false\}$, which is true if all interface requirements and all assumptions are satisfied. The evaluation function, which can be solved by dedicated solvers, is the core functionality for automatic search. The search is initialized with an incomplete description of S, expressing the invariant components, bindings, and parameters of the system. Usually the initial system contains a main or top component, system constraints, parts of the hardware platform, and the physical environment. The search uses a repository of components for allocations, bindings, and parameterizations,



ALGORITHM 1: SMT program for Example 1

```
declare-const C1, SA, SB Bool                    //(R1)
declare-const C1.sensor,SA.sensor,SB.sensor Bool
declare-const delay Int
Imply(C1, delay<8)                               //(R2)
Imply(SA, delay=5)
Imply(SB, delay=9)
C1=C1.sensor                                     //(R3)
SA=SA.sensor; SB=SB.sensor
Imply(C1.sensor, SA.sensor or SB.sensor)         //(R4)
C1=true
```

Fig. 3. CoDeL diagram and SMT program for Example 1. Component C1, has a timing assumption and uses the sensor interface. Sensors SA and SB provide the service but have different delay properties.

as introduced in the previous sections. Since the complexity of the design space prohibits exhaustive search in most cases, advanced reasoning and conflict analysis are standard approaches to reduce the size of the practical design space. As outlined in the related work, many approaches exist to perform the final search. To solve the search with SMT solvers, we propose an algorithm to encode a CoDeL to an SMT program in the next section.

4. CODEL TO SMT TRANSFORMATION

In this section we show how to encode the CoDeL-defined design space, including its components, their properties, and constraints, to an SMT program. We start with a small straightforward component encoding. This first step also illustrates the basic operations of an SMT solver. The second part of the section then explains the encoding of the bindings between components and the propagation of variables between components. This section concludes with the complete algorithm that translates the component-based design space into the SMT program. The code examples for the rules in this section are based on the SMT2 library standard [Barrett et al. 2013]. To improve the readability, we changed the prefix notation of the algebraic terms to a standard infix notation. The rules are annotated as R1 to R8 for reference throughout this article.

4.1. Component Encoding

As the first step, we show how to encode the components and their properties. For this purpose we use a small example (Example (1)), shown in Figure 3. Example (1) shows the three components C1, SA, and SB. We can assume C1 is a control component that needs input from a sensor. The design space in Example 1 offers two sensors (SA and SB), while SA has a delay of 5 and the delay of SB is 9. In this article, and without loss of generality, we use milliseconds as the unit for the delay.

The basic strategy of the component encoding is to express the instantiation of a component as a Boolean variable. In Example (1), the variable C1 is true if component C1 is instantiated in the target design, and C1 is false if C1 is not part of the target design. Therefore, in Example (1) three conditional variables C1, SA, and SB define the component design space. Using SMT-LIB syntax, the three variables are defined as follows.

```
declare-const C1 Bool                    (R1)
declare-const SA Bool
declare-const SB Bool
```

Relations and Assumptions. In SMT, relations and assumptions of the components, which are in turn equations, do not need special encoding, because the underlying SMT

theory can solve the algebraic terms as is. For instance, for the statement `delay=9`, the solver assigns the value 9 to the variable *delay*. If the program contains another unconditional definition of delay, such as `delay<8`, the solver returns UNSAT. Conversely, if the second statement is `delay>8`, the solver returns SAT, since `delay>8` and `delay=9` do not impose a conflict. Such algebraic statements can be conditional or part of a larger statement. For this operation we use the SMT operation *Imply*. Imply(x,y), means that y must be true if x is true. Both, x and y, may be complex terms on their own.

In our component model we use SMT's *Imply* facility to activate relations and assumptions whenever the component is enabled as part of the target design.

```
Imply(component, relation AND assumptions)
```

means that the relations and assumptions have to be true (i.e., satisfied) if the component is enabled. In Example (1) we can write the following.

```
Imply(C1, delay<8)                                    (R2)
Imply(SA, delay=5)
Imply(SB, delay=9)
```

It should be noted that before those statements, the variable *delay*, like all variables, has to be declared.

```
declare-const delay Int
```

Applying the rules discussed thus far, we can proceed to solve the program as follows. First, we set the initial requirement (`C1=true`), and then we check the satisfiability. The commands are

```
C1=true
check-sat,
```

which are also part of the SMT program.

The SMT program we defined so far is executable and satisfiable, but does not necessarily return the expected result. The solver may assign the value of 0 to the delay and disable both sensors, which still satisfies all assumptions. This is caused by the missing expression of possible bindings between the components. Therefore, we describe the interfaces and possible connections next.

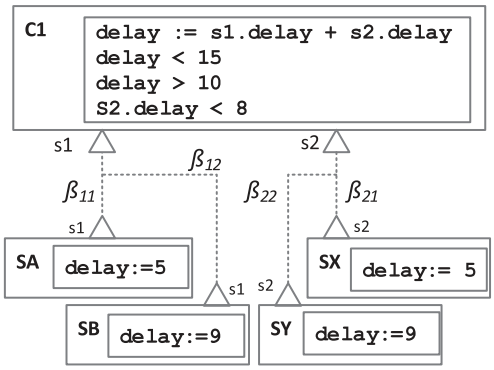
Interfaces. Interfaces in the SMT program are expressed as binary variables, because an interface is either part of the system or is not part of the system. Since interfaces are fixed to their components, we can express the equivalence relation between the interface and its component, which means that the interfaces of the component are part of the design, if and only if the component is active as well. The corresponding SMT statements for Example 1 are the following.

```
C1=C1.sensor                                          (R3)
SA=SA.sensor
SB=SB.sensor
```

Utilizing the interfaces, we rephrase the dependency between C1 and the sensors:

```
Imply(C1.sensor, SA.sensor or SB.sensor)              (R4)
```

which means that the sensor interface from C1 requires at least one sensor interface provided by SA or SB. This completes the rules needed to encode Example 1. The resulting SMT program for Example 1 is shown as Algorithm 1. The program delivers the expected result, that is, the solver returns SAT, and outputs the following result: C1 and SA are enabled, and the delay is 5.



ALGORITHM 2: SMT program for Example 2

```
Imply(SA, SA.delay=5)                          //(R2)
Imply(SB, SB.delay=9)
Imply(SX, SX.delay=5)
Imply(SY, SY.delay=9)
Imply(C1, C1.delay=C1.s1.delay+C1.s2.delay)
Imply(C1, C1.delay<15)
Imply(C1, C1.delay>10)
Imply(C1, C1.s2.delay<8)
C1=(C1.s1 and C1.s2)                           //(R3)
SA=SA.s1; SB=SB.s1; SX=SX.s2; SY=SY.s2
Imply(β11, C1.s1 and SA.s1)                    //(R5)
Imply(β12, C1.s1 and SB.s1)
Imply(β21, C1.s2 and SX.s2)
Imply(β22, C1.s2 and SY.s2)
Imply(β11, not β12)                            //(R7)
Imply(β21, not β22)
Imply(C1.s1, β11 or β12)                       //(R6)
Imply(C1.s2, β21 or β22)
Imply(β11, C1.s1.delay=SA.delay)               //(R8)
Imply(β12, C1.s1.delay=SB.delay)
Imply(β21, C1.s2.delay=SX.delay)
Imply(β22, C1.s2.delay=SY.delay)
C1=true
```

Fig. 4. Example 2. Component C1 uses two sensor interfaces, s1 and s2. For both types of sensor two alternative sensors are available, which results in four possible bindings ($\beta 11 - \beta 22$). The assumptions in C1 allow the combination SB and SX (via $\beta 12$ and $\beta 21$) as the only conflict-free design.

4.2. Bindings

With the encoding rules discussed so far, we can encode very simple systems, such as Example (1). More complex designs, however, require additional rules, in particular to address the binding between components. Example (2), shown in Figure 4, illustrates the binding issue. In Example (2), C1 represents a control algorithm that uses the input of two sensors. C1 has been specified for a combined input delay of 10 to 15. The second sensor must be faster than 8.

Key in the encoding of the bindings is the computation of all possible bindings between the components of the design space. For all possible bindings between two interfaces $i,\ j$, $i \cong j$, we create one propositional variable $\beta_{i,j}$. This variable $\beta_{i,j}$ is true if the binding is active, which means that the interfaces are connected. Figure 4 illustrates the binding variables as β_{11}, β_{12}, β_{21}, and β_{22}.

Naturally, bindings require adjacent interfaces to be present. We can express this dependency in SMT as follows.

```
Imply(β11, C1.s1 and SA.s1)                    (R5)
Imply(β12, C1.s1 and SB.s1)
Imply(β21, C1.s2 and SX.s2)
Imply(β22, C1.s2 and SY.s2)
```

Notably, the implication of the presence of the interfaces as a function of the binding redefines the feasible design space. So far we considered the design space to be defined over the combinations of components. Instead, due to the equivalence relation between the presence of the component and its interfaces (e.g. `C1=C1.s1`), an active binding enables not only the adjacent interfaces, but also the components and their properties.

To the set of rules, discussed so far, we further add two rules to ensure that, first, each required interface I_U has a connected peer, and second, that each interface I_U is connected to not more than one peer interface.

In Example (2), the first rule is needed to ensure that the interfaces *s1* and *s2* of component C1 do not connect to more than one sensor each. In the SMT program this can be described by

$$\begin{aligned}&\texttt{Imply(C1.s1, } \beta 11 \texttt{ or } \beta 12) \\ &\texttt{Imply(C1.s2, } \beta 21 \texttt{ or } \beta 22),\end{aligned} \qquad (R6)$$

which extends and replaces the interface-based dependency rule, $R4$.

Using a second rule, we ensure that the interface is not connected to more than one other interface, that is, if $\beta 11$ is enabled, $\beta 12$ must be disabled, which is expressed as follows.

$$\begin{aligned}&\texttt{Imply(}\beta 11\texttt{, not } \beta 12) \\ &\texttt{Imply(}\beta 21\texttt{, not } \beta 22)\end{aligned} \qquad (R7)$$

Variable forwarding. As introduced in Section 2, the bindings in CoDeL are the mechanism to connect components, but also forward property variables from one component to a connected component. By forwarding property variables, components learn about the properties of the connected components. The accessed properties then can be validated or processed to update the own properties. The principle of variable forwarding was introduced in Figure 2, where the variable v was forwarded to the *i1* interface of component C1. To encode the variable forwarding for the entire design space the following two questions must be addressed: first, how to identify and manage compatible variables, and second, how to realize conditional forwarding.

For variable encoding we introduce scopes to express the host component of each property variable. Therefore, the five *delay* variables of Example (2), are expressed as five distinct variables (C1.delay, SA.delay, SB.delay, SX.delay, and SY.delay). Additionally, each component can access variables of connected components via the interface that is used for the binding. The interface can be considered as another internal scope of the components. In Example 2, component C1 can access all properties of the component connected via interface *s1* in the scope C1.s1. Accordingly the property names in the relations and assumptions from the property model are renamed, which has to be reflected within a refined Rule (R2). For example, the first relation in C1 is

```
C1.delay:=C1.s1.delay+C1.s2.delay,
```

and the updated SMT instruction following rule (R2) is

```
Imply(C1, C1.delay:=C1.s1.delay+C1.s2.delay).
```

Applying the concept of scopes, a binding simply forwards the local variables to the interface of the connected component. If component CX is connected to CY via interface *iy*, the set of relations that describe the forwarded variables (F) is expressed as

$$F = \bigcup_{v \in CY.\mathbb{P}} \text{relation}(CX.iy.v := CY.v),$$

resulting in a set of relations that forward all variables of CY to the scope CX.iy of CX. With the knowledge of the actually used variables within CX, the set of relations can be reduced to only relevant variables. However, one strength of SMT solvers is the efficient handling of unused variables, so that in our experiments, we would not have to measure a performance penalty for forwarding all variables.

Conditional variable forwarding is realized with an implication rule that asserts the correctness of the generated relations (F) only if the corresponding binding is active.

For Example (2), the resulting variable forwarding instructions in SMT are expressed as follows.

```
Imply(β11, C1.s1.delay=SA.delay)                    (R8)
Imply(β12, C1.s1.delay=SB.delay)
Imply(β21, C1.s2.delay=SX.delay)
Imply(β22, C1.s2.delay=SY.delay)
```

Applying the binding rules and variable forwarding rules, discussed in this section, we can conclude the SMT program and solve Example (2). The resulting SMT program without variable definitions is shown in Algorithm 2. The solver confirms the satisfiability (SAT) of the problem with the following assignments: bindings $\beta 12$ and $\beta 21$ are enabled, components C1, SB, and SX are enabled, and the delay in component C1 is 14.

4.3. CoDeL to SMT Encoding Algorithm

In the previous sections we discussed the rules to encode the component-based design space for two small examples. In the following, we present the general algorithm that applies the SMT encoding rules to all design spaces that can be expressed with the component model. The algorithm has the following inputs and outputs.

Input. Repository of components $\mathbb{C}$; each component C_i with its set of used and provided interfaces (I_p, I_u), and its property model $\mathbb{P}$. Additionally given is a start set of components $C_S \subseteq \mathbb{C}$ containing abstract functions and invariable platform components.

Output. An SMT program expressing the design space and the requirements.

The resulting algorithm `generate_SMT`, is shown as Algorithm 3. The rules for the algorithm were discussed in the previous sections. The steps in the algorithm are annotated with the rule numbers R1-R8 (R4 was overruled by R5). Beside the basic rules, the algorithm uses the following steps and external methods. In lines 1 to 6 of Algorithm 3, the set of possible bindings is created. The method `SMT_add_implication`, which is used in lines 9, 11, 15, 23, and 33, adds an implication (*Imply*) instruction, `SMT_add_assertion` (line 28) adds an unconditional assertion, and `SMT_define_variable` adds the definition for a variable into the SMT program. Method `forward_shared_variables` (line 10) generates the set of relations (F) that forward component variables to the corresponding interface scope of the connected component, as described for rule (R8) in the previous section. The resulting set F of forwarding relations is added as implied assertions in line 11. Method `or_conjunct(x)` (Line 33) conjuncts the members of the set o with an OR operation, which is needed for the computation of the allowed bindings for a used interface (Rule R6).

The complexity of Algorithm 3 is determined by the number of possible bindings. For n interfaces, in the worst case, the set B contains n^2 bindings. Since this set is checked for incompatible bindings (Lines 13–16), in the worst case the complexity of the algorithm is $O(n^4)$. In practice, even for large examples, as discussed in the next section, the SMT programs could be generated in a few seconds, as we will discuss in detail in the next section.

5. EXAMPLES AND EVALUATION

The goal of the evaluation presented in this section, is to demonstrate the suitability of CoDeL and the SMT transformation algorithm to address important challenges in the design of embedded systems. Specifically we consider the following desirable criteria.

—*Usability of SMT for system designers.* The motivation is to show that our approach is easier to apply than traditional SMT description techniques for systems.

ALGORITHM 3: Generation of the SMT program from a component repository

Input: system repository $\mathbb{C}$

```
1  //Generate_Bindings
2  foreach used interface i ∈ I_U in ℂ do
3  |  foreach provided interface j ∈ I_P in ℂ do
4  |  |  if i ≅ j then B ← B ∪ β_{i,j}
5  |  end
6  end
7  //add binding rules
8  foreach possible binding β ∈ B do
9  |  SMT_add_implication(β, β.I_U AND β.I_P)                    //(R5)
10 |  F=forward_shared_variables(β)
11 |  foreach relation f ∈ F do SMT_add_implication(β, f)        //(R8)
12 |  //prevent double assign of used interfaces                 //(R7)
13 |  for each possible binding β_2 ∈ B do
14 |  |  if β.I_U = β_2.I_U then
15 |  |  |  SMT_add_implication(β, not β_2)
16 |  |  end
17 |  end
18 end
19 for each component c ∈ ℂ do
20 |  //add component properties,relations,assumption
21 |  for each property p in c.P do
22 |  |  SMT_define_variable(p)                                  //(R1)
23 |  |  foreach relation r ∈ c.R ∪ c.A do SMT_add_implication(c, r)   //(R2)
24 |  end
25 end
26 //add interface rules
27 foreach interface i ∈ c.I do
28 |  SMT_add_assertion(c = i)                                   //(R3)
29 |  //require binding for used interfaces                      //(R6)
30 |  if i ∈ I_U then
31 |  |  o = ∅
32 |  |  foreach β ∈ B do if β.I_U = i then o ← o ∪ β
33 |  |  SMT_add_implication(i, OR_conjunct(o))
34 |  end
35 end
```

—*Expressiveness* demonstrates the suitability of CoDeL to state and process nontrivial structural and overfunctional system properties.
—*Reusability* and *Extensibilty* is the possibility of applying independently developed models and components to compose complex systems in an iterative design effort.
—*Scalability* applies to the SMT transformation algorithm and the performance of the search for suitable system configurations.

To discuss the criteria, we present four use cases, which each highlight a specific aspect of our approach.

(1) *Job Scheduling* (Section 5.2) discusses a very simple scheduling encoding and compares manual SMT encoding to the generated results. The example addresses the criteria *usability* and *extensibility* and facilitates a direct comparison of the generated SMT program to a manual encoding.

(2) *XGRID* (Section 5.3) demonstrates process mapping for a many-core architecture. The use case shows in detail how software mapping and hardware configuration can be solved in SCT, and addresses the concerns *reusabilty* of components and *scalability*. Further, XGRID facilitates a comparison against a state-of-the-art ILP encoding.
(3) *Inverted pendulum* (Section 5.4.1) is a small example that focuses on the *expressiveness*. The example shows in detail the modeling options for one real-time control subsystem. Pendulum is one part of the larger fourth example.
(4) *Distributed control systems* (Section 5.4.2) is a mapping use case of several concurrently executed real-time control applications on a distributed platform. Besides reusability, the example mainly addresses scalability. We apply the example to compare the SMT performance to alternative search methods.

Before the detailed description of the examples, we briefly introduce our design tool, which we applied for the evaluation. A discussion of the gathered results concludes this section.

5.1. Evaluation Tool

For evaluation purposes, we implemented a tool, System Configuration Toolkit (SCT)[2]. SCT allows the designer to express the components, as described in Section 3 of this article, and presents the resulting design space graphically. We can create, load, and connect the components, and manage a component repository, in the graphical notation used throughout this article, or in the underlying XML format. For the experiments, structural and property models of the components examples have been manually added. For extended analysis of over-functional properties we can directly invoke external models in Matlab.

For the modeled components, SCT performs SMT encoding as described in Algorithm 3, and instantiates the SMT solver Z3. In addition to the instantiation of the SMT solver, SCT also employs classic design space exploration techniques, such as tree search and conflict-analyzing tree pruning techniques [Zhang et al. 2001]. We will briefly compare the three applied techniques in Section 5.4.2.

Applying SCT, in the following sections, we discuss the stated design experiments. All measurements were obtained without parallelization on a 3GHz i7-processor-equipped PC.

5.2. Use Case: Job Scheduling

The job scheduling use case concerns offline planning of task schedules for real-time systems. For comparison we use a small example system presented by De Moura and Bjørner [2011]. The example considers three jobs $J_1,\ J_2,\ J_3$. Each job consists of two sequential tasks running on two processing units respectively. It is assumed that the jobs share the PUs. $d_{j,m}$ annotates the time needed for the job j for its task m running on the PU. In this example $d_{1,1} = 2,\ d_{1,2} = 1,\ d_{2,1} = 3,\ d_{2,2} = 1,\ d_{3,1} = 2,\ d_{3,2} = 3$. The question is, can the three jobs be scheduled so that they finish within 8 time units? De Moura presented a small SMT program that solves the problem.

$$\begin{aligned} t_{1,1} \geq 0 \wedge t_{2,1} \geq 0 \wedge t_{3,1} \geq 0 \wedge 8 \geq t_{1,2} + 1 \wedge 8 \geq t_{2,2} + 1 \wedge 8 \geq t_{3,2} + 3 \wedge \\ t_{1,2} \geq t_{1,1} + 2 \wedge t_{2,2} \geq t_{2,1} + 3 \wedge t_{3,2} \geq t_{3,1} + 2 \wedge \\ (t_{1,1} \geq t_{2,1} + 3 \vee t_{2,1} \geq t_{1,1} + 2) \wedge (t_{1,1} \geq t_{3,1} + 2 \vee t_{3,1} \geq t_{1,1} + 2) \wedge \\ (t_{2,1} \geq t_{3,1} + 2 \vee t_{3,1} \geq t_{2,1} + 3) \wedge (t_{1,2} \geq t_{2,2} + 1 \vee t_{2,2} \geq t_{1,2} + 1) \wedge \\ (t_{1,2} \geq t_{3,2} + 3 \vee t_{3,2} \geq t_{1,2} + 1) \wedge (t_{2,2} \geq t_{3,2} + 3 \vee t_{3,2} \geq t_{2,2} + 1) \end{aligned} \tag{1}$$

[2]SCT and the examples are available for download from http://tiny.cc/sctool.

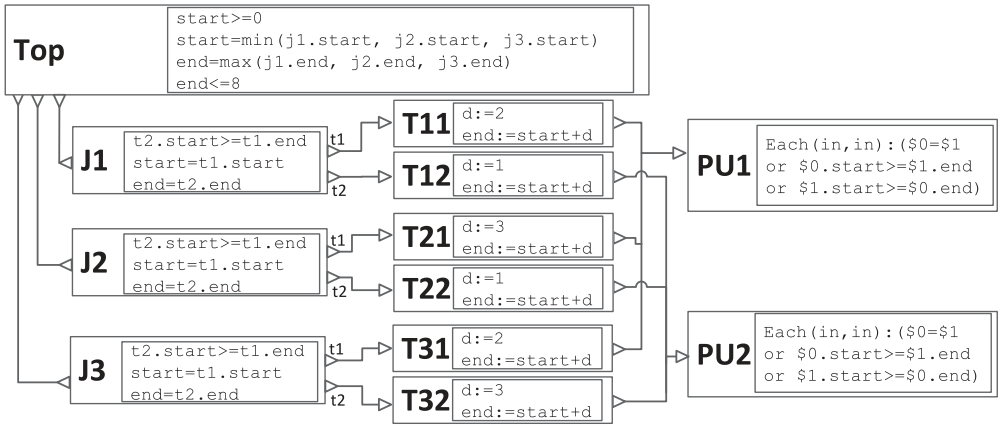


Fig. 5. CoDeL description of De Moura's scheduling use case. The top component initializes three jobs (J1..3), which use two tasks each (T11..T32), which are mapped to two processing units.

In Equation (1), $t_{j,t}$ represents the start time for the task t for job j. This example shows the strength of SMT in combining propositional logic and standard arithmetic. Modern solvers solve the program instantaneously, returning the solution $t_{1,1} = 5,\ t_{1,2} = 7,\ t_{2,1} = 2,\ t_{2,2} = 6,\ t_{3,1} = 0,\ t_{3,2} = 3$. While Equation (1) delivers the correct result the equation is not intuitive to derive or maintain. For comparison we express the same example in CoDeL. Based on the component-oriented understanding of the system (tasks, jobs, process units), a representation of the system in CoDeL is shown in Figure 5, which shows the jobs (J1..J3), the tasks (T11..T32), and the two PUs. Each component only expresses relations and assumptions important in the scope of the component. The jobs only ensure that their task 2 starts after task 1 is finished. The top component sets the assumption that all tasks finish within 8 time units. The PUs ensure that for each pair of tasks only one is active at a time. SCT encodes the system following the rules stated in Algorithm 3, and after executing the SMT solver we obtain a parameterized *start* variable for each task.

Within SCT, the design of this example is further supported by loadable templates of jobs, tasks, and PUs, which only need to be parameterized, while tasks are linked to actual software code and PUs to hardware modules. We discuss the performance as well as the usability of the example in more detail in Section 5.5.

5.3. XGRID

The XGRID many-core architecture [Gunes and Givargis 2014] requires the mapping of software processes on a reconfigurable hardware platform. In this section we show how we can express the problem in CoDeL and compare the results of our approach to the ILP solution presented by Gunes.

XGRID is an embedded many-core processor platform that integrates processing cores and an FPGA-like interconnection network. XGRID uses rows and columns of buses with programmable switching fabrics at the intersections of the row/column buses to route the input/output of logic-blocks. The XGRID interconnect is compile-time configurable for a specific software application. An instance of the XGRID interconnection network with two rows and two columns is shown in Figure 6(a). The figure shows two row and column buses (rails) in the interconnect network, represented as thick lines. Appropriate switches (shown as X boxes) need to be set to establish a communication channel between a pair of cores.

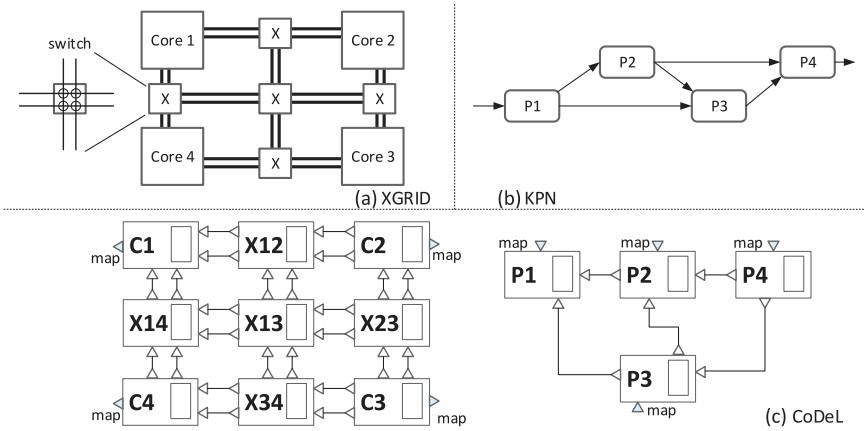


Fig. 6. 2x2 XGRID architecture (a), four-process KPN programming abstraction (b), and CoDeL representation (c). Processes have to be mapped to cores via the map interface.

Design problem. We want to map applications that are expressed as a Kahn Process Network (KPN). A KPN with four processes is shown in Figure 6(b). Arrows between processes indicate communication between the connected processes. In the KPN, each process has to be mapped to one core and each communication channel between two processes corresponds to a connection between the two cores. The connection is not direct but has to be directed through the configurable interconnect network and its switches. The questions to be answered are, first, how to map the processes of the KPN to the PUs, and second, how to configure the interconnect network.

Setup. To answer the questions, we configure XGRID in CoDeL as shown in Figure 6(c). The XGRID system contains three kinds of entities: the cores (fixed netlist), the switches (parameterizable VHDL models), and the processes of the KPN (C code). Both, hardware blocks and software processes are modeled as components with the *map* interface for the mapping. The entities have the following property models:
The *process* has a map interface to cores, and data interfaces to neighbored processes (d1, d2, ...). The properties of a process include a unique id and the constraint that the set of required processes (ids of the neighbored processes) is a subset of the reachable (provided) ids by the mapped core. Therefore the property model of the process is the following.

```
unique id
require:=Union(d*.id)
require<=map.provide
```

The idea of the *interconnect network* is that each active connection must connect two cores without interruption. Further, each rail must be used by one pair of cores only. Therefore we store the two ids of the two cores in the variables id1 and id2, which are propagated by the switches. If a rail is not used, id1 and id2 are 0.
A *core* has two rules: first, a relation that aggregates the ids of all rails into the *provide* property, which is accessible by the processes via the map interface:

```
provide:=Union(p*.id1) ∪ Union(p*.id2),
```

and, second, the assumption that for each interface: either the connected rail contains the local process id—or the rail is disabled (both 0)

```
(p1.id1=map.id) OR (p1.id2=map.id) OR ((p1.id1=0) AND (p1.id2=0)).
```

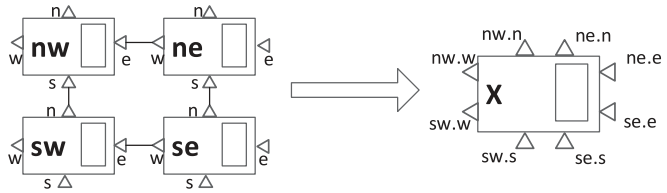


Fig. 7. Hierarchical composition of a 2x2 XGRID switch.

Switches. A single one-rail switch has four interfaces and can connect two or no interfaces, while isolating the other interfaces. Therefore the switch can be configured with a variable (x) that can be set to one of seven settings (off, North-East, NW, NS, SW, SE, EW). The incoming ids are stored in a variable (id1), and assigned to the active outgoing interfaces based on the setting of x. For id1 the property model is the following.

```
id1:=pe.id1 ∪ pn.id1 ∪ pw.id1 ∪ ps.id1
pn.id1:=((x=NW)|(x=NS)|(x=NE))?id1:[0]
pw.id1:=((x=NW)|(x=SW)|(x=EW))?id1:[0]
pe.id1:=((x=NE)|(x=SE)|(x=EW))?id1:[0]
ps.id1:=((x=SW)|(x=SE)|(x=NS))?id1:[0]
```

Larger switches can be built by aligning and connecting four (or 9, or 16) 1x1 intersections and compiling a grouped component as illustrated in Figure 7. The composed 2x2 switch maintains the internal properties and switch settings (ne.x, nw.x, se.x, sw.x), but is easier to handle and reuse within SCT.

Experiments and discussion. To test the setup, we used the benchmarks provided in Gunes and Givargis [2014], that is, parallel matrix multiplication (MMUL), discrete cosine transformation (DCT), and distributed sort (SORT). The initial KPN for the programs was already adapted to utilize the 4, 16, 64, or 256 core setup. We used fixed XGRID architectures with 1, 2, and 3 rails, generated the SMT programs for each mapping problem, solved the systems with Z3 and applied the result of the solution to the implementation files used by the XSIM simulator. The simulator confirmed the correctness of the generated mappings and the performance as in the original work.

Compared to the ILP approach, we see three major advantages for our proposed approach. First a faster run-time of the SMT solver, and the ability to constrain the size of the interconnect network. We could successfully map DCT and MMUL on XGRID architectures with two rails, while SORT could even be mapped on a single rail architecture. The ILP could not express such constraints and required six rails between the cores. Second, our work already results in the settings for each intersection point (the x) that can be automatically applied to the VHDL files. The original work required a manual post-processing step to find feasible routes. Third, we state an improved usability and extensibility of components and systems due to the graphical notation of CoDeL. System architecture properties can be easily changed, while the ILP presented in Gunes and Givargis [2014] consists of a complex system of equations. A disadvantage of our work is the missing optimization of the energy consumption that is part of the ILP solver. The ability of SMT to optimize systems is ongoing research as discussed in Section 5.5.

5.4. Mapping of Control Applications

This section addresses the mapping of control applications on available processing units (PUs). In traditional control design, first the CA is developed and tight constraints for

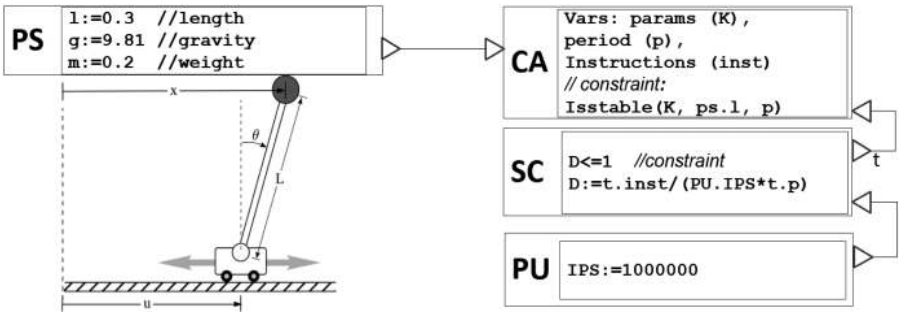


Fig. 8. Inverted Pendulum use case modeled with Physical System (PS), Control Algorithm (CA), Scheduler (SC), and Processing Unit (PU).

the physical and cyber implementation are set [Derler et al. 2013]. While such design contracts can be easily implemented in CoDeL, for our example we want to go one step further and analyze the stability of the system while looking for design alternatives of the computation platform and the parameterization of the physical system. Therefore, first, we show in detail how to model and evaluate the dynamics and stability of a single inverted pendulum system. In the second part we extend the example to a mapping of parallel control systems on a shared distributed computer platform.

5.4.1. Inverted Pendulum Use Case. Next we demonstrate how the nontrivial overfunctional attribute, *stability*, of an inverted pendulum control system can be expressed, evaluated, and packaged as a reusable component in CoDeL. The one degree of freedom case of the system is illustrated on the left of Figure 8. The goal of the system is to produce an appropriate control command u (the cart position) to keep the pendulum in the upright position. Mirzaei et al. [2015] discuss the example in more detail. In the first analysis step we are interested in the stability of the Control Algorithm (CA) for the given Physical Subsystem (PS) and the resource-limited cyber system.

Setup. Figure 8 shows the CoDeL setup for a small estimation system, including variables, describing the length, mass, and gravity of the PS and instructions per seconds (IPS) for the PU. The scheduler (SC) assumes tasks arriving with a fixed period and a number of instructions per instance. The variables of interest in this system include the parameters of the PS, the IPS of the PU, and the sampling rate (period) of the CA. The two constraints in the system concern schedulability and stability. The schedulability can be easily answered since the system is schedulable if and only if the density $\Delta = instructions/period/IPS \leq 1$. More challenging is the assessment of stability, for which the function *isstable* has to be modeled appropriately.

Model of stability. Using the variables shown in Figure 8, the kinetic equations can be stated as $\ddot{x} = \frac{g}{L}(x - u)$, for which the state model $\dot{\vec{x}} = \begin{pmatrix} \dot{x} \\ \ddot{x} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ \frac{g}{L} & 0 \end{pmatrix}\begin{pmatrix} x \\ \dot{x} \end{pmatrix} + \begin{pmatrix} 0 \\ -\frac{g}{L} \end{pmatrix}u$ can be derived. To stabilize the pendulum, the CA uses the state feedback method $(u = -\begin{pmatrix} k_1 & k_2 \end{pmatrix}\begin{pmatrix} x \\ \dot{x} \end{pmatrix} = -K\vec{x})$, for which LQR theory determines the values of K. To test the stability for given L, g, K and period p in CoDeL we have three general options.

(1) *Invocation of a simulation environment.* CoDeL can resolve the *isstable* function by invoking an executable simulation, for example, from Simulink. For our example we instantiate a Simulink model consisting of 20 Simulink blocks as a black box in CoDeL. The Simulink model was designed by control experts to determine the stability of a control system with high accuracy. In SCT, the Matlab function is

called with the relation `isstable=matlab(isstable(ps.l,p))`. The `matlab` function resolves the communication with the Matlab program and defines the `isstable` property. By invoking the model in CoDeL we can combine stability with system attributes such as platform, scheduling, or run-time. A clear disadvantage of this approach is that the run-time does not scale with larger design spaces.

(2) *Derivation of a stability function.* We can also apply knowledge from control experts to express the stability as a static equation. Using the Schur-Cohn algorithm [Stoica and Moses 1992], we can conclude that a system can be successfully discretized if the eigenvalues of matrix F_{cl} reside in a unit circle, enforcing the constraint

$$|det(F_{cl})| < 1 \bigwedge |trace(F_{cl})| < 1 + det(F_{cl}) \tag{2}$$

with
$F_{cl} = \begin{pmatrix} C + k_1(C-1) & \Omega^{-1}S + k_2(C-1) \\ \Omega S + k_1 \Omega S & C + k_2 \Omega S \end{pmatrix}$; $\Omega = \sqrt{\frac{g}{L}}$, $C = \cosh(p\Omega)$, $S = \sinh(p\Omega)$.
The resulting constraint (2) is the stability criterion as a set of algebraic (in)equalities, with parameters L and p that can be expressed as reusable components in CoDeL.

(3) *Discretization of the stability function.* Since Equation (2) contains nonlinear operations that are not supported by all solvers, we alternatively can compute the trajectory of the stability function offline. The result is a table that describes the design space over n fixed regions $(h_{i,1}, h_{i,2}) \times (p_{i,1}, p_{i,2})$, which can be expressed as a conjunct set of assertions: $isstable(h, p) = (h_{1,1} \leq h \leq h_{1,2}) \wedge (p_{1,1} \leq p \leq p_{1,2}) \bigwedge \cdots \bigwedge (h_{n,1} \leq h \leq h_{n,2}) \wedge (p_{n,1} \leq p \leq p_{n,2})$ which can be expressed as one long conjunct statement in the property model of CA.

Experiments and discussion. As one result of this short survey, we obtained three reusable blocks in CoDeL—all designed by control experts—to assess *isstable*. This demonstrates the expressiveness of CoDeL to describe complex over-functional attributes. To test the stability functions, we executed experiments with 100 physical systems (pendulum length 0.01m to 1.0m) and 100 sampling rates (1ms to 100ms). Taking the simulation results as benchmarks, the nonlinear stability function is 99.9% accurate but is three orders of magnitude faster. The discretized models with 20 fix points is 99.0% correct and is about five orders of magnitude faster than the simulation. Based on the beneficial trade-off, we applied the discretized approach for the full mapping example described next.

5.4.2. Mapping of Distributed Control Systems. In this section we extend the single pendulum control system to a system of distributed control applications that should be mapped on available processing units (PUs). The experiment is inspired by the work in Aminifar et al. [2013]. We select a set of control applications (inverted pendulum, falling ball, servos), which each require a physical subsystem and a control implementation. The control applications, expressed as KPN, have to be mapped to a shared computation platform. The goal for each system is to find a suitable set of PUs (selection), map the tasks to the PUs (binding), and parameterize the sampling rate and intercommunication network to satisfy the timing and resource requirements of the CAs.

The example reuses the concepts described in the previous sections: jobs are expressed as KPNs, each of the tasks of the jobs needs a specific set of instructions and has to be mapped on a platform consisting of PUs and interconnects (see XGRID example). The intertask constraints are evaluated as shown in Figure 5, that is, we assume an earliest deadline first (EDF) scheduler with a set of tasks running at a constant sampling rate. As a schedulability test we can evaluate the density of the combined

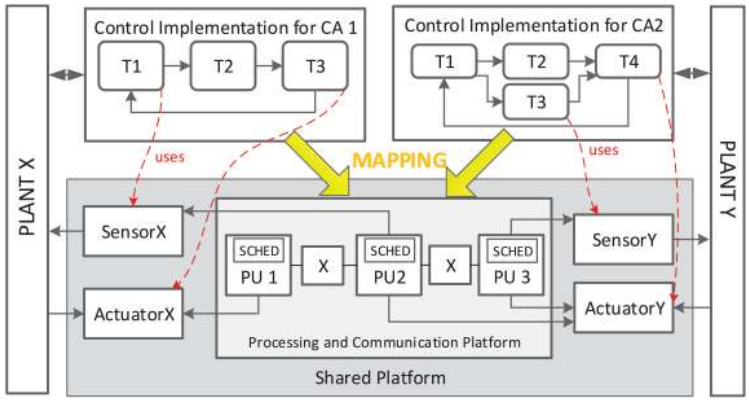


Fig. 9. Example for a mapping problem. Two control applications (CA1, CA2), implemented as a sequential set of tasks should be mapped on shared processing units (PU1-3) and its interconnect network (X). Tasks need access to interfaces that can only be provided by a subset of the PUs (indicated by red arrows).

Table I. Performance Results for the Mapping Test Cases

CAs,Tasks,PUs	INT1	INT2	**SMT**	assertions
5, 25, 4	240 sec	1 sec	<1 sec	510
7, 50, 5	-	5 sec	1 sec	2100
9, 100, 6	-	220 sec	14 sec	8700

task set: $\Delta = \sum_{i=1}^{n} c_i/p_i \leq 1$, where c_i is the computation time and p_i is the constant period [Zhang and Burns 2009]. The stability of the CA is modeled as a discretized stability function.

As a major addition in this example, tasks may require access to physical interfaces, such as sensors (e.g. angle encoder) and actuators (e.g. motor). Therefore, each task states the needed type of interface, and each PU has a list of physical interfaces (sensors, actuators) it can serve.

An example for two CAs (with three and four tasks) and three PUs is illustrated in Figure 9. In the figure, Task T3 from CA1 needs the left actuator, while T1 needs the specific sensor. This relation is illustrated by the thin red arrow.

Experiment and Discussion. For the experiment, we generated random design spaces of variable size, physical properties, PU utilization rate, and resource contention, utilizing the available set of components stored in the project repository. The tests were performed with a 60% PU system utilization rate, which corresponds to the rate in similar industrial applications [Zhu et al. 2012]. To evaluate the performance and scalability of our approach, we compared systems of variable sizes with three approaches: *(INT1)*, a classic backtracking algorithm, that assembles a system along the interfaces and assesses system properties for complete systems, *(INT2)*, which utilizes an early conflict detection mechanism to identify conflict clauses in the property model early in the search, and *(SMT)*, the encoding and solving with the Z3 SMT solver.

We conducted experiments containing design spaces of medium size (25 tasks) up to a relatively large complexity (100 tasks). The generation of the SMT program required about 5 seconds for the largest of the tested systems, which included 8700 assertions. The required time to solve the mapping problem and the number of assertions as part of the SMT program are shown in Table I. The results confirm that the SMT solver clearly has the best performance and shows the best scalability. INT2 is still two orders of magnitude faster than INT1, which could not finish the search for the larger

problems within the five-minute time limit we set for the tests. However, INT2 is still significantly slower than the SMT solver. The advantage of the internal algorithm, compared to an external solver, is the improved user experience, since the internal solver allows direct interactive analysis and an improved conflict analysis. In contrast, the output from the SMT solver is often limited to SAT (yes) or UNSAT (no) with a set of assignments or error clause, respectively.

5.5. Discussion

The examples discussed in this section, first, demonstrate various aspects of the application of SCT, CoDeL, and SMT, and second, facilitate a study of the desired criteria usability, expressiveness, reusability, and scalability. In this section, we summarize the results and discuss strengths and weaknesses of our approach.

Usability. One claim of our work is the easier application of CoDeL compared to a direct description of solver programs. While the choice of a high-level abstraction naturally benefits this claim, we conducted an experiment to gain objective data in this matter. We asked 21 students and researchers in the area of embedded systems to understand and extend a system model described in SMT and CoDeL. For the small scheduling use case (see Section 5.2) the requested modifications are: Job 3 has an additional task (T33) that should be mapped to PU1. The delay of T33 is 3. The total allowed time should be 10.

One correct addition for the SMT Equation (1) is

$$t_{3,3} \geq 0 \wedge t_{3,3} \geq t_{3,2} + 3 \wedge 10 \geq t_{1,2} + 1 \wedge 10 \geq t_{2,2} + 1 \wedge 10 \geq t_{3,3} + 3 \wedge \\ (t_{1,1} \geq t_{3,3} + 3 \vee t_{3,3} \geq t_{1,1} + 2) \wedge (t_{2,1} \geq t_{3,3} + 3 \vee t_{3,3} \geq t_{2,1} + 3). \tag{3}$$

The changes in Equation (3) are minimal, but require in-depth knowledge of the equation, and significant attention in setting the indices.

The required modifications of the CoDeL model in Figure 5 are more practical:

—duplicate one task (T32) and name it to T33;
—add a new port t3 to J3 and add support for t3 in the property model (`t3.start>=t2.end`)—or replace J3 with a three-task job from the repository;
—connect T33 to J3 and PU1;
—change the allowed end time in Top from 8 to 10.

Using feedback from participants, we determined that more than 80% of the responses had the correct adaptation of the CoDeL model, requiring between 1 and 5 minutes to complete the task. In spite of the good results, most answers stated uncertainty on the semantics of CoDeL. In contrast, the equation-based model had the preferred semantics, but less than 30% of the responses were correct, and the stated required time varied between 4 and 15 minutes. The results of this study confirm that even scholars who are trained to express their problems in the form of mathematical clauses, are significantly more confident in expressing the problem space in the form of components, properties, and constraints—as provided by CoDeL. The lack of semantics in CoDeL at this point of development is expected and has been discussed in Section 3.3.

Reusability. Improved usability facilitates extensibility and reusability of existing systems and components. In our examples we showed three kinds of reusability.

—*Reusability of components and models developed by external domain experts*. With the pendulum use case, we showed how complex overfunctional assessment models can be used by system engineers as black boxes in CoDeL. This property is required to enable the separation of concerns, needed for large systems.

—*Extensibility of a system*. The XGRID example showed, step by step, how a large system can be gradually developed and studied starting from small parts of the system, concluding with a complete implementation.
—*Reusability of components between projects*. The distributed control application example reuses components from the XGRID, pendulum, and scheduler examples and extends them with a shared resource model.

All three kinds of reusability are essential for the development of large systems, for which we could demonstrate an advancement of the practical application compared to mathematical languages. In this regard, however, one disadvantage of CoDeL at this point of development is the lack of guidelines and catalogs of supported components and properties, which has to be addressed in order to foster reusability in practice.

Expressiveness. In particular, the pendulum example demonstrated alternatives in the expression of nontrivial structural and overfunctional properties, based on the concepts introduced in Section 3.3. While so far only a few models are practically implemented in CoDeL, we can reuse a large set of existing equation-based and simulation-based models (see Section 3.3.2). The models are exposed to model-specific precision-to-effort-trade-offs, which have to be studied, similar to the pendulum study, case by case in future work.

Scalability. With the XGRID configuration and the distributed mapping examples we could study the performance and scalability of the CoDeL/SMT approach. In particular the discussed real-time system of up to 100 tasks on a many-core platform corresponds in size to typical industrial use cases, for instance, in the automotive industry where dozens of jobs have to be mapped to a network of EPUs [Zhang and Burns 2009]. For these examples we could see that, first, our approach performs significantly better compared to alternative internal solvers, and second, that even very large SMT programs can be parsed and solved within a few seconds. These observations were expected and in fact motivated the work of this article but they also trigger a range of follow-up questions.

First, can we further improve the solving performance with a more efficient encoding? Bjørner [2011] showed that the encoding impacts the solver performance significantly. But so far, no general guidelines or rules are available to define an optimal SMT program. The encoding presented in this article was not subject to optimizations. Algorithm 3 evidently results in an overhead of assertions in the SMT program. For the small scheduling case (Section 5.2), the hand-written SMT program had less than half as many assertions as the generated program (15 to 36). In an extended test, we manually optimized redundant assertions and merged variables for the XGRID use case. In particular, the binding rules (R8) produce many of those invariant equalities to describe the hardware structure, which in XGRID is fixed. Our test could identify only a marginal performance gain ($<2\%$), which originates from the initial input processing and optimization phase of the solver. The actual model finding phase, which entails the complex search process, was not impacted by the reduction of assertions.

A second idea to improve the performance, is to support the solver in finding a solution faster by adding model knowledge, heuristics, and hints. To test this idea for the XGRID case, we enforced the process with the highest connectivity to a core in the center of the system, where the processors with the best connectivity are located. In this experiment, we could reduce the average time to find a solution by up to 15% for the 16-core setup. This outcome was expected since we reduced the complexity of the design space. The risk is, however, that our heuristic is wrong, leading to an

unsatisfiable design situation, or unnecessarily constrains the design space, which would extend the run-time of the solver.

Another reason for the good performance of SMTs, is that satisfiability solvers only look for one satisfiable solution instead of delivering an optimal one.

Model optimization. Even though today's SMT solvers return only one satisfying model, which usually is not the optimal one, SMT solvers can be utilized to find optimal solutions, as for example discussed by Nieuwenhuis and Oliveras [2006]. The general idea is to successively tighten a cost constraint until the solver fails to find a solution. Practically, the idea of optimizing SMTs has been discussed and implemented by Nieuwenhuis and Oliveras [2006], Sebastiani and Tomasi [2012], and Li et al. [2014]. While the first two approaches require specific solvers and input languages, the latter (SYMBA) uses an unmodified Z3 as black box and the SMT-LIB2 syntax, which we use in this article as well. SYMBA supports multiple cost functions, which allows one to cover typical performance and cost trade-offs in the design. While the practicality of using SMT optimizers still has to be studied in future work, SYMBA is a very promising step to complement the results of our article: a system expressed in CoDeL and translated into SMT can be directly analyzed and optimized using SYMBA.

6. CONCLUSIONS

In this article we showed how the performance of the Satisfiability Modulo Theory (SMT) can be systematically harnessed to solve general system synthesis problems in the domain of embedded systems. We described CoDeL—a generic component-based description language to express the structure and properties of individual building blocks of the embedded system, such as tasks, resources, sensors, or actuators. These reusable component models can be leveraged to compose complex designs, respecting the design variabilities such as component selection, component connection, and parameterization. Our proposed SMT encoding algorithm translates system models that can be expressed using CoDeL, to SMT programs that can be solved with state-of-the-art SMT solvers. The key contribution of the encoding scheme is the definition of the design space over possible bindings between the parameterizable blocks. We applied this SMT encoding to scheduling and process mapping examples and demonstrated that our approach can streamline the modeling of design problems and improve the performance in exploring large design spaces.

As a result, we could replace the technicality of solving and analyzing the design space of embedded systems with actual model and system building. Designers may build new systems and add new models on design viewpoints without having to be concerned about how to solve them or how to describe the constraint programs.

REFERENCES

Siddharth Agarwal and Amey Karkare. 2013. Functional SMT solving with Z3 and racket. In *Proceedings of the 1st FME Workshop on Formal Methods in Software Engineering (FormaliSE)*. 15–21.

Amir Aminifar, Petru Eles, Zebo Peng, and Anton Cervin. 2013. Control-quality driven design of cyber-physical systems with robustness guarantees. In *Proceedings of Design, Automation and Test in Europe (DATE)*.

Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. 2013. The satisfiability modulo theories library (SMT-LIB). *www.SMT-LIB.org*.

Victor Berman. 2006. Standards: The P1685 IP-XACT IP metadata standard. *IEEE Des. Test Comput.* 23, 4, 316–317.

Nikolaj Bjørner. 2011. Engineering theories with Z3. In *Proceedings of the 9th Asian Symposium on Programming Languages and Systems (APLAS)*. Springer, 4–16.

Paraskevas Bourgos, Ananda Basu, Marius Bozga, Saddek Bensalem, Joseph Sifakis, and Kai Huang. 2011. Rigorous system level modeling and analysis of mixed HW/SW systems. In *Proceedings of the ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*.

Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. 2010. The openSMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 150–153.

Michael R. Bussieck and Alex Meeraus. 2004. General algebraic modeling system (GAMS). In *Modeling Languages in Mathematical Optimization*. Springer, 137–157.

Werner Damm, Angelika Votintseva, Alexander Metzner, Bernhard Josko, Thomas Peikenkamp, and Eckard Böde. 2005. Boosting reuse of embedded automotive applications through rich components. In *Proceedings of Foundations of Interface Technologies (FIT)*.

Luca De Alfaro and Thomas A. Henzinger. 2001a. Interface automata. *ACM SIGSOFT Softw. Eng. Notes* 26, 5, 109–120.

Luca De Alfaro and Thomas A. Henzinger. 2001b. Interface theories for component-based design. In *Embedded Software*. Springer, 148–165.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: Introduction and applications. *Commun. ACM* 54, 9, 69–77.

Patricia Derler, Edward Lee, Martin Törngren, and Stavros Tripakis. 2013. Cyber-Physical System Design Contracts. In *Proceedings of the ACM/IEEE International Conference on Cyber-Physical Systems*.

Juraj Feljan, Luka Lednicki, Josip Maras, Ana Petricic, and Ivica Crnkovic. 2009. Classification and survey of component models. Tech. Rep. ISSN 1404-3041 ISRN MDH-MRTC-242/2009-1-SE. http://www.es.mdh.se/publications/254-

Robert Fourer, David M. Gay, and Brian W. Kernighan. 1989. AMPL: A mathematical programming language. In *Algorithms and Model Formulations in Mathematical Programming*, Springer, 150–151.

Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. 2009. *Embedded System Design: Modeling, Synthesis and Verification*. Springer Science & Business Media.

Matthias Gries. 2004. Methods for evaluating and covering the design space during early design development. *Integration VLSI J.* 38, 2, 131–183.

Volkan Gunes and Tony Givargis. 2014. XGRID: A scalable many-core embedded processor. In *Proceedings of the IEEE International Conference on Embedded Software and Systems (ICESS)*.

Christine Hang, Panagiotis Manolios, and Vasilis Papavasileiou. 2011. Synthesizing cyber-physical architectural models with real-time constraints. In *Computer Aided Verification*, Springer, 441–456.

Christian Haubelt and R. Feldmann. 2003. SAT-based techniques in system synthesis. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 1168–1169.

Julien Henry, Mihail Asavoae, David Monniaux, and Claire Maïza. 2014. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In *Proceedings of the Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*. 43–52.

Joachim Keinert, Martin Streubühr, Thomas Schlichter, Joachim Falk, Jens Gladigau, Christian Haubelt, Jürgen Teich, and Michael Meredith. 2009. SystemCoDesigneran automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Trans. Des. Autom. Electron. Syst.* 14, 1.

Pratyush Kumar, Devesh B. Chokshi, and Lothar Thiele. 2013. A satisfiability approach to speed assignment for distributed real-time systems. In *Proceedings of the Conference on Design, Automation, and Test in Europe*. 749–754.

Luka Lednicki, Jan Carlson, and Kristian Sandström. 2013. Model level worst-case execution time analysis for IEC 61499. In *Proceedings of the 16th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*.

Philip Levis, Sam Madden, Joseph Polastre, et al. 2005. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, Springer, Berlin, 115–148.

Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. 2014. Symbolic optimization with SMT solvers. In *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 607–618.

Weichen Liu, Zonghua Gu, Jiang Xu, Xiaowen Wu, and Yaoyao Ye. 2011. Satisfiability modulo graph theory for task mapping and scheduling on multiprocessor systems. *IEEE Trans. Parallel Distrib. Syst.* 22, 8, 1382–1389.

Martin Lukasiewycz, Michael Glaß, Christian Haubelt, and Jürgen Teich. 2008. Efficient symbolic multi-objective design space exploration. In *Proceedings of the Asia and South Pacific Design Automation Conference*.

Panagiotis Manolios and Vasilis Papavasileiou. 2013. ILP modulo theories. In *Computer Aided Verification*, Springer, 662–677.

Jose A. Martin, Fabio Martinelli, Ilaria Matteucci, Ernesto Pimentel, and Mathieu Turuani. 2014. On the synthesis of secure services composition. In *Engineering Secure Future Internet Services and Systems*, 140–159.

Hamid Mirzaei, Steffen Peter, and Tony Givargis. 2015. Including variability of physical models into the design automation of cyber-physical systems. In *Proceedings of the Design Automation Conference (DAC)*.

Nina Mühleis, Michael Glaß, Liyuan Zhang, and Jürgen Teich. 2011. A Co-simulation approach for control performance analysis during design space exploration of cyber-physical systems. *SIGBED Rev.* 8, 2, 23–26.

Himanshu Neema, Zsolt Lattmann, Patrik Meijer, James Klingler, Sandeep Neema, Ted Bapty, Janos Sztipanovits, and Gabor Karsai. 2014. Design space exploration and manipulation for cyber physical systems. In *Proceedings of the Workshop on Design Space Exploration of Cyber-Physical Systems (IDEAL)*.

Robert Nieuwenhuis and Albert Oliveras. 2006. On SAT modulo theories and optimization problems. In *Theory and Applications of Satisfiability Testing-SAT*, Springer, 156–169.

Pierluigi Nuzzo, Huan Xu, Necmiye Ozay, John B. Finn, Alberto L. Sangiovanni-Vincentelli, Richard M. Murray, Alexandre Donz, and Sanjit A. Seshia. 2014. A contract-based methodology for aircraft electric power system design. *IEEE Access,* 2, 1–25.

Steffen Peter, Krzysztof Piotrowski, and Peter Langendorfer. 2008. In-network-aggregation as case study for a support tool reducing the complexity of designing secure wireless sensor networks. In *Proceedings of the 33rd IEEE Conference on Local Computer Networks (LCN)*.

Felix Reimann, Michael Glaß, Christian Haubelt, Michael Eberl, and Jürgen Teich. 2010. Improving platform-based system synthesis in the presence of stringent real-time constraints. In *Proceedings of the Design Automation Conference (DAC)*.

Felix Reimann, Martin Lukasiewycz, Michael Glass, Christian Haubelt, and Jürgen Teich. 2011. Symbolic system synthesis in the presence of stringent real-time constraints. In *Design Automation Conference (DAC)*.

Alberto Sangiovanni-Vincentelli and Grant Martin. 2001. Platform-based design and software design methodology for embedded systems. *IEEE Des. Test Comput.* 18, 6, 23–33.

Bernard Schmidt, Carlos Villarraga, Jörg Bormann, Dominik Stoffel, Markus Wedler, and Wolfgang Kunz. 2013. A computational model for SAT-based verification of hardware-dependent low-level embedded system software. In *Proceedings of ASP-DAC*.

Roberto Sebastiani and Silvia Tomasi. 2012. Optimization in SMT with LA(Q) Cost Functions. In *Automated Reasoning*, Springer, 484–498.

Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković. 2008. A component model for control-intensive distributed embedded systems. In *Proceedings of Component-Based Software Engineering (CBSE)*. 310–317.

Timothy E. Sheard. 2012. Painless programming combining reduction and search: Design principles for embedding decision procedures in high-level languages. *ACM SIGPLAN Notices* 47, 9, 89–102.

Simulink. 2013. *Simulation and Model-Based Design*. http://www.mathworks.com/products/simulink/.

Petre Stoica and Randolph L. Moses. 1992. On the unit circle problem: The Schur-Cohn procedure revisited. *Signal Process.* 26, 1, 95–118.

Nikola Trcka, Martijn Hendriks, Twan Basten, Marc Geilen, and Lou Somers. 2011. Integrated model-driven design-space exploration for embedded systems. In *Proceedings of Embedded Computer Systems (SAMOS)*.

A. Vachoux, C. Grimm, and K. Einwich. 2003. SystemC-AMS requirements, design objectives and rationale. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 388–393.

Fengxiang Zhang and Alan Burns. 2009. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Trans. Comput.* 58, 9, 1250–1258.

Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. 2001. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 279–285.

Qi Zhu, Haibo Zeng, Wei Zheng, Marco DI Natale, and Alberto Sangiovanni-Vincentelli. 2012. Optimization of task allocation and priority assignment in hard real-time distributed systems. *ACM Trans. Embed. Comput. Syst.* 11, 4 (2012), 85.

Received September 2014; revised December 2014, February 2015; accepted March 2015