CSL4P: A Contract Specification Language for Platforms

Alessandro Pinto and Alberto L. Sangiovanni Vincentelli

October 19, 2016

Abstract

The contract-based design formalism supports compositional design and verification, and generalizes many other languages where components are defined in terms of their assumptions and guarantees. Most languages and tools for contract-based design allow defining, instantiating and connecting contracts, but fall short in capturing families of potential architectures in a flexible way. This article presents a *Contract-Based Specification Language for Platforms* (CSL4P). A platform comprises a set of contract types and a set of constraints called rules. Contract types can be instantiated and connected to form platform instances. While the meaning of composition is predefined in most languages, composition rules are used in CSL4P to provide a finer control on the semantics of interconnections. The separation of contract types and rules also allows defining different platforms out of the same set of components. This article describes syntax and semantics of the language, a development environment which includes a compiler and a verification back-end, and an application example.

1 Motivation, Background and Contributions

A design process defines a sequence of design, validation and verification activities, as well as guidelines, rules and best practices to help avoiding common pitfalls that may turn into serious design errors. Design processes become essential in large systems not only to manage complexity arising from the large number of components, but most importantly to manage their interactions through controlled interfaces. This is crucial because components and subsystems may span across various disciplines, groups, and organizations.

For the past four decades, system engineering processes have been the subject of standardization efforts. The first military standard, the MIL-STD-499, appeared in 1969 and was revised in 1974 [1]. The MIL-STD-499B [2] is the most recent version that outlines a total system approach for the development of defense systems. Industry standards such as ANSI/GEIA 632 [3] and ISO/IEC 15288 [4] have also been adopted for product development. These standards define processes and tasks to be performed during the entire product life-cycle, including the design phase.

In all these standards, the suggested design process follows the V-model [5] that embodies a water-fall approach to design. This approach is not effective when dealing with large and/or complex systems yielding long re-design cycles that delay the final version of the system as well as increase development costs. Indeed, for software intensive systems, it has been already reported [6] that the V-model leads to problems as the majority of design errors are introduced in the early stages of the design process but only discovered late, during integration and testing. To reduce the chances that these types of problems occur, decisions made in the early stages should be well-informed and early verification should be favored. One way to achieve this goal is through the use of models at higher levels of abstraction that capture properties such as performance and cost, as well as interfaces. These models can be used to check system-level properties or to explore alternative architectural solutions for the same set of requirements. If model fidelity is appropriate, then the results of these high level analyses will still be valid once the actual components are integrated, avoiding the long re-design cycles we mentioned above. However, this approach requires a virtual engineering environment populated with validated models. The development and validation of models is an additional effort but it can be done once and amortized over the design of many products. A virtual engineering environment helps designers to manage models, run analysis and optimization tools and find potential sources of problems. A *formal language* for describing components and rules that must be followed to integrate them into systems is at the core of such computer-assisted design process. The use of a language with formal semantics is also key to enabling inter-operability of tools.

The guiding principles for the definition of the Contract-Based Specification Language For Platforms (CSL4P) come from several observations. For example, widely distributed systems such as swarm systems [1] must be designed in a hierarchical and compositional way and languages that support these design styles are essential parts of the arsenal of the design tools. Further, the design of complex cyber-electro-mechanical systems requires mixing skills and expertise coming from different people who contribute to the same design artifact. Thus, to be effective in serving the needs of system designers, a formal language must satisfy at least the following important requirements:

R1 It must support multiple abstraction levels to assist designers from specification to prototyping and testing. Each refinement step from one level of abstraction to the next adds details to the design. Models of components at one abstraction level are in formal refinement/abstraction relationship with models of components at the previous/next abstraction level. These relationships guarantee traceability in the design process.

R2 It must support multiple viewpoints to fuse together different aspects of components and systems, or different requirements. Viewpoints can be used to model aspects that belong to different disciplines (e.g. mechanical, controls, electrical) and that are yet related to each other. Viewpoints can also be used to bring together different models such as static and dynamic functional models, and non-functional ones such as cost and performance.

R3 It must support "compositional" design and verification. By compositional design we mean a process that allows partitioning the design effort across its sub-systems or components (top-down). By compositional verification we mean the ability to derive system-level properties from component properties (bottom-up).

R4 It must provide ways to embrace legacy systems. The design of complex systems, such as aircraft, often derives from previous designs that are known to work well. Thus, components are often re-used from one design to the next. Moreover these components are not necessarily formally specified.

R5 It must allow defining and studying product families rather than specific products. Gaining market share requires covering the needs of a wide range of customers. Thus, it makes sense to design a product line rather than a single product. This is the case, for example, of consumer electronics (e.g., the family of Apple computers, the family of Nexus mobile devices etc.).

R6 The language should be user-friendly to support designers within their application domain. At the same time it should be generic enough to support several such domains.

These requirements justify our choice of a language based on two important design paradigms: contract-based design (CBD) [7, 8, 9, 10] (R1-R4) and platform-based design (PBD) [11, 12, 13] (R1,R2,R5). Both paradigms will be reviewed in this article and their features will be linked to the requirements stated above. Requirement R6 is addressed by the concrete syntax and features of the CSL4P language such as the ability to package elements into modules, the use of inheritance, and tool support for editing models. These features distinguish CSL4P from many other languages that could be used for the same purpose. From the expressiveness and ease of use standpoint, modeling in CSL4P rather that in a generic logic-based language is analogous to programming in Java rather than assembly.

1.1 Background

1.2 Platform-Based Design

In PBD, at each step, top-down refinements of high-level specifications are mapped onto bottom-up abstractions and characterizations of potential implementations. Each abstraction layer is defined by a design *platform*, which is the set of all architectures that can be built out of a *library* (collection) of *components* according to *composition rules*. In the *top-down phase* of each design step, we formalize the high-level system requirements and we perform an optimization (refinement) phase called *mapping*, where the requirements are

[1]See http://www.terraswarm.org/ for the TerraSwarm Center, part of the SMARTnet, which has as object the design of these systems

mapped onto the available implementation library components and their composition. Mapping is cast as an optimization problem, where a set of performance metrics and quality factors are optimized over a space constrained by both system requirements and component feasibility constraints. Mapping is the mechanism to move from a level of abstraction to a lower one using the available components in the library. Note that when some constraint cannot be satisfied using the available library components or the mapping result is not satisfactory for the designer, additional elements can be designed and inserted in the library. For example, when implementing an algorithm with code running on a processor, we are assigning the functionality of the algorithm to a processor and the code is the result of mapping the "equations" describing the algorithm into the instruction set of the processor. If the processor is too slow, then real-time constraints may be violated. In this case, a new processor has to be found or designed that executes the code fast enough to satisfy the real-time constraint. In the mapping phase, we consider different *viewpoints* (aspects, concerns) of the system (e.g. functional, reliability, safety, timing) and of the components. In the *bottom-up phase*, we build and model the component library (including both plant and controller).

If the design process is carried out as a sequence of refinement steps from the most abstract representation of the design platform (top-level requirements) to its most concrete representation (physical implementation), providing guarantees on the correctness of each step becomes essential.

To do so, we need to formally prove that: (i) a set of requirements are *consistent*, i.e. there exists an implementation satisfying all of them; (ii) an aggregation of components are *compatible*, i.e. there exists an environment in which they can correctly operate; (iii) an aggregation of components *refines* a specification, i.e. it implements the specification and is able to operate in any environment admitted by it. Moreover, whenever possible, we require the above proofs to be performed *automatically* and *efficiently*, to tackle the complexity of today's CPS. Therefore, to formalize the above design concepts, and enable the realization of system architectures and control algorithms in a hierarchical and compositional manner that satisfies the constraints and optimize the cost function(s), we resort to *contracts*.

1.3 Contracts

The idea of using formal descriptions of interfaces to prove the correctness of programs has been studied for the past 40 years. Among the earliest works, we mention the ones from Floyd[14] and Hoare[15], as well as an interesting essay from Dijkstra[16]. An extensive review of previous work and approaches can be found in [17]. Contracts are a generalization of these ideas to system design. In a contract framework, design and verification complexity is reduced by decomposing system-level tasks into more manageable sub-problems at the component level, under a set of assumptions. System properties can then be inferred or proved based on component properties. Rigorous contract theories have been developed over the years, including assume-guarantee (A/G) contracts [8] and interface theories [18]. However, their adoption in industry is limited. A major challenge is the absence of a comprehensive modeling formalism for CPS that addresses complexity and heterogeneity [9, 10].

Contracts can be used in the PBD framework to prove that the refinement steps are correct as outlined above. In particular, contracts have been used traditionally to specify components, and aggregation of components at the *same level of abstraction*; for this reason we refer to them as *horizontal contracts*. Horizontal contracts can be used to prove that an aggregation of components are *compatible*, i.e. there exists an environment in which they can correctly operate.

We use contracts also to formalize and reason about refinement between two different abstraction levels in the PBD process [19, 10]; for this reason, we refer to this type of contracts as *vertical contracts*. Vertical contracts can be used to prove that : a set of requirements are *consistent*, i.e. there exists an implementation satisfying all of them and an aggregation of components *refines* a specification, i.e. it implements the specification and is able to operate in any environment admitted by it. Hence the combination of vertical and horizontal contracts is an important support to formalize the PBD design flow.

1.4 Existing frameworks

We define an architecture loosely as the interconnection of a certain number of components. Based on this definition, most languages used today in the design of systems are suitable to define architectures. They can be divided into categories based on the types of design tools they primarily enable. Some languages focus mainly on enabling simulation while others are geared towards performance modeling, analysis and verification – design activities that are all essential to a design process for complex systems.

The basic syntax of most of these languages allows capturing hierarchy, components, ports and connections. For example, Simulink [2] provides concepts such as blocks, ports, and signals to connect ports; a group of components can be encapsulated into a sub-system, thereby making Simulink models hierarchical. SystemC [20] also provides concepts like modules – which can be nested hierachically –, ports, and channels. The VHDL [21] language provides very similar constructs. SysML [3] defines the notion of blocks, ports and connectors; the internal block diagram of SysML is a graphical representation of an architecture. A unified abstract syntax of these types of languages can be found in [22] which also defines the abstract syntax of Ptolemy II. The features provided by these tools simplify system modeling and should be implemented by CSL4P (Requirement R6).

The Architecture Analysis and Design Language (AADL) [23] provides features to model hardware and software as well as performance metrics (such as the execution rate of a software task). The AADL language also supports the definition of product families [24], although families are meant to represent few variants of the same product and not really platforms as defined later in this article. Similarly, the SysML language can be used to model properties of blocks such as performance numbers, although the language per se does not provide a definitive semantics for those properties. An interesting feature of SysML is the ability to constrain the way a component is used by adding OCL[4] constraints to a model.

Logic-based languages provide some features to model families of architectures. These languages allow describing components using First Order Logic and then verify when an architecture (i.e. the composition of blocks) is valid. FORMULA [25] and Alloy [26] are examples of these languages. Their approach is similar in spirit to the one presented in this article: a model is translated into a logic formula for which an off-the shelf decision procedure exists (Satisfiability Modulo Theories for FORMULA, and SAT for Alloy). In particular, the FORMULA language follows the platform-based design paradigm by providing all the necessary features to model platforms, namely components and rules that define legal compositions. An interesting approach is the one implemented in the PACELAB tool suite [5] where a component is modeled as a set of constraints on variables (some defined as inputs and some as outputs). Thus, a system becomes a network of constraints between variables. A proprietary solver is used to find a consistent value assignment for all variables in the system, if one exists. This approach is a hybrid: it allows the definition of constraints as in the case of logic-based languages, but constraints are defined in imperative code. The Rosetta [27] language follows an approach very similar to CSL4P, but the language is not supported by tools for modeling and verification. The modeling style for components and connection used by CSL4P also resembles the Modelica [28] language. Interestingly, previous work has investigated modeling systems properties in Modelica [29]. These properties can then be checked in simulation or by formal methods. CSL4P extends the notion of property to contracts, provides a declarative specification language, and uses formal methods rather than simulation. As result, CSL4P addressed the limitations expressed by the authors of [29].

The intent of this section is not to provide a comprehensive list of all the available modeling languages and tools that are at the disposal of a system engineer. It provides, however, a list of representative environments for different classes of modeling approaches. In designing complex systems, engineers make use of several tools to perform simulations, analysis and early validation and verification. Models used by these tools are often different and consistency is maintained manually. Further, the design process is divided into subsequent phases at different levels of abstraction such as requirement definition, preliminary system design and detailed design. Each phase uses different models, and even in this case, consistency is maintained manually. The use of CBD and PBD helps in organizing the design process, enabling compositional verification and design-by-refinement, and linking models by formal relations that can be automatically checked. Thus, a language that supports both paradigms is a key enabler for the design of complex systems.

The need for a new language stems from the following considerations. Languages available today follow either the PBD or the CBD paradigms. Most of the work in the definition of a language for CBD has been theoretical with limited implementation which can be used to prove strength and weakness of the approach. The most relevant implementation is the contract specification language developed under the Speculative

[2]http://www.mathworks.com/products/simulink/
[3]http://www.omg.org/spec/SysML/
[4]http://www.omg.org/spec/OCL/
[5]http://www.pace.de/

and Exploratory Design in Systems Engineering (SPEEDS) project [6]. Moreover, the focus of most available languages has been on capturing architecture instances by instantiating and composing components from a library, which is a repository of models. Unfortunately, this is not sufficient to capture product families or platforms, i.e. special libraries where in addition to components, a set of rules contribute to the definition of admissible architectural solutions. The set of rules come from several stake-holders in the design process such as customers, system architects, and sub-system suppliers. They are, therefore, an essential element in the design of complex systems. A path to develop a language that supports both CBD and PBD could be to reuse an expressive language such as Alloy or FORMULA and enforce a certain modeling style to encode contracts and composition rules. However, this choice would lead to a modeling burden since the notion of contract is not natively supported. Thus, this approach would not be able to satisfy our requirement R6.

It would be challenging to develop a new language that is a superset of all the languages mentioned in this short section while adding support for CBD and PBD. The first step is to be able to capture structural properties (i.e. components and their connections), non-functional properties such as performance and cost, and state invariants (i.e. characterization of sets where the trajectories of a system lie). This is the area that CSL4P enhances with respect to the frameworks listed in this section.

1.5 Contributions

In this paper, we present a contract specification language for platforms called CSL4P. This language belongs to the class of logic-based languages. It follows the platform-based design paradigm as well as the contract-based design paradigm allowing the language to be effectively used in a component-based and design-by-refinement development process.

A *platform* is defined by a set of contract types and a set of composition (or design) rules. Contract types can be instantiated and connected to form a *platform instance* (or *architecture*). The instance of a contract type is also referred to as *contract instance*. A *contract type* specifies the environments in which an actual component implementing the contract can be used and its resulting behaviors. It can be thought of as a "data sheet" that describes a component available from a certain vendor. The component is an actual part used in a system and it can be thought of as the artifact produced by the vendor – who has also validated the component and certified that it adheres to the data sheet. *Rules* define when a composition of contract instances is legal. CSL4P allows modeling two different classes of rules. *Validity* rules are used to express those requirements that are implicit in a specific application domain and that all architectures must satisfy. For example, control units shall never be placed close to a heat source. Validity rules are also used to express system implementation constraints such as the minimum radius of curvature of a pipe. *Assertion* rules are used to model constraints that always hold such as the laws of physics. Assertion rules are true statements in all cases while validity rules can be violated, indicating that the architecture does not belong to the platform. We show that an extension of First Order Logic is required to model aggregate quantities often used in the specification of systems such as the total heat entering a system. We define the syntax and semantics of such rules. Finally, we present a development environment for CSL4P. The development environment includes an editor, a compiler and an SMT solver to check the validity of architectures.

In this article we only consider static models, i.e. models that do not use the notion of time (either discrete or continuous). This extension is left for future work and requires a different mechanism to define composition. The verification of architectures is supported by Satisfiability Modulo Theories solvers. These solvers are becoming capable of answering satisfiability queries on problems that may contain non-linearities [30, 31, 32, 33], and differential equations [34]. Thus, we expect to be able to increase the expressiveness of CSL4P in the future by introducing the notion of discrete and continuous time.

2 Definition of the CSL4P language

An abstract platform for power distribution systems is modeled to explain the usage of the language and its main features. Then, the syntax and semantics of CSL4P are formally defined.

[6] http://www.speeds.eu.com/downloads/D_2_5_4_RE_Contract_Specification_Language.pdf

2.1 CSL4P at a glance

A power distribution system consists of a set of power sources, or generators, a set of power sinks, or loads, a set of storage elements, buses, and "contactors" (or switches) that are used to connect components together and transfer power from generators to loads under different operating and degraded conditions. A generator is connected to a load through a path in the power distribution system (a path that may traverse several buses and contactors). In the preliminary design phase, the topology of the power distribution system can be hidden by abstracting paths into point-to-point connections between power sources and power sinks. Such abstraction could be used to determine the number of generators required in a specific application and the distribution of loads among them.

The *abstract power distribution system platform* contains only two classes of components: generators and loads. These classes are modeled as *contract types* in CSL4P, as shown in Table 1. The contract on the left of Table 1 is the standard model of a non-ideal generator that is the serial connection of an ideal generator and a resistor. The generator type has an `Electrical` view. Parameter R is the internal resistance of the generator while parameter v_0 is its nominal voltage. Variable v and variable i are the voltage and current, respectively. There is no distinction between input and output variables in these models which can be considered acausal. If needed, the direction of variables, as well as the rules defining valid connections among them, must be modeled in the platform. The view contains two sections that define what the component *assumes* about the environment, and what it *guarantees* in return. In this example, the component assumes that the environment does not require a total power greater than $200,000$[7] and guarantees that the output voltage is equal to the nominal voltage minus the drop on its internal resistor.

The contract in the center of Table 1 is an abstraction of a more complex generator component where a controller keeps the output voltage close to the nominal one. Finally, the contract on the right represents a constant power load. This contract assumes that the environment provides the nominal voltage. Under this assumption, the component guarantees to use constant power equal to P. Notice that each contract in this table is a type and not an instance. Instantiating a contract corresponds to renaming its variables. These three contract types together form a library that will be called `GenLoad`. Many instances of these contracts can be created and added to an architecture as follows:

```
architecture A from GenLoad {
        //Component Instances
        Generator $g$ ;
        ConstantPowerLoad $l_1$ ;
        ConstantPowerLoad $l_2$ ;
        //Parameter setting
        $g.R = 1$ ; $g.v_0 = 270$ ; $l_1.P = 10e3$ ; $l_1.v_{nom} = 270$ ;
            $l_2.P = 10e3$ ; $l_2.v_{nom} = 270$ ;
        Connected($g$,$l_1$) ; Connected($g$,$l_2$) ;
}
```

Instances of contract types are created similarly to most programming languages by using a pair consisting of the type name and an identifier. Architecture `A` has three contract instances: a generator `g` and two loads `l1` and `l2`. Notice that variables and parameters are scoped by the contract instances and the dot notation is used to refer to them. Thus, instantiating a contract type corresponds to renaming all its variables (e.g., we have three distinct current variables in architecture `A`: $l_1.i$, $l_2.i$ and $g.i$). The contract instances can be configured by setting their internal parameters. For example, the generator has an internal resistance equal to 1 and a nominal voltage equal to 270. To connect contract instances, the CSL4P language provides a predicate $Connected(s, d)$ where s and d are contract instances. For example, generator g is connected to loads l_1 and l_2. Such predicate does not define what a connection means (in the example above, the two

[7]We will always omit unit of measure to avoid introducing inconsistencies between examples written in pseudo-code and explanations in natural language. CSL4P is expressive enough to capture unit of measure through predicates over terms. In this example, the model could be expanded by introducing the assumption unitOfMeasure(i,$Ampere$), and the guarantee unitOfMeasure(v,$Volt$), where unitOfMeasure is a relation between terms and units of measure. Also notice that assertion rules could be used to define the unit of measure for derived terms. For example, the following rule defines the unit of measure for the product of current and voltage: $\forall\ x$: Real, $\forall\ y$: Real, unitOfMeasure(x,$Ampere$) $\wedge$ unitOfMeasure(y,$Volt$) $\Rightarrow$ unitOfMeasure($i \cdot v$,$Watt$).

connections (g, l_1) and (g, l_2) do not impose any relation among the variables in each instance). Domain experts would interpret such connections as follows: the voltage at the two loads is equal to the voltage at the generator, and the total current provided by the generator is the sum of the currents required by l_1 and l_2. These rules try to capture *facts* that are part of the domain knowledge and that automatically hold; phenomena that are determined by "nature", so to speak. We call these rules *assertions*. The following rule defines the relationship among the voltage at the generator and at the loads of any platform instance:

```
assertion voltageValueRule {
        forall g : Generator, forall l : ConstantPowerLoad,
      Connected(g,l) ⇒ g.v = l.v ;
}
```

An additional assertion rule can be added to the platform to state that connections are bidirectional:

```
assertion connectionsAreBidirectional {
        forall g : Generator, forall l : ConstantPowerLoad,
        Connected(g,l) ⇒ Connected(l,g) ∧ Connected(l,g) ⇒ Connected(g,l)
}
```

Formalisms such as Bond-Graphs [35] incorporate these rules directly as part of the semantics of the language. CSL4P, alike these languages, is based on modeling with constraints, but tries to retain generality by allowing such knowledge to be defined by the user. These rules can, however, be encapsulated into libraries that can be reused.

A different type of rules is used to specify when architectures are considered *valid*. For example, it is not acceptable to have power distribution systems with unconnected generators which bring no performance benefits and only add extra weight making products less competitive. The following rule declares valid only those architectures with no unconnected generators:

```
validity noUnconnectedGenerators {
        forall g : Generator, exists l : ConstantPowerLoad, Connected(g,l)
}
```

We believe that assertion and validity rules provide a flexible mechanism to define the semantics of connections, effectively exposing a fine grained mechanism to define the composition operator over contracts. This flexibility is at the core of the expressiveness of CSL4P.

2.2 Logical framework

In Section 2.1, we provided an informal description of the CSL4P language. Before describing the syntax and semantics of CSL4P formally, we introduce the underlying logical framework. CSL4P is based on Order-Sorted First Order Logic (FOL) with equality. Let the signature of this logic be $\Sigma = (\mathcal{T}, \triangleright, F, P, \Gamma, \sigma)$. $\mathcal{T}$ is a set of sorts or types and $\triangleright \subseteq \mathcal{T} \times \mathcal{T}$, the extension relation, is a partial order over $\mathcal{T}$. F is the set of function symbols, P is the set of predicate symbols and Γ is a set of constant symbols. The sets $\mathcal{T}$, F, P and Γ are disjoint. The function $\sigma : \{F \cup P \cup \Gamma\} \to \mathcal{T}^*$ maps each function, predicate, and constant symbol to its arity (i.e. a sequence of types). For a predicate symbol $p \in P$, $\sigma(p) = (\tau_1, \ldots, \tau_{n(p)})$, and for a function symbols $f \in F$, $\sigma(f) = (\tau_0, \tau_1, \ldots, \tau_{n(f)})$, where τ_i and σ_i are types, and both $n(p)$ and $n(f)$ are strictly positive. For a function symbol f, we also denote $\sigma(f)[0]$ with $\tau(f)$ (where for a sequence $\vec{\tau} = (\tau_1, \ldots, \tau_n)$, we have denoted the i-th element τ_i with $\vec{\tau}[i]$). For a constant symbol c, $\sigma(c) = (\tau)$ is a sequence with one element which is the type of the constant also denoted $\tau(c)$. We will use the notation $c : \tau(c)$ to say that a constant c has type $\tau(c)$.

We now define terms and formulas. We assume that there exists a denumerable set of variable symbols $\mathbb{V}$ and that each variable $v \in \mathbb{V}$ is associated with a type $\tau(v)$. We will use the notation $v : \tau(v)$ to say that a variable v has type $\tau(v)$

A constant term is $c \in \Gamma$ and has type $\tau(c)$. A variable terms is $v \in \mathbb{V}$ and has type $\tau(v)$. A function term is a function symbol $f \in F$ followed by a list of terms $(t_1, \ldots, t_{n(f)})$ where t_i is of type τ_i and $\tau_i \triangleright \sigma(f)[i]$, for $i = 1, \ldots, n(f)$. The function term $f(t_1, \ldots, t_{n(f)})$ has type $\tau(f) = \sigma(f)[0]$.

An atomic expression over Σ is a predicate symbol p followed by a list of terms $(t_1, \ldots, t_{n(p)})$ where t_i is of type τ_i and $\tau_i \triangleright \sigma(p)[i]$, for $i = 1, \ldots, n(p)$. A Σ-formula is defined as follows. All atomic expressions over Σ are Σ-formulas. Let Φ and Ψ be two Σ-formulas and x a variable symbol, then: $\neg\Phi$, $\Phi \wedge \Psi$, and $\forall x : \tau(x).\Phi$ are Σ-formulas (these are also used to define the logical operator $\vee$ and the quantifier $\exists$ in the usual way). We will use the term "formula" rather than Σ-formula to make the presentation less verbose.

A Σ-interpretation μ is a map such that that: (1) each sort $\tau \in \mathcal{T}$ is mapped to a domain of objects D_τ such that if $\tau_1 \rhd \tau_2$ then $D_{\tau_1} \subseteq D_{\tau_2}$; (2) each variable $v \in \mathbb{V}$ is mapped to an object $v^\mu \in D_{\tau(v)}$; (3) each constant $c \in \Gamma$ is mapped to an object $c^\mu \in D_{\tau(c)}$; (4) each function symbol $f \in F$ with arity $(\tau_0, \tau_1, \ldots, \tau_{n(f)})$ is mapped to a function $f^\mu : D_{\tau_1} \times \ldots \times D_{\tau_{n(f)}} \to D_{\tau_0}$; (4) each predicate symbol $p \in P$ with arity $(\tau_1, \ldots, \tau_{n(p)})$ is mapped to a relation $p^\mu \subseteq D_{\tau_1} \times \ldots \times D_{\tau_{n(p)}}$.

Given a Σ-interpretation μ, and a formula over Σ, it is possible to check whether the formula is true or false as follows. An atomic formula $p(t_1, \ldots, t_n(p))$ is true if $(t_1^\mu, \ldots, t_{n(p)}^\mu) \in p^\mu$. Let Φ and Ψ be two formulas and x a variable symbol, then: $\neg\Phi$ is true if Φ is false, $\Phi \wedge \Psi$ is true if Φ is true and Ψ is true, and $\forall x : \tau(x).\Phi$ is true if $\Phi(x \to o)$ is true for all objects $o \in D_{\tau(x)}$ (where $\Phi(x \to o)$ is a formula obtained by replacing occurrences of the bound variable x with o). Finally, a formula is satisfiable if there exists an interpretation that makes the formula true, and is valid if it is true under all interpretations (or if its negation is not satisfiable).

A Σ-theory is a pair $T = (\Sigma, Ax)$ where Ax is a set of Σ-sentences. A Σ-formula is satisfiable in a theory T if there exists a Σ-interpretation μ that satisfies all the sentences in Ax and that makes the formula hold (please refer to [36] for a detailed explanation). Let $\Sigma_1 = (\mathcal{T}_1, \rhd_1, F_1, P_1, \Gamma_1, \sigma_1)$ and $\Sigma_2 = (\mathcal{T}_2, \rhd_2, F_2, P_2, \Gamma_2, \sigma_2)$ be two signatures with disjoint sets of types, function, predicate and constant symbols. Then their union $\Sigma_1 \cup \Sigma_2 = (\mathcal{T}, \rhd, F, P, \Gamma, \sigma)$ is such that $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$, $F = F_1 \cup F_2$, $P = P_1 \cup P_2$, $\Gamma = \Gamma_1 \cup \Gamma_2$, $\sigma|_{F_1 \cup P_1 \cup \Gamma_1} = \sigma_1$ and $\sigma|_{F_2 \cup P_2 \cup \Gamma_2} = \sigma_2$. Given two theories $T_1 = (\Sigma_1, Ax_1)$ and $T_2 = (\Sigma_2, Ax_2)$ over disjoint signatures, their composition is $T = (\Sigma_1 \cup \Sigma_2, Ax_1 \cup Ax_2)$. For decidability results of combination of theories, please refer to [36]. The basic signature defined above shall be considered as the empty theory, or the theory T_{EUF} of equality and uninterpreted functions [37]. We will also use the theory of real numbers $T_\mathbb{R}$ with signature $(\{Real\}, \emptyset, \{+, -\}, \{\leq\}, \Gamma_\mathbb{R}, \sigma_\mathbb{R})$, where the function symbols and predicate symbols have the natural interpretation.

2.3 Abstract Syntax of CSL4P

In this section we define the syntax of platforms and platform instances. We then introduce a second order operator to describe constraints on aggregate quantities that are typically required when defining rules.

Definition 1 (Platform)**.** *A platform is a tuple $\mathcal{P} = (\mathcal{L}, \mathcal{R})$ where:*

- $\mathcal{L} = \{C_1, \ldots, C_n\}$ *is a set of* contract types *(i.e. $C_i \in \mathcal{T}$). A contract type $C_i(V_i, \phi_i^A, \phi_i^G)$ is associated with a set of variables V_i, and two formulas over V_i, ϕ_i^A and ϕ_i^G, called assumption and guarantee. We assume that $\forall C_i \in \mathcal{L}$ and $\forall C_j \in \mathcal{L}$, $C_i \neq C_j \Rightarrow V_i \cap V_j = \emptyset$. For a contract type C, we will also use the notation V_C to denote its set of variables, ϕ_C^A to denote its assumption and ϕ_C^G to denote its guarantee*

- $\mathcal{R}$ *is a set of rules partitioned into a set of* assertion rules $\mathcal{R}_A = \{r_1, \ldots, r_a\}$*, and a set of* validity rules $\mathcal{R}_V = \{r_{a+1}, \ldots, r_{a+v}\}$*.Rules can have two forms: $\forall c : C.\phi(c)$ or $\exists c : C.\phi(c)$ where $C \in \mathcal{L}$ is a contract type and $\phi(c)$ is a formula of the same form or a quantifier free formula. The rest of the variables appearing in a rule are $c.v$ where $v \in V_{\tau(c)}$.*

The set of types $\mathcal{T}$ contains a distinguished element *Contract* such that, for all contract types C that might be defined as belonging to a platform, $C \rhd Contract$. Moreover, we introduce a predicate symbol $Connected \in P$ with arity $(Contract, Contract)$ that is used to specify connections among components. Contract types can be instantiated and connected to form an architecture (or platform instance).

Definition 2 (Platform Instance)**.** *A platform instance from a platform $\mathcal{P} = (\mathcal{L}, \mathcal{R})$ is a tuple $I_\mathcal{P} = (comp, conf, conn)$ where:*

- *$comp = \{c_1, \ldots, c_k\}$ is a set of contract instances with types $\tau(c_i)$.*

- *$conf = \{\pi_1(V_I), \ldots, \pi_m(V_I)\}$ is a set of quantifier free formulas called configuration constraints, where $V_I = \cup_{c \in comp}\{c.v : v \in V_{\tau(c)}\}$ is the set of variables of the platform instance.*

- *$conn \subseteq comp \times comp$ is a set of connections*

A set of connections *conn* is said to be defined over the set of contracts *comp* if for all $(a, b) \in conn$, both a and b belong to *comp*. Relation *conn* is the restriction of the *Connected* predicated over the domain of contracts *comp*.

Differently from most languages, we do not provide any specific semantics for the predicate $Connected(c_i, c_j)$ that simply defines a relation between component instances. Rather, CSL4P provides a general mechanism for defining composition rules. Users of the language can define their own rules using the expressiveness of FOL. However, CSL4P extends the rule language with second order terms of a certain form to allow capturing rules that are common in practice. To understand the reasons for this language extension, consider Kirchhoff's laws for electric circuits and how such rules should be expressed in the power distribution system example: The sum of the currents absorbed by loads connected to a generator must be equal to the current provided by the generator:

$$\forall g : Generator \;.\; g.i = \sum_{l \in Conn_L} l.i \;,$$

where $Conn_L$ is the set of `ConstantPowerLoad` contracts connected to the generator. CSL4P provides the ability to define second order terms to map sets of objects of a certain type C to values. A set of objects is defined by a characteristic formula $r(c)$ over the free variable c. Intuitively, an object c' belongs to the set if $r(c')$ evaluates to true. Let $o \in F$ be a binary operator over elements of a type τ (i.e. $\sigma(o) = (\tau, \tau, \tau)$) such that (D_τ, o) is a commutative monoid with neutral element e_o. Then, the following term:

$$\mathop{\mathcal{O}}_{c:C|r(c)} c.v \;,$$

is an iterator that applies the binary operator o to the terms $c.v$ of all those objects of type C in the set defined by the characteristic formula $r(c)$. The term is well-defined if type C has a variable v and if operator o is defined over $\tau(v)$ – two conditions that can be statically checked in CSL4P. To define the precise semantics of this term, we use a standard construction as follows. For a Σ-interpretation μ, consider the power set $\mathcal{U} = 2^{D_\tau}$. In this expanded domain, we can interpret an operator $\mathcal{O}'(X.v)$ as follows: $\mathcal{O}'(\emptyset.v)$ is equal to e_o, and $\mathcal{O}'((\{x\} \cup X).v) = o(x.v, \mathcal{O}'(X.v))$. Let X' be the largest set in $\mathcal{U}$ such that $\forall c' \in X$, $r(c')$ is true, then $\mathcal{O}_{c:C|r(c)} c.v = \mathcal{O}'(X'.v)$.

The following example shows the concrete syntax used in this paper:

```
assertion rule totalCurrent {
  forall g: Generator,
    g.i = Sum{c:Contract | Connected(g,c)}[i]
}
```

In this composition rule, `Sum` is the operator $\mathcal{O}$ and $r(c) \equiv$ `Connected(g,c)`. The corresponding binary operator o is the sum over reals. The variable selector `[i]` plays the role of the variable v. The term $g.i$, that indicates the current of the generator, is constrained to be equal to the sum of the currents of all contract instances connected to it.

The second order term just introduced is similar to the Object Constraint Language (OCL) `iterate` operation over collections[8].

2.4 CSL4P semantics

Preliminaries A formula ϕ defines a set of interpretations $[\![\phi]\!] = \{\mu : \mu \models \phi\}$, also referred to as the models of ϕ. Thus, we use formulas to represent concrete components (i.e., sets of behaviors). Let M_1 and M_2 be two concrete components and ϕ^{M_1} and ϕ^{M_2} be the two formulas that describe their behaviors. Then, the behaviors of the composition $M_1 \times M_2$ is the formula $\phi^{M_1} \wedge \phi^{M_2}$. This composition operator is associative and commutative. A static A/G-contract is a pair $\mathcal{C}(\phi^A, \phi^G)$ of formulas. We will use the calligraphic font for A/G-contracts. Models of ϕ^A are environments that an implementation of this contract must accept. For such environments, an implementation must satisfy ϕ^G. More formally, a component M implements the

[8]`http://www.omg.org/spec/OCL/2.4/` Section 7.6.6

static A/G-contract $\mathcal{C}$ if and only if $\phi^A \wedge \phi^M \Rightarrow \phi^G$ holds. As explained in [10], an implementation of $\mathcal{C}$ is also an implementation of $\mathcal{C}'(\phi^A, \phi^G \vee \neg\phi^A)$ which is said to be saturated, or in canonical form.

A saturated static A/G-contract $\mathcal{C}(\phi^A, \phi^G)$ is compatible if and only if ϕ^A is satisfiable (i.e. $[\![\phi^A]\!] \neq \emptyset$, meaning that there exists an environment in which an implementation of the contract can be used), and is consistent if and only if ϕ^G is satisfiable (i.e. $[\![\phi^G]\!] \neq \emptyset$, meaning that the set of required behaviors is non-empty). Given two saturated static A/G-contracts $\mathcal{C}_1(\phi_1^A, \phi_1^G)$ and $\mathcal{C}_2(\phi_2^A, \phi_2^G)$, $\mathcal{C}_1$ refines $\mathcal{C}_2$, written $\mathcal{C}_1 \preceq \mathcal{C}_2$, if and only if $(\phi_2^A \Rightarrow \phi_1^A) \wedge (\phi_1^G \Rightarrow \phi_2^G)$ is valid (i.e. the refinement accepts more environments and provides stricter guarantees). Finally, the composition $\mathcal{C}_1 \otimes \mathcal{C}_2$ is a saturated static A/G-contract $\mathcal{C}(\phi^A, \phi^G)$ where $\phi^G = \phi_1^G \wedge \phi_2^G$ and $\phi^A = (\phi_1^A \wedge \phi_2^A) \vee \neg\phi^G$. The composition operator is associative and commutative.

Semantics of platforms and platform instances. A platform $\mathcal{P} = (\mathcal{L}, \mathcal{R})$ defines a set of platform instances $[\![\mathcal{P}]\!]$. In this section, we provide the definition of such set and we also formulate the problem of checking whether a given platform instance belongs to such set. Given a platform instance $I_\mathcal{P} = (comp, conf, conn)$, we construct a static A/G-contract $\mathcal{C}_I$ that takes into account contract instances in *comp*, configuration constraints *conf*, connections *conn* and assertion rules. Then, we construct a static A/G-contract $\mathcal{C}_v$ for the validity rules. We use these two A/G-contracts to state the conditions under which a platform instance belongs to a platform. Figure 1 shows the roles played by the elements of a platform instance in a design process. The set *comp* corresponds to a contract $\mathcal{C}$ obtained by composition of all its elements. Assertion rules are guaranteed by "nature" without any further assumption (contract $\mathcal{C}_n$). The platform instance is configured by a designer who sets connections and defines parameters. By doing so, the designer effectively guarantees that all constraints in *conf* hold without further assumptions (contract $\mathcal{C}_f$). Finally, validity rules are requirements that must be satisfied by any platform instance belonging to $[\![\mathcal{P}]\!]$. While assertion rules are composed with the components of the platform instance as a component manifesting its own behaviors, validity rules play a different role: they define architectural requirements and therefore they must be "refined" by the platform instance.

More formally, each contract instance $c : C$ in the set *comp*, is associated with a static A/G-contract $\mathcal{C}_c(\phi_C^A[V_C \to c.V_C], \phi_C^G[V_C \to c.V_C])$ (where we have used the notation $\phi[x \to y]$ to denote variable substitution, and naturally extended the notation to sets so that $\phi[X \to c.X]$ means $\phi[x \to c.x]$ for all $x \in X$). The semantics of the set of components *comp* is a static A/G-contract $[\![comp]\!] = \mathcal{C} = \otimes_{c \in comp} \mathcal{C}_c$.

Recall the definition of $V_I = \cup_{c \in comp}\{c.v : v \in V_{\tau(c)}\}$, the set of variables of a platform instance. Then, the semantics of the set of constraints *conf* and the set of connections *conn* is a static A/G-contract defined as follows:

$$[\![conf, conn]\!] = \mathcal{C}_f = (\phi_f^A, \phi_f^G) = \left(true, \bigwedge_{\pi_i \in conf} \pi_i \wedge \bigwedge_{(u,v) \in conn} Connected(u, v) \right) .$$

Assertion rules are also guaranteed to hold by "nature" without any further assumption. Thus, the semantics of the set of assertion rules $\mathcal{R}_A$ is a static A/G-contract defined as follows:

$$[\![\mathcal{R}_A]\!] = \mathcal{C}_n = (\phi_n^A, \phi_n^G) = (true, \bigwedge_{r \in \mathcal{R}_A} r) .$$

Consider now the set of validity rules $\mathcal{R}_V$. They are expressed without restricting the set of environments to be accepted by a platform instance. The contract associated with the validity rules is the following:

$$[\![\mathcal{R}_V]\!] = \mathcal{C}_v = (true, \phi_v^G), \text{ where } \phi_v^G = \bigwedge_{r \in \mathcal{R}_V} r .$$

Given a platform instance $I_\mathcal{P} = (comp, conf, conn)$, its semantics is a static A/G-contract $[\![I_\mathcal{P}]\!] = \mathcal{C}_I = \mathcal{C} \otimes \mathcal{C}_f \otimes \mathcal{C}_n$. The semantics of a platform is a set of platform instances $[\![\mathcal{P}]\!]$ such that $I_\mathcal{P} \in [\![\mathcal{P}]\!]$ if and only if

- $I_\mathcal{P}$ is consistent and compatible: $[\![\phi_I^A]\!] \neq \emptyset \wedge [\![\phi_I^G]\!] \neq \emptyset$
- $I_\mathcal{P}$ satisfies the validity rules in some environment: there exists $\mathcal{C}_E = (\phi_E^A, true)$ such that $[\![\phi_E^A]\!] \neq \emptyset$ and $\mathcal{C}_I \preceq \mathcal{C}_v \otimes \mathcal{C}_E$

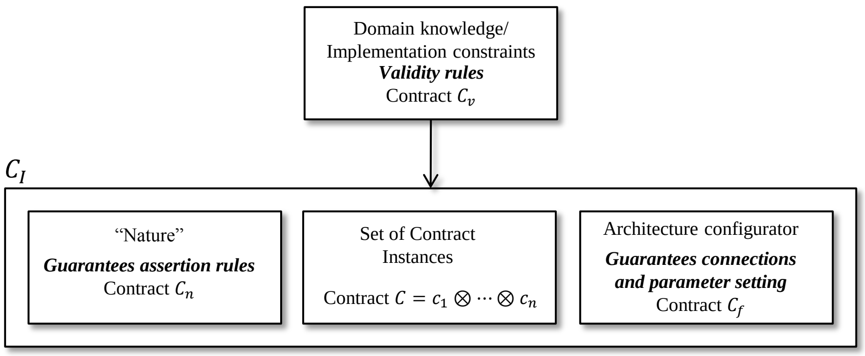
Domain knowledge/
Implementation constraints
Validity rules
Contract $\mathcal{C}_v$
$\mathcal{C}_I$
"Nature"
Guarantees assertion rules
Contract $\mathcal{C}_n$
Set of Contract Instances
Contract $\mathcal{C} = c_1 \otimes \cdots \otimes c_n$
Architecture configurator
Guarantees connections and parameter setting
Contract $\mathcal{C}_f$

Figure 1: Interpretation of the contracts that define a platform instance.

Proposition 1. *Let $\mathcal{P}$ be a platform and $I_{\mathcal{P}}$ a platform instance. If $\phi_I^G \Rightarrow \phi_v^G$ is valid and $[\![\phi_I^A \wedge \phi_I^G]\!] \neq \emptyset$, then $I_{\mathcal{P}} \in [\![\mathcal{P}]\!]$.*

Proof. The platform contract $\mathcal{C}_I$ must refine $\mathcal{C}_v \otimes \mathcal{C}_E = (\phi_E^A \vee \neg\phi_v^G, \phi_v^G)$, meaning that $\phi_I^G \Rightarrow \phi_v^G$ and $\phi_E^A \vee \neg\phi_v^G \Rightarrow \phi_I^A$ must both be valid. It remains to show that there exists a formula ϕ_E^A such that $[\![\phi_E^A]\!] \neq \emptyset$ and $\phi_E^A \vee \neg\phi_v^G \Rightarrow \phi_I^A$ is valid. Since contracts are in canonical form, $\neg\phi_I^A \Rightarrow \phi_I^G$ is valid, and $(\neg\phi_I^A \Rightarrow \phi_I^G) \wedge (\phi_I^G \Rightarrow \phi_v^G) \Rightarrow (\neg\phi_v^G \Rightarrow \phi_I^A)$. Thus, the largest assumption that satisfies $\phi_E^A \vee \neg\phi_v^G \Rightarrow \phi_I^A$ is $\phi_E^A \equiv \phi_I^A \wedge \phi_v^G$. From $[\![\phi_I^A \wedge \phi_I^G]\!] \neq \emptyset$ and $\phi_I^G \Rightarrow \phi_v^G$ we can conclude $[\![\phi_E^A]\!] \neq \emptyset$. Finally, from $[\![\phi_I^A \wedge \phi_I^G]\!] \neq \emptyset$ we conclude $[\![\phi_I^A]\!] \neq \emptyset \wedge [\![\phi_I^G]\!] \neq \emptyset$, i.e. the platform instance is consistent and compatible. □

The condition $[\![\phi_I^A \wedge \phi_I^G]\!] \neq \emptyset$ used in Proposition 1 is a sufficient condition since $[\![\phi_I^A \wedge \phi_I^G]\!] \subseteq [\![\phi_I^A \wedge \phi_v^G]\!]$. However, this condition is desirable for a platform since it guarantees the existence of implementations ϕ^M such that $[\![\phi^M \wedge \phi_I^A]\!] \neq \emptyset$ and $\phi^M \wedge \phi_I^A \Rightarrow \phi_I^G$.

Corollary 1. *Let $\mathcal{P}$ be a platform and $I_{\mathcal{P}}$ a platform instance. Then, $I_{\mathcal{P}} \subseteq [\![\mathcal{P}]\!]$ if the following conditions hold:*

1. *The following formula is unsatisfiable:*
$$\phi_I^G \wedge \neg \bigwedge_{r \in \mathcal{R}_V} r \ .$$
2. *The following formula is satisfiable:*
$$\bigwedge_{c:C \in comp} \phi_C^G[V_C \to c.V_C] \wedge \bigwedge_{\pi_i \in conf} \pi_i \wedge \bigwedge_{(u,v) \in conn} Connected(u,v) \wedge \bigwedge_{r \in \mathcal{R}_A} r \wedge \bigwedge_{c:C \in comp} \phi_C^A[V_C \to c.V_C] \ .$$

The first unsatisfiability condition is equivalent to the validity of $\neg\phi_I^G \vee \phi_v^G$, or equivalently of $\phi_I^G \Rightarrow \phi_v^G$. The second satisfiability condition is equivalent to $[\![\phi_I^A \wedge \phi_I^G]\!] \neq \emptyset$.

The satisfiability problem for the formulas in Corollary 1 is undecidable in the general case. However, the following theorem provides decidability results for some classes of models written in the CSL4P language.

Theorem 1. *Given a platform instance $I_{\mathcal{P}} = (comp, conf, conn)$ such that comp is a finite set of contract instances, checking that $I_{\mathcal{P}} \subseteq [\![\mathcal{P}]\!]$ is decidable.*

Proof. Under the assumption of a finite set of components $comp = \{c_1, \ldots, c_n\}$, it is possible to eliminate second order terms and quantifiers. Assumptions and guarantees are quantifier free formulas; thus it is sufficient to discuss the elimination of quantifiers from rules. Recall that rules can be of two types: $\forall c : C.\phi(c)$ or $\exists c : C.\phi(c)$ where C is a contract type and $\phi(c)$ is a formula of the same form or a quantifier free formula. Let $E(\phi, comp)$ denote the quantifier elimination operator such that ϕ is a formula with the same form as rules and *comp* is a finite set of contract instances:

$$E(\phi, comp) = \begin{cases} \bigwedge\limits_{c:C \in comp} E(\psi(c), comp) & \text{if } \phi \equiv \forall c : C.\psi(c) \\ \bigvee\limits_{c:C \in comp} E(\psi(c), comp) & \text{if } \phi \equiv \exists c : C.\psi(c) \\ \phi & \text{otherwise} \quad . \end{cases} \tag{1}$$

A second order term $\underset{c:C|r(c)}{\mathcal{O}} c.v$ can be expanded to:

$$E(r'(c_1), comp) \cdot c_1.v \ o \ \ldots \ o \ E(r'(c_n), comp) \cdot c_n.v$$

We have used the notation $\phi \cdot v$ in the expansion where ϕ is a formula and v is a variable. The meaning of this notation is the following: $True \cdot v = v$ and $False \cdot v = e_o$.

After eliminating quantifiers and second order expressions, the validity of $I_{\mathcal{P}} \subseteq [\![\mathcal{P}]\!]$ is equivalent to solving two satisfiability problems (Corollary 1) of quantifier-free formulas in the theory $T_{EUF} \cup T_{\mathbb{R}}$ which is decidable [36]. □

2.5 Complexity and Expressiveness

The verification of platform instances reduces to satisfiability of quantifier free FOL formulas. Because CSL4P relies on an SMT solver, it is possible to expand the signature Σ defined in Section 2.2 by composing theories. In this paper, we have used the theory of equality and uninterpreted functions T_{EUF}, and the theory of real numbers $T_{\mathbb{R}}$, but there are no technical difficulties in adding other decidable theories, such as the theory of arrays. The complexity of solving a satisfiability problem for quantifier free formulas in the theory T_{EUF} is decidable in polynomial time, and the same holds for the theory $T_{\mathbb{R}}$ (although the simplex method, which has exponential complexity, is often used). Thus, even though quantifier elimination for CSL4P can be carried out only with a finite set of contract instances, it is possible to still have variables with infinite domains (such as the real numbers).

The CSL4P language relies on FOL which is sufficiently expressive as it allows describing relations among objects. The extension in CSL4P allows defining relations among sets of objects, but this is possible only for special kinds of operators that satisfy the requirements in Section 2.3. However, CSL4P is not a second order language and therefore does not allow defining relations among relations. This feature could be very useful to further generalize the notion of composition and the way in which assumptions and guarantees are composed. In fact, the composition operator on contracts takes two (A, G) pairs, which are relations, and maps them to another pair (A, G). It is therefore a second order statement which is hard-coded in the compilation algorithm.

3 CSL4P Development Environment

CSL4P models are edited in a dedicated environment based on the Eclipse[9] framework. The environment has been built using the Xtext[10] Textual Modeling Framework (TMF) that allows defining the grammar of a domain specific language, creating custom scoping mechanisms, developing code generators, and implementing specialized routines to check problems such as type consistency. The CSL4P language requires checking more advanced properties such as that an architecture belongs to a platforms (the problem defined by Theorem **??**). Thus, the development environment includes a compiler and a Satisfiability Modulo Theories (SMT) solver. The compiler translates a model of an architecture into an SMT problem instance, and an SMT solver is used to check if the architecture is correctly specified. SMT solvers are capable of answering the following question: given a certain context Φ (i.e. a set of sentences that are asserted to hold) and given a formula ϕ (also called query), does the context entail the formula? Formally, an SMT solver checks for the validity of the following formula $\Phi \models \phi$. In this section we describe how the membership problem is compiled into an instance of an SMT problem.

Let $I_{\mathcal{P}} = (comp, conf, conn)$ be an instance of a platform $\mathcal{P} = (\mathcal{L}, \mathcal{C})$. The first step in the encoding of the membership problem is the definition of the contract types. A contract type $C_i \in \mathcal{L}$ is encoded as a

[9]`https://www.eclipse.org/`
[10]`http://www.eclipse.org/Xtext/`

record type that defines all the variables of the contract type, and three predicates for the assumption ϕ_i^A, the guarantee ϕ_i^G and the saturated guarantee $\phi^G \vee \neg\phi_i^A$. The elements of the record type are recursively defined as follows. The variables V_i are part of the record. If a contract type $C_j \in \mathcal{L}$ is such that $C_i \triangleright C_j$, then the record also includes all the variables of the record type for C_j. In addition, the record includes a special variable named `port` of type `Interface` that is used to define connections among contract instances. The reason for introducing this variable is that languages available to express SMT problems do not support sub-typing. Thus, it is not possible to define a generic type `Contract` and then let all other contract type extend it. Therefore, it would not be possible to encode the *Connected* predicate. By introducing variable `port` in each contract, such predicate can now be encoded by an equivalent predicate symbol `Connected` of arity `(Interface,Interface)`. For example, the variables of the contract type `Generator` presented in Section 2.1 are encoded by a record type `(R Real, v0 Real, i Real, v Real, port Interface)`. Each contract type is associated with three predicates: the assumption `A_C`, the guarantee `G_C`, and the guarantee `Gp_C` of the saturated contract. The following listing shows the SMT code for contract type `Generator` in the CVC language [11]:

```
Generator : TYPE = [# R : REAL , v0 : REAL  , i : REAL  ,
                     v : REAL , port : Interface #] ;
A_Generator : Generator -> BOOLEAN = LAMBDA(c : Generator) :
              c.v0*c.i<= 200000;
G_Generator : Generator -> BOOLEAN = LAMBDA(c : Generator) :
              c.v=c.v0 - c.R*c.i ;
Gp_Generator : Generator -> BOOLEAN = LAMBDA(c : Generator) :
              G_Generator(c) OR NOT A_Generator(c);
```

Once contract types have been defined, it is possible to declare constants corresponding to contract instances. For each contract instance $c : C \in comp$, a constant `c:C` is added to the context of the SMT solver.

Contract C_f is defined by two guarantee predicates `G_conf` and `G_conn` corresponding to the set of constraints and connections defined by the designer of the platform instance. `G_conf` is the conjunction of all constraints in $conf$. These constraints are ground and can be directly used in the definition of the predicate. `G_conn` is the conjunction of all connection constraints: $\mathit{Connected}(c1, c2)$ is encoded by the predicate `Connected(c1.port,c2.port)`.

Assertion and validity rules require quantifier elimination. Let C be a contract type and $comp(C) = \{c_1, \ldots, c_k\} \subseteq comp$ be the set of contract instances of type C. Let $ra_i \in \mathcal{R}_A$ be an assertion rule (either of the form $\forall c : C.\phi(c)$, or $\exists c : C.\phi(c)$. Then the following predicate is generated:

```
RA_i : BOOLEAN = f(c1) B ... B f(ck)
```

where `B=AND` for universally quantified rules and `B=OR` for existentially quantified rules, and `f(ci)` is the encoding of $\phi(c_i)$ obtained by quantifier elimination.

For second order terms $\underset{c:C|r(c)}{\mathcal{O}} c.v$, the compiler generates the following term:

$$\text{IF } r(c_1) \text{ THEN } c_1.\text{v ELSE } e_o \text{ ENDIF o } \ldots \text{ o IF } r(c_k) \text{ THEN } c_k.\text{v ELSE } e_o \text{ ENDIF}$$

The same procedure is used to compile a validity rule rv_i into the corresponding SMT predicate `RV_i`. Finally, the platform instance guarantee `G` can be generated as the following predicate:

$$\texttt{G} = \bigwedge_{c \in comp, C=\tau(c)} \texttt{Gp\_C(c)} \wedge \bigwedge_{i \in [1,a]} \texttt{RA\_i} \wedge \texttt{G\_conf} \wedge \texttt{G\_conn},$$

and the platform instance assumption `A` as the following predicate:

$$\texttt{A} = \left(\bigwedge_{c \in comp, C=\tau(c)} \texttt{A\_C(c)} \right) \vee \neg\texttt{G} \ .$$

[11] `http://cvc4.cs.nyu.edu/wiki/CVC4%27s_native_language`

Two queries are generated: the first query checks for the validity of the following formula:

$$\mathtt{G} \Rightarrow \bigwedge_{j\in[1,v]} \mathtt{RV\_j},$$

while the second query checks for the satisfiability of $\mathtt{A} \wedge \mathtt{G}$.

4 Application Example

The preliminary design of a new aircraft must take into account the interactions among several sub-systems such as propulsion, thermal management and electric power generation and distribution. A platform for preliminary design contains contract types modeling components that are used in all of these sub-systems. These components are characterized by different aspects or views such as mechanical, thermal and electric. Contract based design supports multiple view-points. Let $\mathcal{C}_1(\phi_1^A, \phi_1^G)$ and $\mathcal{C}_2(\phi_2^A, \phi_2^G)$ be two A/G-contracts. Their conjunction is $\mathcal{C}(\phi^A, \phi^G) = \mathcal{C}_1 \wedge \mathcal{C}_2$ such that $\phi^A = \phi_1^A \vee \phi_2^A$ and $\phi^G = \phi_1^G \wedge \phi_2^G$. When combined with the conjunction operator, $\mathcal{C}_1$ and $\mathcal{C}_2$ are called views. CSL4P supports the definition of multiple views. They are conjoined as first step of the compilation process.

In this section, we describe a platform for preliminary design and show the CSL4P verification flow.

Contract types The platform for preliminary design contains the following contract types: *fuel tanks*, *electric pumps*, *heat loads*, *air-fuel heat exchangers*, *engines*, *fuel splitters*, *electric power generators* and *electric loads*. This example is taken from [38] and [39].

To show the concrete syntax of the language and explain some extra features that the development environment provides, we start by presenting the model of a load in the electric power system:

```
platform aircraft_preliminary_design{
        component HeatSource {
                var heat Real;
        }
        component ElectricLoad
        {
                var v Real ; var i Real ;
        }
        component Load extends HeatSource, ElectricLoad {
                view Electrical {
                        var vnom Real ; var pnom Real ;
                        guarantee {
                                if ( v>=9/10*vnom and  v <= 11/10*vnom )
                                        (vnom*i = pnom)
                                else
                                        (i = 0) ;
                        }
                }
                view Thermal {
                        var vnom Real ; var pnom Real ;
                        var eff Real ;
                        guarantee {
                                if ( v>=9/10*vnom and v <= 11/10*vnom )
                                    (heat = pnom*eff)
                                else
                                        (heat = 0) ;
                        }
                }
        }
        ...
}
```

A contract type definition for a component starts with the keyword "component", while a contract type for a view starts with the keyword "view". The keyword is then followed by the name of the type and possible extensions as in object oriented programming. The `Load` type extends the two basic interfaces by adding the `Electrical` and the `Thermal` views. The extension mechanism is just a convenient way of defining a type hierarchy (and also specifying the ordering $\triangleright$ defined in Section 2.2). In this example, a contract of type `Load` is also of type `HeatSource` and of type `ElectricLoad`. Moreover, the extension mechanism is used to define common variables that belong to all contracts of that class. The model of the load is slightly different from the one presented in Section 2.1. In this model, the load absorbs constant power if the voltage is within a range around the nominal value, otherwise the load disconnects from the power distribution system (and does not generate any heat)[12]. This is not the only way to model such a contract. For example, an alternative model could include two views: the nominal view with assumption `v>=9/10*vnom and v <= 11/10*vnom` and guarantee `vnom*i = pnom`, and an off-nominal view with assumption `not (v>=9/10*vnom and v <= 11/10*vnom)` and guarantee `i = 0`. Also, common assumption among views (such as `v>=9/10*vnom and v <= 11/10*vnom`) could be defined in a base type which is then extended by views sharing such assumption. In this section we briefly describe the other contracts and a platform instance to show the verification flow.

Fuel tanks, heat loads, splitters, engines, heat exchangers and electric pumps are all components used in the fuel and thermal management systems. They have one "inlet" and one "outlet", except splitters which have two outlets. Inlets and outlets are ports characterized by two variables: the flow rate and the temperature. Electric pumps have also an additional power port that connects to electric power generators.

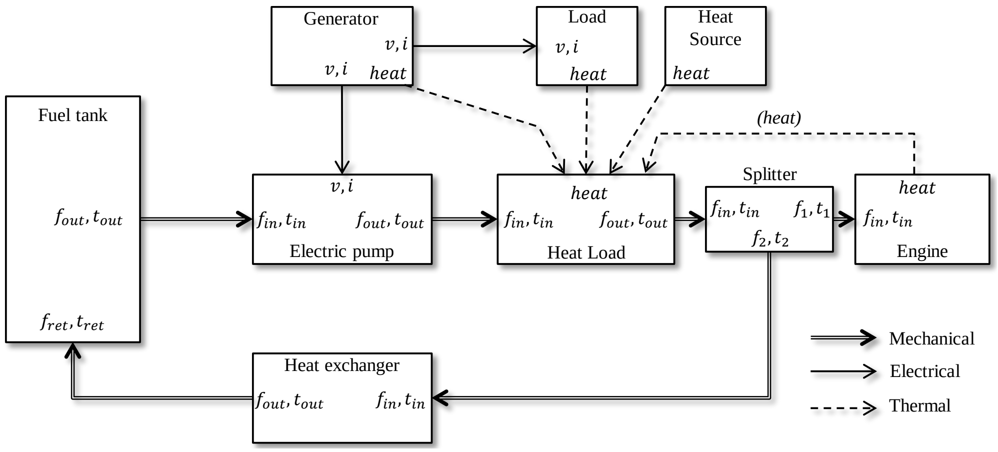

Figure 2: A platform instance derived from the platform described in this section. This instance refers to an aircraft configuration where fuel is used as a coolant and pumps are electric.

Figure 2 shows a platform instance built using these components. A pump moves fuel in a circuit around the aircraft while a heat load transfers heat from several sources to the fuel. The splitter (a passive component) is used to connect both the engine and the return path to the main fuel circuit. The engine consumes fuel from the main circuit to generate thrust. The fuel not used by the engine is sent to a heat exchanger which decreases its temperature before sending it back to the fuel tank. Connections (shown as arrows in Figure 2) are of three different types: mechanical connections transport mass, electrical connections transport current, and heat connections only transport heat.

During flight, and without considering in-air refueling, the flow rate at the inlet of the fuel tank should be smaller than the flow rate at the outlet (an assumption in the contract type for fuel tanks). The heat load contract guarantees that the heat at its input is completely transfered to the fuel which heats up while transiting from the input to the output of the heat load. The engine guarantee is the following: if the fuel

[12]The full model can be found at `http://www.alessandropinto.net/csl4p`

temperature is withing a certain range, then the engine consumes fuel and generates heat at nominal rates, otherwise the engine does not consume any fuel and does not generate any heat. The fuel pump guarantees that the flow rate at its output is maintained at a constant value (a parameter of the model). To meet this guarantee, the voltage level provided by the electric power generator must be within a certain range. If this assumption is satisfied, then the pump also guarantees to absorb a certain power level from the electric power generator that depends on the flow rate and pressure drop value. Finally, the heat exchanger guarantees to decrease the temperature of the fuel depending on the temperature of the outside air and on the effectiveness parameter (a design parameter of the component).

Platform Rules The following assertion rules are balance equations for heat and current:

```
assertion rule totalHeat {
  forall hl: Heatload,
    hl.h = Sum{c:Component | Connected(c,hl)}[h]
}
assertion rule totalCurrent {
  forall g: Generator,
    g.i = Sum{c:Component | Connected(g,c)}[i]
}
assertion rule heatIsPositive {
  forall hl: HeatSource,
    hl.h >= 0
}
```

We now introduce a validity rule. In these types of systems, it is desirable to keep the temperature of the fuel tank constant. This is a validity rule and not an assumption of the fuel tank because it is an architectural choice and not a limitation of the component itself.

```
validity rule fuelTankTemperature {
  forall ft : FuelTank,
     (ft.fin > 0) implies (ft.tin <= ft.tout+5 and ft.tin >= ft.tout-5);
}
```

Platform instance and verification The platform instance in Figure 2 corresponds to the following architecture model in CSL4P:

```
architecture arch from aircraft_preliminary_design {
  fuelTank FuelTank ;
  pump ElectricPump ;
  heatLoad HeatLoad ;
  splitter Splitter ;
  engine Engine ;
  hex HeatExchanger ;
  g Generator ;
  l Load ;
  hs HeatSource ;
  //Parameter settings
  fuelTank.tout = 15 ;
  pump.vnom = 270 ;
  ...
  //Connections
  connected(fuelTank.tout,pump.tin) ;
  connected(fuelTank.fout, pump.fin) ;
  connected(pump.tout, heatLoad.tin) ;
  connected(pump.fout, heatLoad.fin) ;
  connected(heatLoad.tout, splitter.tin) ;
  connected(heatLoad.fout, splitter.fin) ;
  connected(splitter.f1, engine.fin) ;
  connected(splitter.t1, engine.tin) ;
  connected(splitter.f2, hex.fin) ;
  connected(splitter.t2, hex.tin) ;
  connected(hex.tout,fuelTank.tret);
  connected(hex.fout,fuelTank.fret) ;
  connected(g.v,pump.v);
  connected(g.v, l.v) ;
  connected(l,heatLoad);
  connected(g,heatLoad);
  connected(hs,heatLoad);
  connected(engine,heatLoad);
}
```

The set of constraints imposed by the platform configurator assign values to the parameters of each component. The system is coupled in many ways: The engine is expecting fuel at a certain temperature that is regulated by the fuel flow rate. A higher fuel flow rate implies a lower temperature at the engine inlet. However, the part of the fuel that is not used by the engine returns to the tank through a heat exchanger which needs to be sized so that fuel returning to the tank is appropriately cooled (not too hot and not too cold). Moreover, adjusting the flow rate had an impact on the power and the heat generated by the electric power generator. The effectiveness of the heat exchanger and the flow rate are two key parameters. The list of relevant parameters is the following:

- Fuel tank: $tout = 15$;
- electric pump : flow rate $0 \leq f \leq 5$
- engine: $tmax = 112$, $tmin = 51$, (nominal fuel consumption) $fnom = 0.7$, (nominal heat) $hnom = 40$;
- heat exchanger: (outside air temperature) $tair = -30$, (effectiveness) $0.1 \leq eff \leq 0.6$;
- electric load : $power = 150,000$, (efficiency) $eff = 0.85$;
- heat source : $h = 60,000$

In a typical session, a system engineer interacts with the tools: the model is modified, the SMT problem is generated and the solver is queried with the three satisfiability problems in Theorem **??**. The result of these queries can be used to guide new changes to the model until the system engineer finds a good configuration.

We started with an initial configuration [13] where $f = 0.7$ for the pump and $eff = 0.3$ for the heat exchanger. The guarantee and the assumption of the platform instance are both satisfiable (conditions 1 and 2 from Theorem **??**). However, validity rules are not satisfied. The last query submitted to the SMT solver is to check for the validity of `G` $\Rightarrow$ `RV` where `RV` is the conjunction of all validity rules. Since this query is invalid, we can ask the SMT solver to produce a counter-model, part of which is listed below:

```
...
ASSERT (fuelTank = (# fout := 7/10, fin := 7/10, tin := 800, tout := 15...
...
ASSERT (engine = (# tmax := 112, fnom := 7/10, fout := 800, hnom := 40000,
  fin := 0, tmin := 51, tin := 8090/7, h := 0, port := (engine).port #));
...
```

The temperature at the inlet of the engine is too high. As a consequence, the engine cannot run and cannot consume fuel. In fact, the fuel tank input flow rate is equal to the output flow rate but the fuel going back to the tank is at a very high temperature which violates the validity rules.

[13]We will omit the unit of measure for brevity

To solve this problem we first increase the flow rate of the pump to $f = 2$. The temperature of the fuel at the engine drops to 63, but the temperature at the inlet of the fuel tank is 35 which is still too high and violates the validity rules. Increasing the effectiveness of the heat exchanger to 0.55 results in a valid instance.

5 Conclusions

We presented CSL4P, a language for the contract-based specification of platforms. The language allows defining *platforms* as a set of contract types and composition rules. A *platform instance* can be created by instantiating, configuring and connecting instances of contract types. The semantics of a platform instance is a static A/G-contract derived from the its components, the configuration constraints, the connection statements, and the assertion rules of the platform. The semantics of a platform is a set of platform instances that are consistent and compatible, and that refine a static A/G-contract associated with the validity rules.

CSL4P is a first step towards a language that satisfies the requirements introduced in Section 1. The language is based on Order-Sorted First Order Logic which is expressive and generic, making CSL4P suitable as a modeling language across several application domains. Its signature can be extend by introducing new functions, predicates and types, and by combining theories as described in Section 2.2. The concrete syntax, as shown in Section 4, has been developed to provide a simple way of defining types, type extensions, contracts, rules and platform instances (R6). Product families are captured by platforms which represent sets of architectures that conform to certain rules (R5). The use of contracts allows combining viewpoints, checking refinement among platform instances, and abstracting implementations at their interfaces (R1, R2, R3). Moreover, CSL4P enjoys the compositional properties that have been already shown for contracts in general [10] (R4).

Finally, we have presented a development environment for CSL4P which includes an editor and a compiler. The compiler reads a CSL4P description and generates the input for an SMT solver that checks whether a platform instance belongs to a certain platform by solving three satisfiability problems. We have shown an application example where CSL4P has been used to model an architecture for the preliminary design of a prototypical aircraft.

References

[1] *Military Standard: Engineering Management.* Department of Defense, 1974.

[2] *Military Standard: System Engineering.* Department of Defense, 1993.

[3] *ANSI/GEIA EIA-632, Processes for Engineering a System.* ANSI/GEIA, 2003.

[4] *Systems and software engineering – System life cycle processes.* ISO/IEC, 2003.

[5] K. Forsberg and H. Mooz, "System engineering for faster, cheaper, better," 1998.

[6] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," tech. rep., National Institute of Standards and Technology, 2002.

[7] H. Giese, "Contract-based component system design," in *Proc. 33rd Annual Hawaii Int System Sciences Conf*, 2000.

[8] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple viewpoint contract-based specification and design," in *Formal Methods for Components and Objects* (F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, eds.), vol. 5382 of *Lecture Notes in Computer Science*, pp. 200–225, Springer Berlin Heidelberg, 2008.

[9] A. L. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems," *Eur. J. Control*, vol. 18, no. 3, pp. 217–238, 2012.

[10] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. G. Larsen, "Contracts for System Design," Rapport de recherche RR-8147, INRIA, Nov. 2012.

[11] A. Sangiovanni-Vincentelli, "The tides of eda," *IEEE Des. Test*, vol. 20, pp. 59–75, Nov. 2003.

[12] A. Sangiovanni-Vincentelli, "Quo vadis, sld?: Reasoning about the trends and challenges of system level design," *Proc. of the IEEE*, vol. 95, no. 3, pp. 467–506, 2007.

[13] A. Pinto, A. Bonivento, A. L. Sangiovanni-Vincentelli, R. Passerone, and M. Sgroi, "System level design paradigms: Platform-based design and communication synthesis," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, pp. 537–563, June 2004.

[14] R. W. Floyd, "Assigning meanings to programs," *Mathematical aspects of computer science*, vol. 19, no. 19-32, p. 1, 1967.

[15] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, Oct. 1969.

[16] E. W. Dijkstra, *A Discipline of Programming.* Prentice Hall, 1976.

[17] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee, "A theory of synchronous relational interfaces," *ACM Trans. Program. Lang. Syst.*, vol. 33, pp. 14:1–14:41, July 2011.

[18] L. de Alfaro and T. A. Henzinger, "Interface automata," in *Proc. Symp. Foundations of Software Engineering*, pp. 109–120, ACM Press, 2001.

[19] P. Nuzzo, A. Sangiovanni-Vincentelli, X. Sun, and A. Puggelli, "Methodology for the design of analog integrated interfaces using contracts," *IEEE Sensors J.*, vol. 12, pp. 3329–3345, Dec. 2012.

[20] Open SystemC Initiative, *IEEE 1666: SystemC Language Reference Manual.* `www.systemc.org`.

[21] IEEE, *Std 1076-2000: IEEE Standard VHDL Language Reference Manual.* IEEE, 2000.

[22] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii)," Tech. Rep. UCB/EECS-2008-28, EECS Department, University of California, Berkeley, Apr 2008.

[23] "Architecture Analysis & Design Language (AADL)," Tech. Rep. AS5506 Rev. B, SAE International, Sept 2012.

[24] P. Feiler, "Modeling of system families," Tech. Rep. CMU/SEI-2007-TN-047, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2007.

[25] E. K. Jackson, E. Kang, M. Dahlweid, D. Seifert, and T. Santen, "Components, platforms and possibilities: towards generic automation for MDA," in *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, (New York, NY, USA), pp. 39–48, ACM, 2010.

[26] D. Jackson, *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[27] P. Alexander, *System-Level Design with Rosetta.* Morgan Kaufmann, 2006.

[28] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica.* Wiley-IEEE Computer Society Pr, 2003.

[29] A. Jardin, D. Bouskela, T. Nguyen, N. Ruel, E. Thomas, L. Chastanet, R. Schoenig, and S. Loembe, "Modelling of system properties in a Modelica framework," in *Proc. of the 8th International Modelica Conference*, 2011.

[30] D. Jovanovi and L. de Moura, "Solving non-linear arithmetic," in *Proceedings of the 6th International Joint Conference on Automated Deduction*, 2012.

[31] P. Nuzzo, A. Puggelli, S. A. Seshia, and A. L. Sangiovanni-Vincentelli, "CalCS: SMT solving for non-linear convex constraints," in *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pp. 71–79, October 2010.

[32] M. Frnzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert, "Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 1, pp. 209–236, 2007.

[33] S. Gao, S. Kong, and E. M. Clarke, "dreal: An smt solver for nonlinear theories over the reals.," in *CADE* (M. P. Bonacina, ed.), vol. 7898 of *Lecture Notes in Computer Science*, pp. 208–214, Springer, 2013.

[34] A. Eggers, N. Ramdani, N. Nedialkov, and M. Fränzle, "Improving SAT modulo ODE for hybrid systems analysis by combining different enclosure methods," in *Proceedings of the 9th International Conference on Software Engineering and Formal Methods*, SEFM'11, (Berlin, Heidelberg), pp. 172–187, Springer-Verlag, 2011.

[35] D. C. Karnopp, D. L. Margolis, and R. C. Rosenberg, *System Dynamics: Modeling and Simulation of Mechatronic Systems*. New York, NY, USA: John Wiley & Sons, Inc., 2006.

[36] C. Tinelli and C. Zarba, "Combining decision procedures for sorted theories," in *Proceedings of the 9th European Conference on Logic in Artificial Intelligence (JELIA'04), Lisbon, Portugal* (J. Alferes and J. Leite, eds.), vol. 3229 of *Lecture Notes in Artificial Intelligence*, pp. 641–653, Springer, 2004.

[37] Z. Manna and C. Zarba, "Combining decision procedures," in *Formal Methods at the Crossroads. From Panacea to Foundational Support* (B. Aichernig and T. Maibaum, eds.), vol. 2757 of *Lecture Notes in Computer Science*, pp. 381–422, Springer Berlin Heidelberg, 2003.

[38] G. A. Mathew and A. Pinto, "Stochastic analysis and design of systems," tech. rep., DARPA V2D2 Study (#FA9550-10-C-0116) Final Report, 2011.

[39] B. T. Murray, A. Pinto, R. Skelding, O. de Weck, H. Zhu, S. Nair, N. Shougarian, K. Sinha, S. Bopardikar, and L. Zeidner, "Complex systems design and analysis (CODA)," tech. rep., DARPA META II (#FA8650-10-C-7080) Final Report, 2011.

component Generator {	**component** ControlledGenerator {	**component** ConstantPowerLoad {
view Electrical {	**view** Electrical {	**view** Electrical {
param R , v_0 : real ;	**param** v_0 , e : real;	**param** P , v_{nom} : real ;
var v , i : real ;	**var** v , i : real ;	**var** v , i : real ;
assume { $v_0 \cdot i \leq 200e3$ };	**assume** { $v_0 \cdot i \leq 200e3$ };	**assume** { $v = v_{nom}$ };
guarantee { $v = v_0 - R \cdot i$ };	**guarantee**	**guarantee** { $v_{nom} \cdot i = P$ };
}}	{ $(1-e) \cdot v_0 \leq v \leq (1+e) \cdot v_0$ };	}}
	}}	

Table 1: Two models of a generator contract type (left and center), and a model of a constant power load type (right) .