Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2013, Article ID 934349, 11 pages
http://dx.doi.org/10.1155/2013/934349

Research Article

OntCheck: An Ontology-Driven Static Correctness Checking Tool for Component-Based Models

Xi Lin,[1,2,3] Hehua Zhang,[1,3] and Ming Gu[1,3]

[1] *School of Software, Tsinghua University, Beijing 100084, China*
[2] *Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China*
[3] *Tsinghua National Laboratory for Information Science and Technology, Beijing 100084, China*

Correspondence should be addressed to Hehua Zhang; zhanghehua@tsinghua.edu.cn

Received 8 February 2013; Accepted 21 March 2013

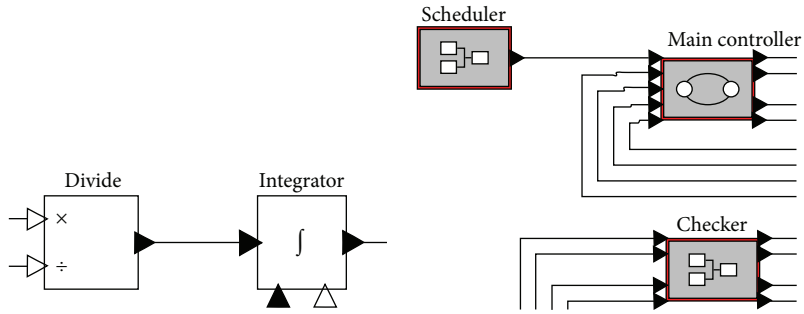
Academic Editor: Xiaoyu Song

Component-based models are widely used for embedded systems. The models consist of components with input and output ports linked to each other. However, mismatched links or assumptions among components may cause many failures, especially for large scale models. Binding semantic knowledge into models can enable domain-specific checking and help expose modeling errors in the early stage. Ontology is known as the formalization of semantic knowledge. In this paper we propose an ontology-driven tool for static correctness checking of domain-specific errors. two kinds of important static checking, semantic type and domain-restrcted rules, are fulfilled in a unified framework. We first propose a formal way to precisely describe the checking requirements by ontology and then separately check them by a lattice-based constraint solver and a description logic reasoner. Compared with other static checking methods, the ontology-based method we proposed is model-externally configurable and thus flexible and adaptable to the changes of requirements. The case study demonstrates the effectiveness of our method.

1. Introduction

Embedded systems are usually reactive systems composed of software, hardware, and networks. Interacting with physical environment increases the complexity of these systems and makes the development a difficult task [1]. Component-based models are widely used to develop embedded systems [2]. The models are composed of hierarchical components with input and output ports linked to each other. Components are atomic or composite functional units that execute concurrently. Typical developing tools for embedded systems such as Simulink [3], Ptolemy II [4], and SCADE [5] all support this design philosophy.

In a component model of embedded systems, components communicate with each other via links on ports. The links among ports must be correct. Mismatched links or assumptions among components could result in failures, especially for large scale models. As a result, the designed models need to be verified as correct, ensuring the satisfiability of requirements. Many works applied traditional formal methods on models to verify the correctness. Chen et al. [6] presented a translation mechanism from a Metropolis design to a Promela description. They took the SPIN model checker to verify the design of embedded systems at multiple levels of abstraction. Rockwell Collins built a set of tools [7] translating Simulink models into the input formats of several formal analysis tools, which enabled the analysis of Simulink and SCADE Suite models with many model checkers and theorem provers, including NuSMV, ACL2, and PVS. However, the mismatched links among components can usually be found by static checking methods. Besides, the rules to be verified are closely related to a specific application domain and may probably change as time goes on. In this paper, we developed a tool called OntCheck which is driven by ontology to improve model engineering techniques. Ontology is a formal, explicit specification of a shared conceptualization [8]. It can capture the semantic concepts of specific domains, keep consistency of concepts, and elicit specifications. Most of the applications of ontology stay in using it to generate specifications in requirements



(a) Two components linked through ports (b) Complex links among components

Figure 1: Two snippet models of linked components.

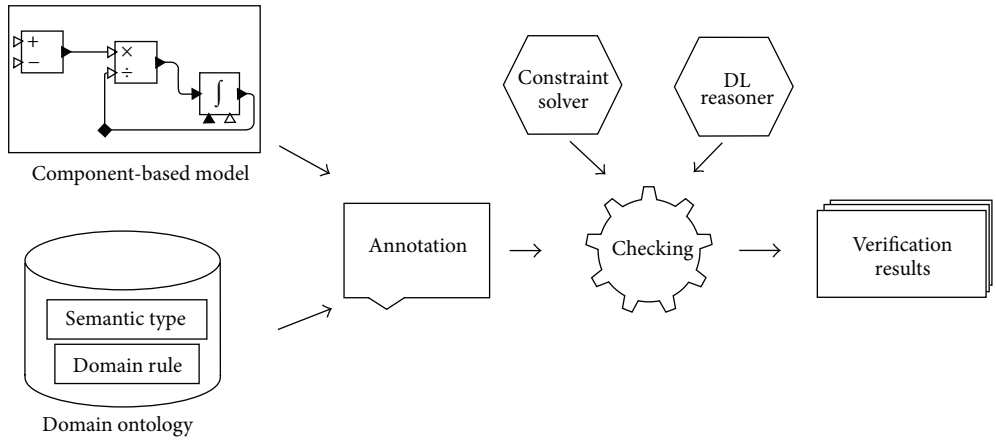


Figure 2: The overall framework of OntCheck.

engineering [9–11]. We take a further step in using ontology to drive the static correctness checking of component-based models. Compared with implementing the static checking straightforward in code, the advantage that the ontology-based method can bring is its model external configurability and thus its flexibility and adaptability to the changes of requirements.

OntCheck is a static checking tool for component-based models aiming at the mismatched links among components. Just like a front end of a compiler which checks whether a program is correctly written in terms of the syntax and semantics of programming language, we would like to check whether a model is correctly designed in terms of *semantic type compatibility* and *domain rules*. To illustrate the key idea, consider two snippet models shown in Figure 1.

The left part shows two components linked through ports. We need to ensure that such composition complies with the designer's intent. In a physical dimension domain which contains the concepts *acceleration*, *speed*, *time*, and *position*, a *Divide* component may produce a signal with the meaning *acceleration* or *speed* based on different input data. An *Integrator* component may produce the *speed* signal when receiving an *acceleration* signal or produce the *position* signal when receiving a *speed* signal. A mismatched error happens when *Integrator* requires *speed*, but *Divide* offers *acceleration*. This is out of control of typical type systems, where both ports are declared as the same basic type *double*. With a domain-specific semantic type system, we can check this kind of linking compatibility.

The right part shows the links among three composite components. According to the architecture constraints of the domain, *Scheduler* can only link to *MainController*. The connection between *Scheduler* and *Checker* is forbidden. That is to say, only the *MainController* component can react to commands from the *Scheduler* component, and *Checker* is only used to ensure that the environment constraints are satisfied for specific commands. This is a domain-restricted rule imposed by domain experts. These kinds of rules are often neglected during model development, since they are often common-sense by domain experts but stay unknown by model designers. A model is prone to hazard if it violates the requested domain rules.

These two types of requirements cover a majority part of design errors in models. In this paper, we propose OntCheck as a novel tool that takes ontology to drive the static correctness checking for component-based models. We focus on the checking of these two semantic errors. Figure 2 illustrates the overall framework of OntCheck. The information of semantic types and domain rules are modeled into a formal ontology, and annotations are then used to connect model elements

with their semantic concepts. After that, we check semantic type compatibility with a lattice-based constraint solver and use a description logic reasoner to check domain rules. In this way we can check the inconsistency between the model and the original design intention.

The rest of our paper is organized as follows. We first describe the method to build an ontology containing both semantic types and domain rules in Section 2. We then introduce the methods of semantic type compatibility and domain rule checking in Sections 3 and 4, respectively. Section 5 presents the design of the OntCheck tool. A case study is also presented to demonstrate the effectiveness of the approach. In Section 6, we summarize our work and discuss future works.

2. Description of Semantic Types and Domain Rules with Ontology

As a formal representation of domain knowledge, ontology is an abstract description of concepts and their relationships in the real world. Some ontologies have been brought out for requirements engineering, such as the ontology system in [9], the enterprise information ontology in [10], and the metamodel in [11]. However, they all lack a formal definition to support ontology-based checking.

OWL is a description logic-based ontology language. Its core elements are *class*, *individual*, *object property,* and *data property* [12]. It contains three species: OWL-DL, OWL Lite, and OWL Full. Among them, OWL-DL not only supports maximum expressiveness without losing computational completeness and decidability of reasoning systems but also has a rich tool support on construction and consistency checking. Therefore we take OWL-DL to formally describe our domain ontology.

Before presenting the formal definition of our ontology, we need a methodology to guide the construction of ontology for component-based models. In reality, every concept is related to a set of attributes. In models, systems are built hierarchically through atomic or composite components. Components, especially composite ones, often represent concepts of the system, while signals exchanged among ports represent specific attributes. This corresponds to the view of Formal Concept Analysis (FCA) [13]. FCA is a mathematical modeling method based on lattice theory to model real world in a variety of objects and attributes. It starts with a *formal context* defined as a triple $K = (G, M, I)$, where G is a set of objects, M is a set of attributes, and I is a binary relation between G and M.

We take FCA's methodology to build our ontology. The domain ontology contains the information of semantic types and the domain rules. For semantic types, *data property* of OWL is used to represent attributes, which denote signals on ports. As to domain rules, they consist of domain concepts and relations among them. Concepts can be the vocabulary for rules. Relations are the detailed constraints. *Class* of OWL is used to represent concepts, and *object property* is used to represent relations. Moreover, a domain ontology should be knowledge independent of problem solutions, so as to be shared and reused in the same domain. As a result, it needs not to include conceptual instances, which leaves out *individual* of OWL at the construction phase.

Now, we can give a formal definition of our domain ontology.

Definition 1. A domain ontology is a tuple $O := (T, h_t, C, h_c, R)$, where T is a set of semantic type and h_t is a partial order relation on T with $h_t \subseteq T \times T$. $(t_p, t_q) \in h_t$ means t_p is the superclass of t_q. C is a set whose elements are concepts. Similar to h_t, h_c is a partial order relation defined on C. R is a domain rule set, whose domain and range are all elements of C. T and h_t form *data property* of OWL, while C and h_c form *class* in OWL. R forms *object property* of OWL.

This definition will guide us to construct proper domain ontology. In the next sections, we will use *SHOIN* [14], the description logic behinds OWL-DL, to describe semantic information for the sake of preciseness and conciseness.

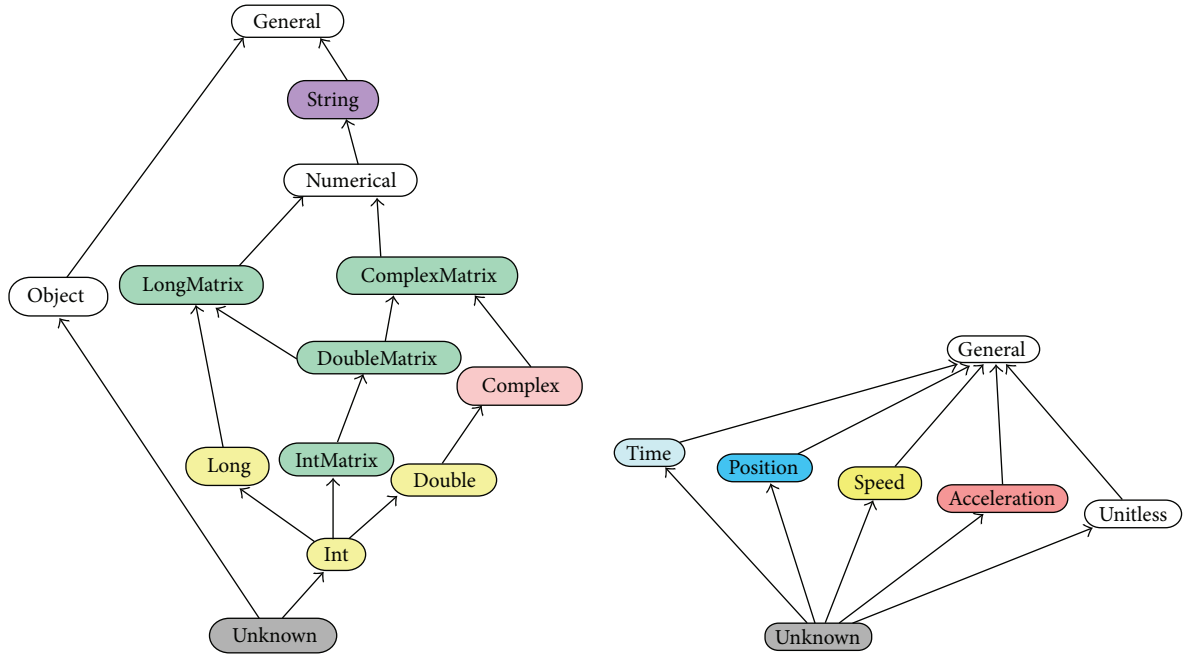
3. The Semantic Type Checking Method

Semantic type checking in OntCheck aims at ensuring the consistency on the connections among ports as shown on the left of Figure 1. Each port has a semantic type, and types at both sides of a link should be compatible. This technique can expose modeling errors early in the design phase. It offers similar benefit provided by a typical type system but is more powerful compared with it. At the same time, the semantic types in a model are typically rather domain-specific. The reason is that a different domain has different data in exchange. Therefore, we need to construct domain-specific semantic type system within a domain-specific ontology.

Definition 2. Semantic types are T in domain ontology O, which is a set of terms describing data exchanged in specific domain. They have a "is-a" partial order relation defined on themselves as h_t in O. A conversion from subclass to superclass is compatible while it is not for the other direction.

In a typical type system, type checking needs to manually declare the types of all the variables first and then check whether two types in one link are compatible or not. Instead of this "declare-check" style, we would like to use a lattice-based constraint solver, which is modified from the property system proposed in [15], to automatically decide the compatibilities of all ports' semantic types with few manual type declarations. Our method consists of a concept lattice extracted from the domain ontology, a collection of constraints associated with ports, and an efficient constraint solving algorithm. It reduces manually efforts and enhances correctness.

3.1. Concept Lattice. The concept lattice is the repository for semantic types. It is a complete lattice, that is, a set P and



(a) Lattice for traditional type system

(b) Lattice for dimension domain semantic type system

Figure 3: Two typical concept lattices.

a binary partial order relation. A complete lattice requires every subset of P has a unique least upper bound and a greatest lower bound. A typical type system can also be expressed in concept lattice [16], as shown on left of Figure 3, where each node represents a data type, and the arrows among them represent an ordering relation. Type α is greater than type β if there is a path upwards from β to α, meaning types α and β are comparable. If type α is greater than type β, and type β is greater than type α, that means type α is equal to type β. We generate a semantic type system like the dimension domain concept lattice proposed in [15] as shown on right of Figure 3.

Since we have expressed semantic type information in domain ontology's *data property*, what we need to do is mapping it to a concept lattice.

A semantic type, *speed*, for example, is written in *SHOIN* as follows:

$$speed \sqsubseteq topDataProperty, \tag{1}$$

where *topDataProperty* is a predefined OWL element, that is, the superdata property of all data properties. The partial order relation in lattice is expressed through the symbol $\sqsubseteq$ (SubPropertyOf relation). As a result the nodes in concept lattice can be mapped from each element of data properties, and the partial order relation can also be extracted.

In this way we can map concept lattice from OWL data properties. Besides, we need to make sure that the *General* least upper bound and an *Unknown* greatest lower bound are added.

3.2. Proposing the Constraint Description Language. The theory foundation of the checking method is the unique least fixed point solution for a monotonic function defined on a finite complete lattice [17]. All the functions are expressed in a set of inequalities. They represent semantic type constraints for components. Terms in inequalities are ports' semantic types. Manually specified ports will become constants while the others will be variables. Concept lattice is the source of variable assignments and the limit bound of reasoning. A unique least fixed point solution is a satisfied assignment to variables for all inequalities, which can be solved by algorithm D [17] when it is satisfiable. If there is not a satisfied assignment, it means we have error links that cause inconsistent inequalities. More details can be found in [15].

We design a Constraint Description Language (CDL) for model designers to write constraints of components. Constraints are converted into inequalities for later checking procedure. The abstract syntax of CDL is shown in Figure 4. We directly use port name to denote type of port in order to be compact. The expression includes *EqualExp* for equalities and *GreaterExp* to define inequalities. *EqualExp* will be translated into two inequalities. *GreaterExp* has a function term to handle multiconditions in inequality. Considering the features of multiple conditions, we take ternary conditional operator "?:," borrowed from programming languages, to express it. We also need a *Conflict* term to ensure the completeness and accuracy.

To illustrate the constraints, we use the components shown in Figure 5 with the dimension concept lattice as an example.

Constraints = *Exp*∗
Exp = *Port* (*EqualExp* | *GreaterExp*) **SEMICOLON**
EqualExp = **EQUAL** *Port*
GreaterExp = **GREATER** (*Port* | *FunctionTerm*)
FunctionTerm = **LPARENT** (*FunctionExp* **COLON**)∗
Otherwise **RPAREN**
FunctionExp = *Condition* **HOOK** *SemanticType*
Condition = *ConditionAtom* ((*AND* | *OR*) *ConditionAtom*)?
ConditionAtom = *Port* **EQUAL** *SemanticType* |
LPARENT *Condition* **RPAREN**
SemanticType = **STRING** *Port* = **STRING** *Otherwise* = **CONFLICT**
AND = **&&** *OR* = ||*LPARENT* = **(** *RPARENT* = **)**
EQUAL = **==** *GREATER* = **>** = *HOOK* =**?**
SEMICOLON =**;** *COLON* =**:**

Figure 4: The Abstract Syntax of CDL.

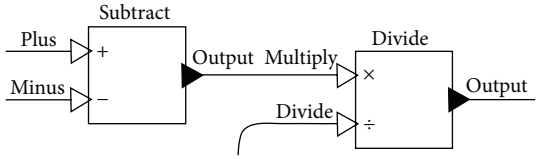


Figure 5: The Subtract and Divide components.

For the *Subtract* component on the left of Figure 5, all values have the same semantic type. Therefore, the constraints are four inequalities:

$$\begin{aligned} plus &>= minus,\\ minus &>= plus,\\ plus &>= output,\\ output &>= plus. \end{aligned} \tag{2}$$

Using *EqualExp* ("=="), we can write these constraints as

$$\begin{aligned} plus &== minus,\\ plus &== output. \end{aligned} \tag{3}$$

As to the right component *Divide*, the type of *output* is decided by different combinations of *multiply* and *divide*, based on the physical rules that

$$\begin{aligned} speed \div time &= acceleration,\\ position \div time &= speed,\\ position \div speed &= time,\\ speed \div acceleration &= time. \end{aligned} \tag{4}$$

As a result, we have inequality $output \geq f(multiply, divide)$, which reads as *the semantic type of port output will be greater than or equal the value of* $f(multiply, divide)$. The function f is defined as follows:

$$f(multiply, divide) = \begin{cases} Unknown, & if\ multiply = Unknown \\ & or\ divide = Unknown, \\ acceleration, & else\ if\ multiply = speed \\ & and\ divide = time, \\ speed, & else\ if\ multiply = position \\ & and\ divide = time, \\ time, & else\ if\ multiply = position \\ & and\ divide = speed, \\ time, & else\ if\ multiply = speed \\ & and\ divide = acceleration, \\ Error, & otherwise. \end{cases} \tag{5}$$

This is a typical *if-else if-else* structure that can be expressed using programming languages' ternary conditional operator "**?:**" which usually has the form

$$x = (y == 0\,?\,0 : (z == 1\,?\,1 : -1)). \tag{6}$$

Thus, a sample constraint can be written in CDL like this:

$$\begin{aligned} &output \\ &\geq (multiply == Unknown \,\|\, divide == Unknown\,?\,Unknown : \\ &\quad multiply == speed \&\& divide == time\,?\,acceleration : \\ &\quad multiply == position \&\& divide == time\,?\,speed : \\ &\quad multiply == position \&\& divide == speed\,?\,time : \\ &\quad multiply == speed \&\& divide == acceleration\,?\,time : \\ &\quad Conflict). \end{aligned} \tag{7}$$

As its similarity to general programming language, designers can easily describe constraints using CDL, based on domain knowledge.

3.3. Checking with the Constraint Solver. The checking is performed through a constraint solver which works on the defined constraints to ensure the connection compatibility of components. We firstly use the partial model shown in Figure 5 to see how constraints can be used to infer unspecified port types. In the beginning, all unspecified ports are in type *Unknown*, which makes all inequalities satisfiable:

$$Unknown >= Unknown. \tag{8}$$

If we manually specify the *minus* port of the *Subtract* component as type *speed*, then the inequality of *plus* >= *minus* will be *Unknown* >= *speed*, which is unsatisfiable. Then the type of *plus* should become

$$Unknow \sqcup speed = speed. \tag{9}$$

We handle all the other unsatisfiable inequalities in the same way. The output of the *Subtract* component will be *speed*. If we manually specify the type of the *divide* port of the *Divide* component to be *time*, we can infer the output of *Divide* component to be *acceleration*.

That is the core idea of type inference. With few manual declarations, we can infer all ports' types. To efficiently resolve all the constraints expressed as inequalities, we use algorithm D^+ in Algorithm 1 modified from the algorithm D in [17]. It is a linear time satisfiability determination algorithm. If all constraints are satisfiable, it can give out a satisfied assignment to variables for all inequalities. Otherwise we find errors in the model. The algorithm requires an error state set E as input. This is the *Conflict* set that denotes reaching incompatible states. In line 1, it builds the inequality set C from two sources: the constraints on each components and the constraint $sendType \leq receiveType$ on every link to guarantee lossless information transfer (pointed out in [16]). The next step is to divide inequalities into the const set and the variable set based on the greater term in inequality (manual declaration will be const here). In lines 4–6, the algorithm defines *Clist*, which is a hash list whose key is a variable and its corresponding value is inequalities. It initiates all the unannotated ports variables as *Unknown* and generates *usList*, the set of current unsatisfied inequalities. It then starts checking iteratively until all inequalities are satisfied. Inside the iteration, the algorithm first picks the last inequality from *usList*, looks for the Least Common Ancestor (LCA [18]) of the terms in two sides of the inequality, assigns the LCA to current inequality's variable, and updates all the inequalities in *Clist* indexed by this variable. It is implemented by removing the satisfiable inequalities from *unList* and adding unsatisfiable inequalities to it. The iteration stops until all the inequalities are satisfied. At last, it checks all inequalities in C_{cnst} to ensure the inferred assignments satisfying all known type declarations.

The major difference between the proposed algorithm D^+ and algorithm D is the error state. Algorithm D only judges the happening of an error when the iteration comes to the lattice's top—*General*. However, in practice, some superdata properties may not be compatible for two or more subdata properties. Therefore, our method supports user-defined error states to handle more complicated situations.

4. The Domain Rule Checking Method

The goal of domain rule checking is to express the more implicit constraints inside a domain, as the example shown on the right of Figure 1. As the fact that they usually exist deeply in mind of domain experts as important background information, they are often neglected by model designers and lead to failures. Therefore, it is important to formally define domain rules in order to help standardize not only semantic concepts but also constraints across a development team. Besides, for being able to define rules in formal, we need to construct a domain concepts vocabulary. The OWL ontology contains proper fields for both vocabulary and rules.

Definition 3. Domain rules are built on top of a domain concepts vocabulary. They are background knowledge widely existent in different domains, which can be used to maintain the correctness of model design.

We use *class* of OWL to represent concepts in domain as the vocabulary and *object property* for rules. To check whether a model is met with domain rules, we annotate model elements with domain concepts, instantiate them back into OWL ontology as individuals, and adopt a DL reasoner to check the consistency. If it becomes corrupt, the model violates some rules.

4.1. Specifying the Vocabulary. The traditional method to specify domain vocabulary in software engineering is to take UML and make a *classmodel* in the semivisual form with a *classdiagram*. Here we choose OWL for two reasons. One reason is the great overlap of OWL with UML class models. The other and more important reason is that it has a formal logic semantics which can be used for formal verification.

Class in OWL defines a group of individuals that belong together to share properties. Classes can be organized in hierarchy using the *subClassOf* relationship. In OWL, there is a built-in most general class named *Thing* as the superclass of all OWL classes. *Class* covers terms C and h_c in Definition 1.

Let us see an example. In a gate control system, there are four devices: the gate, the latch, the lift platform, and the pull-push unit. Using *SHOIN* we can get the following statements:

$$\begin{gathered} Gate \sqsubseteq Device, \\ Latch \sqsubseteq Device, \\ LiftPlatform \sqsubseteq Device, \\ PullPushUnit \sqsubseteq Device. \end{gathered} \tag{10}$$

4.2. Rule Categories. Wagner in [19] divides domain rules into four categories: integrity rules, derivation rules, reaction rules, and production rules. Integrity rules, also known as integrity constraints, are used to ensure the definition accuracy of concepts and relations. Derivation rules, also called deduction rules, consist of one or more conditions and one or more conclusions in general and can express more complex restrictions. Reaction rules consist of a mandatory triggering event term, an optional condition, and a triggered action

Require: Error States E
Ensure: Constraint satisfiability
(1) $C \leftarrow$ *ports constraints* $\sqcup$ *linking constraints*
(2) $C_{\text{cnst}} \leftarrow \{\tau \le A \in C \mid A \text{ is a const}\}$
(3) $C_{\text{var}} \leftarrow \{\tau \le A \in C \mid A \text{ is a variable}\}$
(4) init hash list $Clist[\beta]$ for distinct variable β in C
(5) init all un-annotated ports variables as *Unknown*
(6) $usList = \{\tau \le \beta \in C_{\text{var}} \mid \tau \le \beta \text{ is unsatisfied}\}$
(7) **while** $usList \neq \emptyset$ **do**
(8) $\quad \tau \le \beta = last\ constraint\ in\ usList$
(9) $\quad \beta \leftarrow$ LCA of τ and β
(10) $\quad$ **if** $\beta \in E$ **then**
(11) $\quad\quad$ **return** False
(12) $\quad$ **else**
(13) $\quad\quad$ *update* $Clist[\beta]$
(14) $\quad$ **end if**
(15) **end while**
(16) **if** *there are unsatisfied constraints in* C_{cnst} **then**
(17) $\quad$ False
(18) **else**
(19) $\quad$ True
(20) **end if**

Algorithm 1: The algorithm D^+ of constraint solver.

term or a postcondition, describing the behaviors of model. Production rules, popular as a widely used technique to implement "expert system" in the past, consist of a condition and a produced action, which can be equal to derivation rules when they implemented the form *if-Condition-then-assert-Conclusion*. Since we focus on static analysis of embedded system model, the rules should emphasize the integrity of domain concepts. As a result, we consider the integrity rules in this paper.

Consisting of constraint assertions, integrity rules can be divided into two categories: link rules and inclusion rules.

4.2.1. Link Rules. Link rules in a model describe the topological relations among components. It tells which components can link to and which are not allowed to link to for each component. Since OWL is under Open World Assumption [20] that if some things are not described in OWL, they can be true facts in reasoning, we need to explicitly define both the *link* and *notLink* object properties with a disjoint assertion.

A sample link rule is that "a *Checker* can link to some *GateController*, *LatchController*, *LiftPlatformController,* or *PullPushController* but is not allowed to link to any *Scheduler*." This can be written as follows:

$$\begin{gathered} link \sqsubseteq topObjectProperty, \\ notLink \sqsubseteq topObjectProperty, \\ link \sqcap notLink \equiv \emptyset, \\ Checker \sqsubseteq \exists link.GateController, \\ Checker \sqsubseteq \exists link.LatchController, \\ Checker \sqsubseteq \exists link.LiftPlatformController, \\ Checker \sqsubseteq \exists link.PullPushController, \\ Checker \sqsubseteq \exists notLink.Scheduler. \end{gathered} \tag{11}$$

topObjectProperty is another predefined element in OWL that is the superobject property of all object properties. Besides the descriptions like *someValuesFrom* ($\exists$), OWL also supports cardinality restrictions. There are *minCardinality*, *maxCardinality*, and *cardinality* restrictions.

4.2.2. Inclusion Rules. Inclusion rules in a model describe the inclusion relations among components. It tells what a component has inside. Same as link rules, we have disjoint *has* and *notHas* object properties in pair to guide the reasoning of DL reasoner. We can also use cardinality restrictions to refine constraints.

A sample inclusion rule that "*Device* should have a *UpLimit* to limit its movement" is written in *SHOIN* as follows:

$$\begin{gathered} has \sqsubseteq topObjectProperty, \\ notHas \sqsubseteq topObjectProperty, \\ has \sqcap notHas \equiv \emptyset, \\ Device \sqsubseteq \exists has.UpLimit. \end{gathered} \tag{12}$$

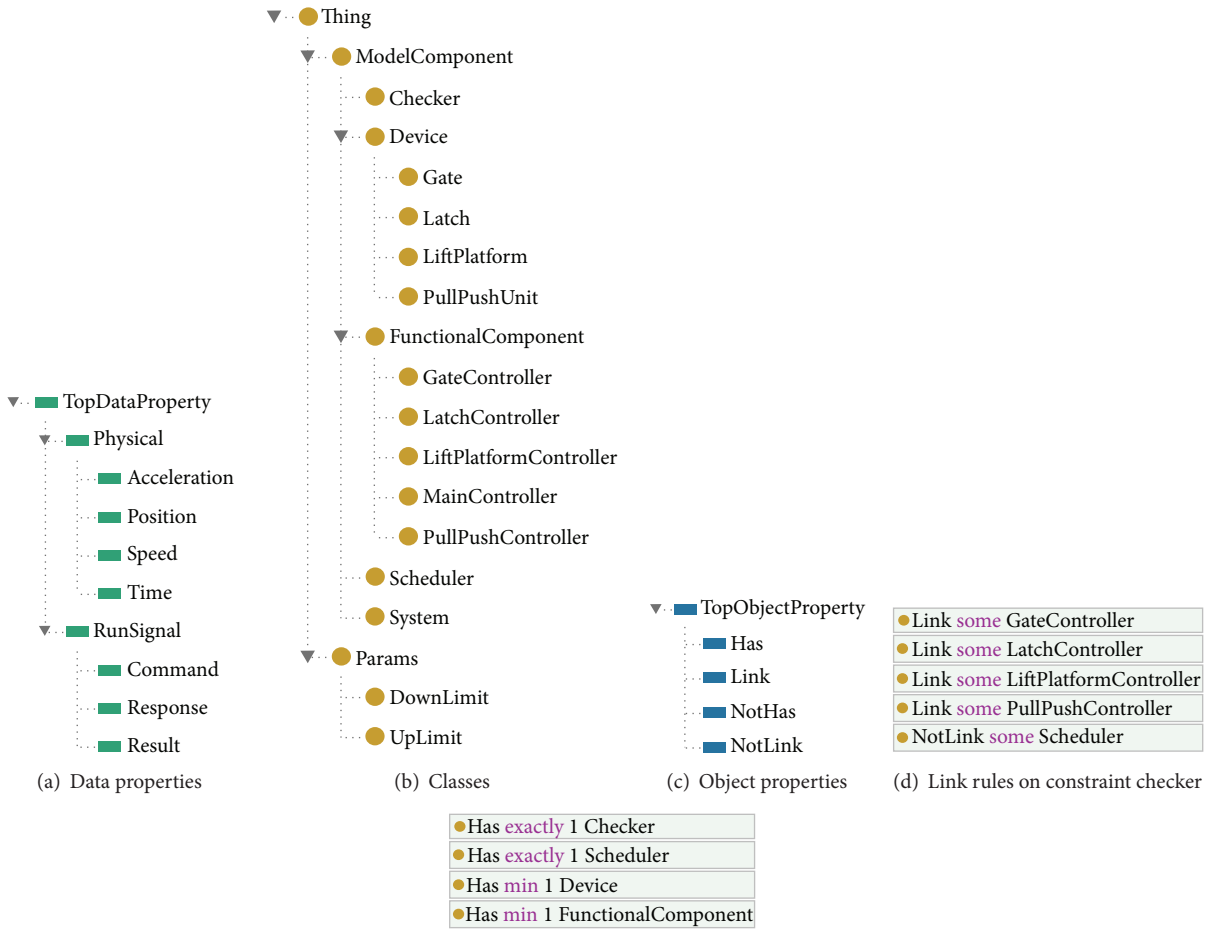


Figure 6: The ontology of gate control system.

4.3. Checking with the DL Reasoner. With *class* and *object properties* together, we build domain rules in OWL ontology. To check the consistency of model, we offer a method to annotate components in model with ontology concepts. We then use this information to instantiate domain ontology into a case specific one. At last, the instantiated ontology can be checked with mature DL reasoners.

The annotation process is a trivial part. For the instantiation, we need to

(1) generate OWL *individual* from components according to their *class* annotations,

(2) create *link* and *has* in individuals based on components' links and hierarchies,

(3) create *link*/*notLink* and *has*/*notHas* for individuals by the definition of corresponding *class*,

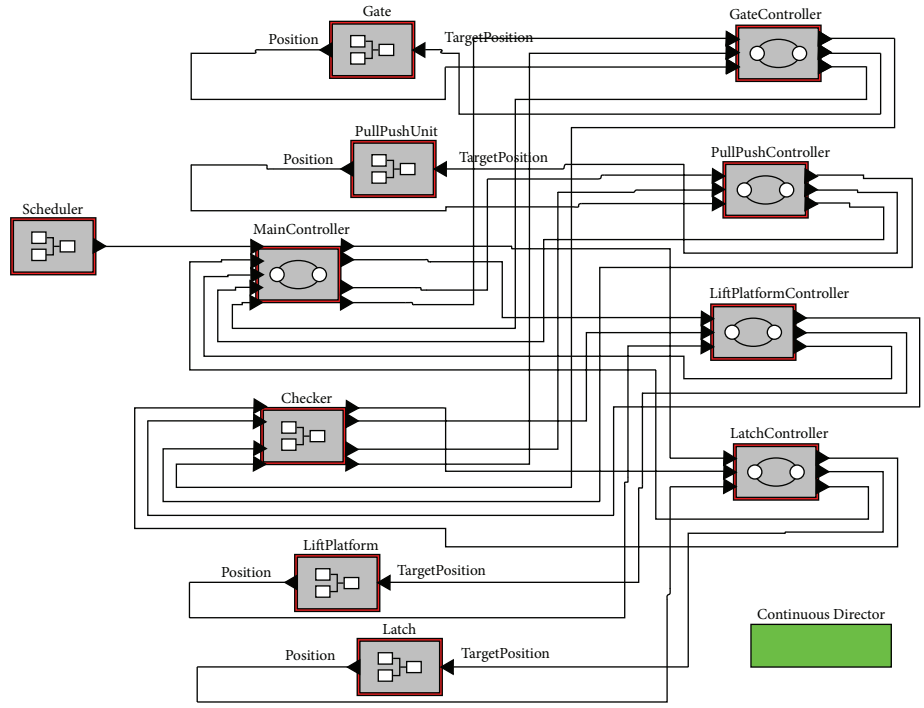
(4) set each individual as different using *owl*:*distinctMembers* in order to support cardinality restrictions.

We developed the procedure in OntCheck with Jena [21], an API for ontology model, to get an instantiated OWL ontology. For the checking part, we use Pellet [22]. It is a sound and complete OWL-DL reasoner that has extensive support for reasoning with individuals and qualified cardinality restrictions, to get the result of consistency checking.

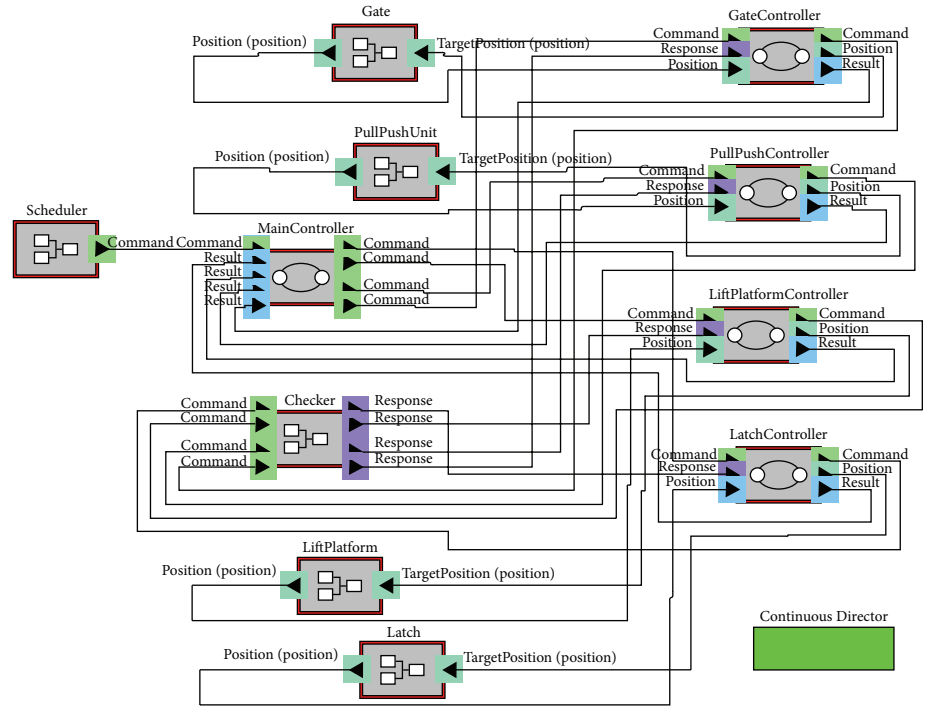
5. Tool Implementation and Case Study

OntCheck is a standalone tool for static correctness checking on models. We integrate Protégé [23], an OWL ontology modeling tool, as ontology builder and apply OntCheck to Ptolemy II [4], a component-based open source modeling design environment oriented to embedded systems. OntCheck has three main functions. First, it generates a concept lattice from OWL ontology's data properties and supports to define semantic type constraints corresponding to this lattice on components. Second, it implements the semantic type constraint solver based on algorithm shown in Algorithm 1. Third, it supports to annotate components with domain concepts, produces instantiated ontology using Jena, and invokes Pellet to check the consistency.

We demonstrate our work through a gate control system. This is a real system for a palace in Jiangsu Province, China [24]. The object of this system is to move out a gate from



(a) Original model

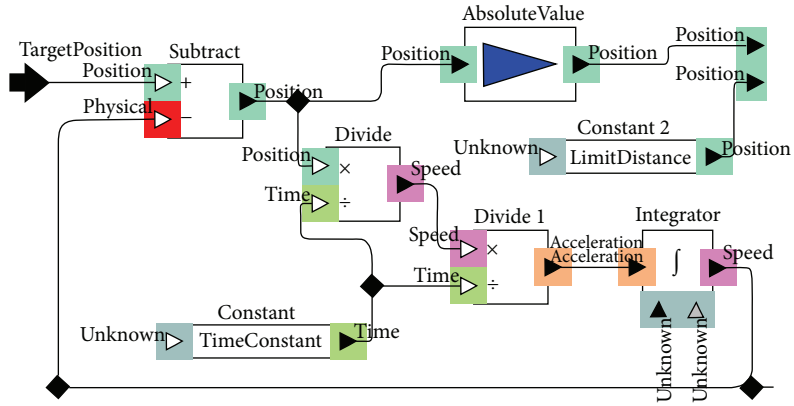


(b) Model after semantic type checking

FIGURE 7: The model of gate control system.

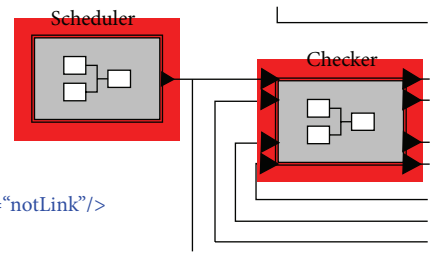
the gate repository through sequences of actions under limitations. The finished OWL ontology built by Protégé is shown in Figure 6. In this system, signals exchanged between components are *physical* for dimension data and *runSignal* for command related signals. These are data properties that will be mapped to concept lattice for semantic type checking (Figure 6(a)).

The vocabulary for domain rules is built in OWL *Class* (Figure 6(b)). The whole model corresponds to the class *System*. A *Scheduler* sends operation commands to



(a) The model with semantic type errors

```
<rdf:Description rdf:about="checker">
<j.0:link rdf:resource="Scheduler"/>
<j.0:notLink rdf:resource="Scheduler"/>
</rdf:Description>
<rdf:Description rdf:about="link">
<owl:propertyDisjointWith rdf:resource="notLink"/>
</rdf:Description>
```



(b) The model with domain rule errors

FIGURE 8: The error models of gate control system.

MainController, and the *MainController* asks *FunctionalComponents* to work one by one. A *Checker* ensures the satisfaction of movement limits, and it is requested by *FunctionalComponent* for moving permission. There are *Devices* including *Latch*, *LiftPlatform*, *PushPullUnit*, and *Gate* and their corresponding controllers. Link rules and inclusion rules adopted in the real world are described using *object property* (Figure 6(c)). The shown link rules mean a *ConstraintCheck* can link to some controllers but is not permitted to link to a *Scheduler* directly (Figure 6(d)). The shown inclusion rules represent that a *System* must have exactly 1 *Scheduler*, 1 *Checker*, and at least 1 *Device* and *FunctionalComponent* (Figure 6(e)).

Ptolemy II model of this system is shown on the left of Figure 7. After writing semantic type constraints for components by our language CDL, we can use the constraint solver to verify type correctness. The result of checking is shown on the right of Figure 7. We mark all inferred semantic types in different colors (e.g., the output port of *Scheduler* is marked as green, and the type *command* is shown in label). If there is a corruption in model, the error link is marked. As shown on the left of Figure 8, the output of *Integrator* is a *speed* signal but is connected to input of *Subtract* which desires a *position* input. Here we add *physical* into error set E since a *speed* signal cannot be mixedly used as *position*. After assigning the LCA of these two signal, we reach the incompatible state *physical*. As for domain rule checking, we annotate components with corresponding domain concepts first, instantiate ontology using model information, and invoke Pellet to check the consistency. When Pellet finds errors, we parse the error report and mark the related components like the error link shown on the right of Figure 8.

We can see that, using domain ontology, semantic type constraints and domain rules can be specified to verify the correctness of model. It leverages domain knowledge. Since the OWL ontology is standalone, it is flexible to be modified and reused, comparing to hard-code rule based checking tools. For model designers, it is easy to correct the model based on the error reports.

6. Conclusion and Future Works

Domain knowledge plays an important role in the process of software development. For component-based model development, there exist two types of special requirements, semantic type compatibility and the conformance with domain-restricted rules. In this paper, we first suggest a formal approach to precisely describe them in ontology and then use a constraint solver and a DL reasoner to verify the correctness of model. Through this approach, we formally describe semantic knowledge and ensure the design of model complying with domain-specific requirements.

As to the future work, it is worth to extend SWRL [25] for the description of rules. It is a rule language combining OWL and RuleML, which is designed to be used for rule descriptions but at the price of decidability and practical implementations. It is also meaningful to investigate the possibility of using local closed-world assumptions to make reasoning more efficiently for model elements. Another

future work is to extend semantic type into a part of vocabulary for domain rules, making a wider view to express more domain rules for correctness checking.

Acknowledgments

This research is sponsored in part by NSFC Program (No. 61202010, 91218302), National Key Technologies R&D Program (nos. SQ2012BAJY4052), and 973 Program (no. 2010CB328003) of China.

References

[1] P. Derler, E. Lee, and A. Sangiovanni-Vincentelli, "Addressing modeling challenges in cyber-physical systems," Tech. Rep. EECS-2011-17, University of California, Berkeley, Calif, USA, 2011.

[2] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 145–164, 2003.

[3] C. M. Ong, *Dynamic Simulation of Electric Machinery: Using MATLAB/SIMULINK*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[4] J. Eker, J. W. Janneck, E. A. Lee et al., "Taming heterogeneity—the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–143, 2003.

[5] F. X. Dormoy, "SCADE 6—a model based solution for safety critical software development," in *Proceedings of the Embedded Real-Time Systems Conference*, 2008.

[6] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, "Formal verification for embedded system designs," *Design Automation for Embedded Systems*, vol. 8, no. 2-3, pp. 139–153, 2003.

[7] M. Whalen, D. Cofer, S. Miller, B. H. Krogh, and W. Storm, "Integration of formal analysis into a model-based software development process," *Formal Methods for Industrial Critical Systems*, vol. 4916, pp. 68–84, 2008.

[8] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199–220, 1993.

[9] H. Kaiya and M. Saeki, "Ontology based requirements analysis: lightweight semantic processing approach," in *Proceedings of the 5th International Conference on Quality Software (QSIC '05)*, pp. 223–230, September 2005.

[10] J. Z, "Ontology-based requirements elicitation," *Chinese Journal of Computers*, vol. 23, no. 5, pp. 486–492, 2003.

[11] H. Kaiya and M. Saeki, "Using domain ontology as domain knowledge for requirements elicitation," in *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE '06)*, pp. 186–195, September 2006.

[12] D. L. Mcguinness and F. van Harmelen, "OWL web ontology language overview," http://www.w3.org/TR/owl-features/.

[13] B. Ganter and R. Wille, *Formal Concept Analysis*, Springer, Berlin, Germany, 1999.

[14] P. Hitzler, S. Rudolph, and M. Krötzsch, *Foundations of Semantic Web Technologies*, Chapman & Hall/CRC, 2009.

[15] M. Leung, T. Mandl, E. Lee et al., "Scalable semantic annotation using lattice-based ontologies," in *Model Driven Engineering Languages and Systems*, vol. 5795 of *Lecture Notes in Computer Science*, pp. 393–407, Springer, Berlin, Germany, 2009.

[16] Y. Xiong and E. Lee, "An extensible type system for component-based design," in *Tools and Algorithms For the Construction and Analysis of Systems*, S. Graf and M. Schwartzbach, Eds., vol. 1785 of *Notes in Computer Science*, pp. 20–37, Springer, Berlin, Germany, 2000.

[17] J. Rehof and T. A. Mogensen, "Tractable constraints in finite semilattices," *Science of Computer Programming*, vol. 35, no. 2-3, pp. 191–221, 1999.

[18] M. Bender and M. Farach-Colton, "The LCA problem revisited," in *LATIN 2000: Theoretical Informatics*, pp. 88–94, 2000.

[19] G. Wagner, "Rule modeling and markup," in *Reasoning Web*, N. Eisinger and J. Maluszynski, Eds., vol. 3564 of *Lecture Notes in Computer Science*, pp. 251–274, Springer, Berlin, 2005.

[20] A. Rector, N. Drummond, M. Horridge et al., "OWL pizzas: practical experience of teaching OWL-DL: common errors and common patterns," in *Engineering Knowledge in the Age of the Semantic Web*, pp. 63–81, gbr, October 2004.

[21] B. McBride, *Jena: Implementing the Rdf Model and Syntax Specification*, 2001.

[22] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: a practical OWL-DL reasoner," *Web Semantics*, vol. 5, no. 2, pp. 51–53, 2007.

[23] J. H. Gennari, M. A. Musen, R. W. Fergerson et al., "The evolution of protégé: an environment for knowledge-based systems development," *International Journal of Human-Computer Studies*, vol. 58, no. 1, pp. 89–123, 2003.

[24] R. Wang, M. Zhou, L. Yin et al., "Modeling and validation of PLC-controlled systems: a case study," in *Proceedings of the 6th International Symposium on Theoretical Aspects of Software Engineering (TASE '12)*, pp. 161–166, IEEE, 2012.

[25] I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "SWRL: a semantic web rule language combining OWL and RuleML," in *W3C Member Submission*, vol. 21, p. 79, 2004.