To appear in the proceedings of ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS), Denver, CO, USA, 4-9 October, 2009.
DRAFT VERSION dated Aug. 12, 2009.

Scalable Semantic Annotation using Lattice-based Ontologies ⋆

Man-Kit Leung[1], Thomas Mandl[2], Edward A. Lee[1],
Elizabeth Latronico[2], Charles Shelton[2], Stavros Tripakis[1], and Ben Lickly[1]

[1] UC Berkeley, Berkeley CA 94720, USA,
mankit, eal, stavros, blickly @eecs.berkeley.edu
[2] Bosch Research LLC, Pittsburgh, PA 15212 USA
thomas.mandl@at, elizabeth.latronico@us, charles.shelton@us.bosch.com

Abstract. Including semantic information in models helps to expose modeling errors early in the design process, engage a designer in a deeper understanding of the model, and standardize concepts and terminology across a development team. It is impractical, however, for model builders to manually annotate every modeling element with semantic properties. This paper demonstrates a correct, scalable and automated method to infer semantic properties using lattice-based ontologies, given relatively few manual annotations. Semantic concepts and their relationships are formalized as a lattice, and relationships within and between components are expressed as a set of constraints and acceptance criteria relative to the lattice. Our inference engine automatically infers properties wherever they are not explicitly specified. Our implementation leverages the infrastructure in the Ptolemy II type system to get efficient and scalable inference and consistency checking. We demonstrate the approach on a non-trivial Ptolemy II model of an adaptive cruise control system.

1 Introduction

Model-integrated development for embedded systems [1, 2] commonly uses actor-oriented software component models [3, 4]. In such models, software components (called actors) execute concurrently and communicate by sending messages via interconnected ports. Examples that support such designs include Simulink, from MathWorks, LabVIEW, from National Instruments, SystemC, component and activity diagrams in SysML and UML 2 [5–7], and a number of research tools such as ModHel'X [8], TDL [9], HetSC [10], ForSyDe [11], Metropolis [12], and

⋆ This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, Lockheed-Martin, National Instruments, Thales, and Toyota.

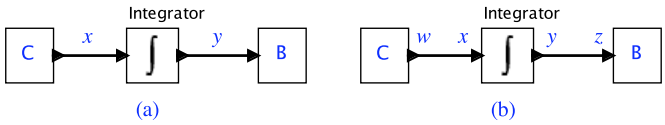

Fig. 1. Models using an Integrator, where (a) labels connections and (b) labels ports.

Ptolemy II [13]. The techniques of this paper can also be extended for equational models such as Modelica [14].

The goal of this paper is to improve model engineering techniques by providing semantic annotations. Semantic annotations help in several ways. First, if we can check consistency across such annotations, then they expose modeling errors early in the design process. This is similar to the benefits provided by a good type system. Second, they engage a designer in a deeper understanding of the model. Third, they help standardize semantic information across a development team. This can help prevent misunderstandings. Annotations can be provided manually by the designer or inferred by a tool. A model may have multiple sets of annotations, each specific to a particular use case domain.

To illustrate the key idea, consider a simple modeling component commonly used in a language such as Simulink for control system design, an Integrator. Such a component might be represented graphically as shown in Figure 1. The inputs and outputs of this component are represented as ports, depicted as small black triangles, with the input port pointing in to the component and the output port pointing out. These ports mediate communication between components. Components are composed by interconnecting their ports, and our goal is to ensure that such composition is consistent with the designer's intent.

The Integrator component has some particular properties that constrain its use. First, its input and output ports receive and send continuous-time signals, approximated in a software system by samples. Second, the samples will have data type double. Third, if the input represents the speed of a vehicle, then the output represents the position of the vehicle from some starting point; if the input represents acceleration, then the output represents speed. Fourth, the output value may vary over time even if the input does not.

A conventional type system can check for correct usage with respect to the second property, the data type of the ports. Such a type system can check for incompatible connections, and also infer types that may be implied by the constraints of the components. A behavioral type system can check for correct usage with respect to the first property, the structure of the signals communicated between components [15]. The purpose of this paper is to give a configurable and extensible mechanism for performing checks and inference with respect to properties like the third and fourth.

We refer to the third and fourth properties as **semantic types**, or more informally as **properties**. Properties in a model will typically be rather domain specific. The fact that a model operates on signals representing "speed" and

"acceleration" is a consequence of the application domain for which the model is built. Thus, unlike type systems, in our case it is essential for the model builders to be able to construct their own domain-specific **property system**. Our goal is to provide a framework for doing that without requiring that application designers understand the nuances of type theories.

An even more essential goal is that our system be sound, correct, and scalable. This will be our primary goal. Making it easy to construct and use such a property system is a secondary goal, equally important to the success of the technique, but useless without the primary goal. To accomplish the primary goal, we build on the theory of Hindley-Milner type systems [16], the efficient inference algorithm of Rehof and Mogensen [17], the implementation of this algorithm in Ptolemy II [18], and the application of similar mathematical foundations to formal concept analysis [19].

The paper is organized as follows: we introduce first the concept lattice data structure and review some of its useful properties. Section 3 then gives an overview of the mathematical foundation of our property system as a fixed point of a monotonic function. Section 4 shows how the monotonic function can be defined implicitly by a set of composable constraints associated with model components. We then give an in-depth application in Section 5, an adaptive cruise control model. Finally, we briefly describe the software architecture of our implementation in Section 6 and discuss related work in Section 7.

2 Concept Lattice

In a Hindley-Milner type system, data types are elements of a complete lattice, an example of which is illustrated in Figure 2. In that diagram, each node represents a data type, and the arrows between them represent an ordering relation. In type systems this relation can be interpreted as an "is a" relation or as a "lossless convertability" relation. For example, an Int can be converted losslessly to a Long or a Double, but a Long cannot be converted to a Double nor vice versa.

A **complete lattice** is a set P and a binary relation $\leq$ satisfying certain properties. Specifically, the relation is a **partial order relation**, meaning it is **reflexive** ($\forall\ p \in P, p \leq p$), **antisymmetric** ($\forall\ p_1, p_2 \in P, p_1 \leq p_2$ and $p_2 \leq p_1 \Rightarrow p_1 = p_2$), and **transitive** ($\forall\, p_1, p_2, p_3 \in P, p_1 \leq p_2$ and $p_2 \leq p_3 \Rightarrow p_1 \leq p_3$). A lattice also requires that any two elements $p_1, p_2 \in P$ have a unique least upper bound (called the **join** and written $p_1 \vee p_2$) and a greatest lower bound (called the **meet** and written $p_1 \wedge p_2$). To be a complete lattice we further require that every subset of P has a join and a meet in P. Every complete lattice has a top element and a bottom element. The top element is typically written as $\top$ and the bottom element $\bot$. A concept lattice is a complete lattice.

3 Property Systems

A **property system** consists of a **concept lattice**, a collection of constraints associated with modeling components, and a collection of acceptance criteria.

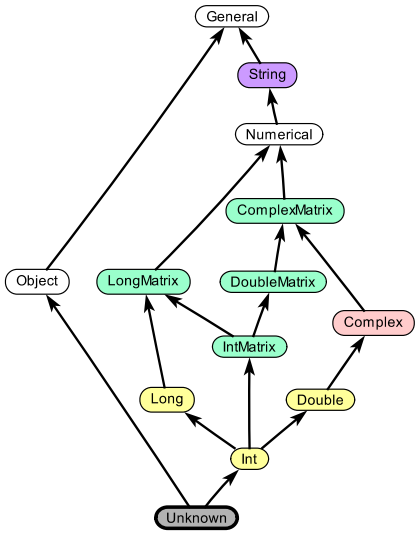

Fig. 2. A type lattice modeling a simplified version of the Ptolemy II type system.

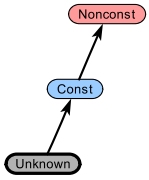

Fig. 3. A property lattice modeling signal dynamics.

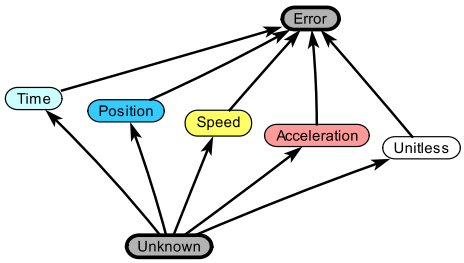

Fig. 4. A lattice ontology for dimensions Time, Position, Speed, Acceleration, and Unitless (dimensionless).

The type lattice of Figure 2 is an example of a concept lattice, as are figures 3 and 4. We will illustrate how to use the *dimension* concept lattice (Figure 4) to check for correct usage of an Integrator component as discussed above.

Consider a very simple model with three components as shown in Figure 1(a). Component C provides samples of a continuous-time signal to the integrator, which performs numerical integration and provides samples of a continuous signal to component B. Suppose that we associate the input x of the Integrator with a concept p_x in the concept lattice L. We say that the input of the Integrator "has property" p_x. We wish to catch errors, where, for example, component C sends position information to the Integrator, and component B expects speed information. This is incorrect because position is the integral of speed, not the other way around. We can construct a property system that systematically identifies such errors.

The concept lattice for this property system is shown in Figure 4. To complete the property system, we need to encode the constraints imposed by the integrator. To do this, we leverage mathematical properties of a complete lattice.

Suppose we have a model that has n model elements with properties. In Figure 1(a), we have two such elements, x and y, and their properties are $(p_x, p_y) \in L^2$, where L is the concept lattice of Figure 4. A property system for this model defines a **monotonic function** $F\colon L^2 \to L^2$ mapping pairs of properties to pairs of properties. Monotonic means that

$$(p_x, p_y) \leq (p'_x, p'_y) \Rightarrow F(p_x, p_y) \leq F(p'_x, p'_y).$$

A **fixed point** of such a function is a pair (p_1, p_2) where $(p_1, p_2) = F(p_1, p_2)$. The theory of lattices tells us that any such function has a unique least fixed point that can be found iteratively as follows

$$(p_1, p_2) = \lim_{n \to \infty} F^n(\bot, \bot). \tag{1}$$

We define the **inferred properties** of a model to be this least fixed point. The least fixed point associates with each model element a property in the lattice, which is the inferred property for that model element. If the lattice is finite, the above induction terminates in a finite number of steps.

Even for the simple Integrator example above, defining the function F is rather tedious (we explain below that it can be defined implicitly in an elegant and modular way). To reflect the constraints of the integrator, the function is

$$F(p_x, p_y) = \begin{cases} (\top, \top) & \text{if } p_x = \top \text{ or } p_y = \top \\ (p_x \vee A, p_y \vee S) & \text{else if } p_x = A \text{ or } p_y = S \\ (p_x \vee S, p_y \vee P) & \text{else if } p_x = S \text{ or } p_y = P \\ (p_x \vee U, p_y \vee T) & \text{else if } p_x = U \text{ or } p_y = T \\ (p_x \vee p_y, p_x \vee p_y) & \text{otherwise} \end{cases} \tag{2}$$

where $L = \{\bot, T, P, S, A, U, \top\}$ are the elements of the lattice in Figure 4.

The least fixed point of this function is $(p_x, p_y) = (\bot, \bot)$, found in one step by (1), which we interpret to mean that we do not have enough information to draw conclusions about the properties associated with x and y.

Suppose that component B is known to read data at its input that is interpreted as Speed. Then the function F simplifies to

$$F(p_x, p_y) = \begin{cases} (\top, \top) & \text{if } p_x = \top \text{ or } p_y = \top \\ (p_x \vee A, S) & \text{otherwise.} \end{cases}$$

In this case, the least fixed point is $(p_x, p_y) = (A, S)$. The fact that x has property Acceleration is inferred.

Suppose further that component C is known to provide data at its output that is interpreted as Position. We can encode that fact together with the previous assumptions with the function:

$$F(p_x, p_y) = (\top, \top)$$

which has least fixed point $(p_x, p_y) = (\top, \top)$, which we can interpret as a modeling error. Of course, we don't want model builders to directly give the function F. We will show below how it is inferred from constraints on the components.

We are closer to being able to formally define a property system. A **property system** for n modeling elements is a concept lattice P, a monotonic function $F\colon P^n \to P^n$, and a collection of **acceptance criteria** that define whether the least fixed point yields an acceptable set of properties. We next show how the monotonic function F can be implicitly defined in a modular way by giving constraints associated with the components.

4 Property Constraints and Acceptance Criteria

Rehof and Mogensen [17] give a modular and compositional way to implicitly define a class of monotonic functions F on a lattice and an efficient algorithm for finding the least fixed point of this function. The algorithm has been shown to be scalable to very large number of constraints, and is widely used in type systems, including that of Ptolemy II, which we leverage. Specifically, for a fixed concept lattice L, this algorithm has a computational upper bound that scales linearly with the number of inequality constraints, which is proportional to the number of model components, or the model size.

First, assume model element x (such as a port) has property $p_x \in L$, and model element y has property $p_y \in L$. For any two such properties $p_x, p_y \in L$, define an **inequality constraint** to be an inequality of the form

$$p_x \leq p_y. \tag{3}$$

Such an inequality constrains the property value of Y to be higher than or equal to the property value of X, according to the ordering in the lattice. An arbitrary collection of inequality constraints implicitly defines a monotonic function $F\colon L^n \rightarrow L^n$ that yields the least $(p_1, \cdots, p_n)$ that satisfies the inequality constraints for modeling elements 1 through n. Of course, two inequality constraints can be combined to form an **equality constraint**,

$$p_x \leq p_y \text{ and } p_y \leq p_x \Rightarrow p_x = p_y \tag{4}$$

because the order relation is antisymmetric.

In Figure 1(a), we implicitly assumed an equality constraint for the output of C and the input of the Integrator. We could equally well have assumed that each port was a distinct model element, as shown in Figure 1(b), and imposed inequality constraints $p_w \leq p_x$ and $p_y \leq p_z$. These constraints are implied by each connection between ports. Our tool permits either interpretation for the port connections, equality or inequality constraints.

Rehof and Mogensen also permit constraints that we call **monotonic function constraints**, which have the form

$$f(p_1, \cdots, p_n) \leq p_x \tag{5}$$

where $p_1, \cdots, p_n$ and p_x represent the properties of arbitrary model elements, and $f\colon P^n \rightarrow P$ is a monotonic function whose definition as a function of the property variables $p_1, \cdots, p_n$ is part of the definition of the constraint. Notice that this function does not have the same structure as the function F above. Its domain and range are not necessarily the same, so it need not have a fixed point. An example of such a monotonic function is a **constant function**, for example

$$f_s(p_1, \cdots, p_n) = S$$

where S represents Speed. Hence, to express that component B in Figure 1(b) assumes its input is Speed, we simply assert the constraint

$$f_s(p_1, \cdots, p_n) \leq p_z \ ,$$

which of course just means

$$S \leq p_z \ . \tag{6}$$

However, this does not quite assert that $p_z = S$. Indeed, that assertion would require an inequality different from (5) that is not permitted by Rehof and Mogensen's algorithm. Hence, to complete the specification, we can specify **acceptance criteria** of the form

$$p_i \leq l \tag{7}$$

where $l \in L$ is a particular constant and p_i is a variable representing the property held by the i^{th} model element. For example, we can give the acceptance criterion

$$p_z \leq S \ , \tag{8}$$

which when combined with (6), means $p_z = S$, or z is Speed. We can also declare an acceptance criterion that for each model element i with property p_i,

$$p_i < \top \ , \tag{9}$$

which means that $\top$ is not an acceptable answer for any property.

Acceptance criteria do not become part of the definition of the monotonic function F, and hence have no effect on the determination of the least fixed point. Once the least fixed point is found, the acceptance criteria are checked. If any one of them is violated, then we can conclude that there is no fixed point that satisfies all the constraints and acceptance criteria. We declare this situation to be a modeling error.

Constraints of the Integrator include one given in the form of (5) as

$$f_I(p_z) \leq p_x \quad \text{where} \quad f_I(p_z) = \begin{cases} \bot & \text{if } p_z = \bot \\ S & \text{if } p_z = P \\ A & \text{if } p_z = S \\ U & \text{if } p_z = T \\ \top & \text{otherwise} \end{cases} \tag{10}$$

This constraint is a property of the Integrator and is used together with other constraints to implicitly define the monotonic function F. The constraint (10) is more intuitive than (2) because it directly describes constraints of the Integrator component, and more modular because it only describes a constraint of the Integrator. The complete constraints for the Integrator is shown in Table 1.

To see how this works in Figure 1(b), suppose we assume constraints (6) and (10). Together, these imply that $A \leq p_x$. Our inference engine finds the least fixed point to be $p_w = p_x = A$ and $p_y = p_z = S$. This solution meets the acceptance criterion in (8). We leave it as an exercise for the reader to determine that if instead of (6) we require $A \leq p_z$, then the least fixed point is $p_w = p_x = p_y = p_z = \top$, which fails to meet acceptance criterion (9). This would be a modeling error because the output of the Integrator cannot represent Acceleration in our ontology.

In summary, a **property system** is a concept lattice, a set of constraints in the form of (3) or (5), and a set of acceptance criteria in the form of (7). The

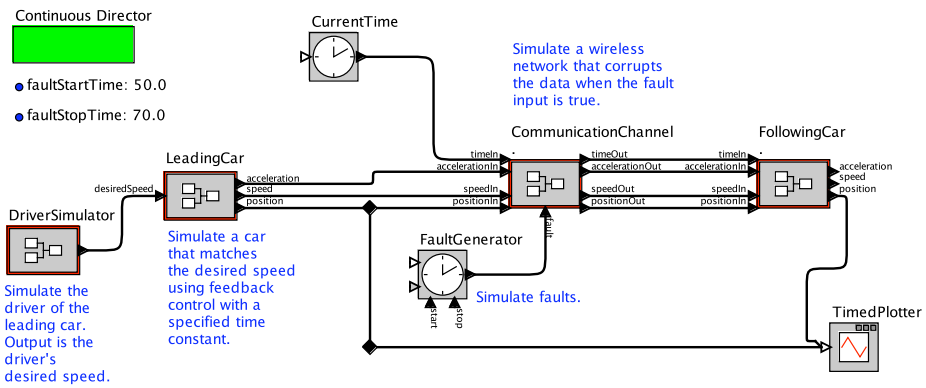

Fig. 5. Top level of an actor-oriented model of an adaptive cruise control system.

constraints come from component definitions, an interpretation for connections between components, and annotations made on the model by the model builder.

5 Adaptive Cruise Control Example

We now give a detailed example showing how this mechanism can be used in practical models. Consider an adaptive cruise control system that detects slower vehicles in front of a following vehicle and adjusts the speed of the following vehicle accordingly. Adaptive cruise control requires some form of inter-vehicle coordination, which can be implemented with a radar transmitter/receiver in the following vehicle [20]. The system must tolerate faults in coordination, such as sensor misalignment or erroneous power supply voltage for radar transceivers.

A model of such a system is shown in Figure 5 (inspired by Ptolemy II demo created by Jie Liu). In that (oversimplified) example, a leading car transmits via some channel a packet that consists of a time stamp and its current acceleration, speed, and position. A following car will use that information to adjust its speed, but only if it trusts the information it is receiving. To determine whether it trusts that information, it checks the information against a simple model of the leading car. Specifically, if a packet indicates a certain position and speed at a particular time, then when it gets a new packet, it performs a simple sanity check to see whether the new position makes sense, given the previous position and speed. If it does, then it trusts the packet.

The model composes submodels, and our task will be to show that our ontology framework can detect errors in such composition, and thus help ensure correctness of the model. Our framework can also help transform or optimize models by enabling transformations that are based on semantic annotations.

The component on the far left of Figure 5 is a model of a driver, the internals of which are not shown. The driver submodel feeds data to a car model (labeled Leading Car Model), the internals of which are shown in Figure 6. This models

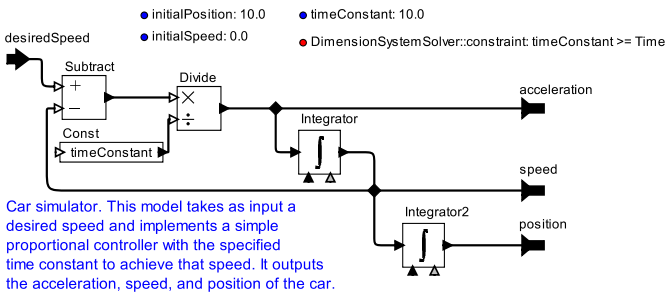

Fig. 6. A model of a car that accepts a desired speed and matches it using a feedback control loop. This model has three parameters, the initial position, the initial speed, and the time constant of the control loop.

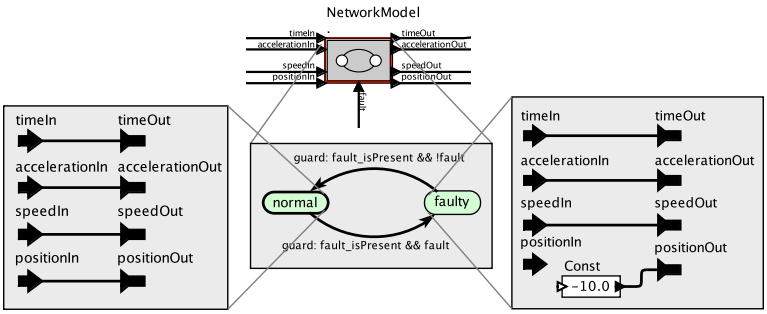

Fig. 7. A model of a wireless network that passes inputs to outputs unchanged in normal operation, but replaces an input with an arbitrary constant upon faults.

the dynamics of the leading car. Specifically, given an input desired speed, it accelerates to achieve that speed using a control loop with a specified time constant. It uses an Integrator component to convert acceleration to speed, and another Integrator to convert speed to position. The output is the acceleration, speed, and position as a function of time. These data are then sampled and transmitted over a wireless network, as shown in the middle of Figure 5.

Given a suitable ontology, our framework can infer that if the input to the Leading Car Model is a Speed, then its outputs are Acceleration, Speed, and Position, respectively. Moreover, our ontology system can be used to check that the Following Car model uses the position as a Position, not as a Speed, and vice versa. Many possible design errors can be caught by such models.

The wireless network submodel is shown in Figure 7. This is a **modal model** with two modes of operation, normal and faulty. In the normal mode, inputs are passed directly to the outputs. In the faulty mode, one of the inputs is replaced with an arbitrary constant (-10 in this simple example).

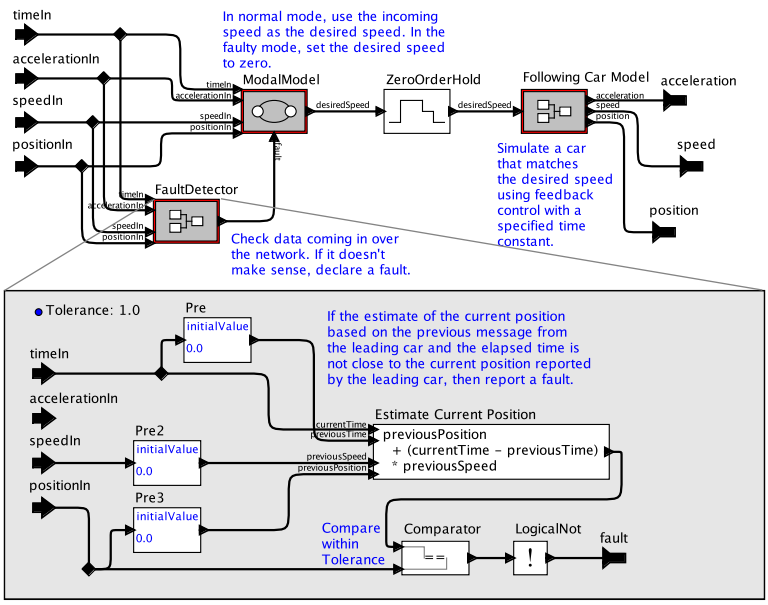

Fig. 8. A model of a following car with a simple fault detection algorithm and fault adaptation policy.

The model of the following car is shown in Figure 8, where a Fault Detector component performs the above mentioned sanity check, and uses the result to control another modal model. The details of this modal model are not shown, but like that of Figure 7, it has two modes, normal and faulty. In the normal mode, its output is equal to the input speed, and in the faulty mode its output is zero. Thus, the policy of this particular cruise control algorithm is for the following car to stop if it does not trust the data coming from the leading car, thus returning control to the driver. The output of the modal model is a desired speed, which is converted to a continuous-time signal by the ZeroOrderHold component, which then feeds it into another car model like that shown in Figure 6, which simulates the dynamics of the following car.

To perform property inference and checking for the adaptive cruise control example, we need a collection of constraints for components in the model, an illustrative subset of which are shown in Table 1. These constraints form part of a property system that can be reused in a variety of models. In addition, we added constraints and acceptance criteria that are specific to this model. Once these are specified, we can run our property inference tool on the model. A portion of the result of such a run is shown in Figure 9, where the inferred properties of ports and parameters are highlighted by the tool in a color matching that of the concept lattice elements in Figure 4. The inferred properties are also shown in text next to each port.

Component	Elements	Constraints	Where
CurrentTime	output y	$T \leq p_y$ $p_y \leq T$	
Add/Subtract	plus x, minus y, output z	$p_y \vee p_z \leq p_x$ $p_x \vee p_z \leq p_y$ $p_x \vee p_y \leq p_z$	
Integrator	input x, initialState y, output z	$f_I(p_z) \leq p_x$ $f_O(p_x) \leq p_z$ $p_y \leq p_z$ $p_z \leq p_y$	$f_I(p_z) = \begin{cases} \bot & \text{if } p_z = \bot \\ S & \text{if } p_z = P \\ A & \text{if } p_z = S \\ U & \text{if } p_z = T \\ \top & \text{otherwise} \end{cases}$ $f_O(p_x) = \begin{cases} \bot & \text{if } p_x = \bot \\ P & \text{if } p_x = S \\ S & \text{if } p_x = A \\ T & \text{if } p_x = U \\ \top & \text{otherwise} \end{cases}$
Divide	multiply x, divide y, output z	$f_D(p_x, p_y) \leq p_z$	$f_D(p_x, p_y) = \begin{cases} \bot & \text{if } p_x = \bot \text{ or } p_y = \bot \\ A & \text{if } p_x = S \text{ and } p_y = T \\ S & \text{if } p_x = P \text{ and } p_y = T \\ T & \text{if } p_x = P \text{ and } p_y = S \\ T & \text{if } p_x = S \text{ and } p_y = A \\ U & \text{if } p_x = p_y \\ p_x & \text{if } p_y = U \\ \top & \text{otherwise} \end{cases}$
Scale	input x, factor y, output z	$f_S(p_x, p_y) \leq p_z$	$f_S(p_x, p_y) = \begin{cases} \bot & \text{if } p_x = \bot \text{ or } p_y = \bot \\ S & \text{if } p_x = A \text{ and } p_y = T, \text{ or} \\ & \quad p_x = S \text{ and } p_y = U \\ P & \text{if } p_x = S \text{ and } p_y = T, \text{ or} \\ & \quad p_x = P \text{ and } p_y = U \\ p_y & \text{if } p_x = U \\ p_x & \text{if } p_y = U \\ \top & \text{otherwise} \end{cases}$

Table 1. Some of the constraints for components used in the Cruise Control example.

In Figure 9, there is exactly one constraint specified by the model builder, which is that the *timeConstraint* parameter has a property greater than or equal to Time. The input to this model resolves to Speed because we have specified similar constraints upstream in the driver model (not shown). Everything else resolves to Time, Acceleration, Speed, or Position as a consequence of the component constraints in Table 1 and the constraints implied by connections between components. Such a visual display of the inferred properties makes it easy to identify inconsistencies in the model, if there are any. Our model has none.

A property system is domain specific. We can construct multiple property systems, and even use them within the same model. Another example of a concept lattice is given in Figure 3. We interpret the property Const, when associated with a port, to mean that the value of data on that port is constant throughout the execution of the model. The property Nonconst means that the value may change during execution. We have applied this property system to analyzing the same cruise control example, and find that it successfully identifies portions of

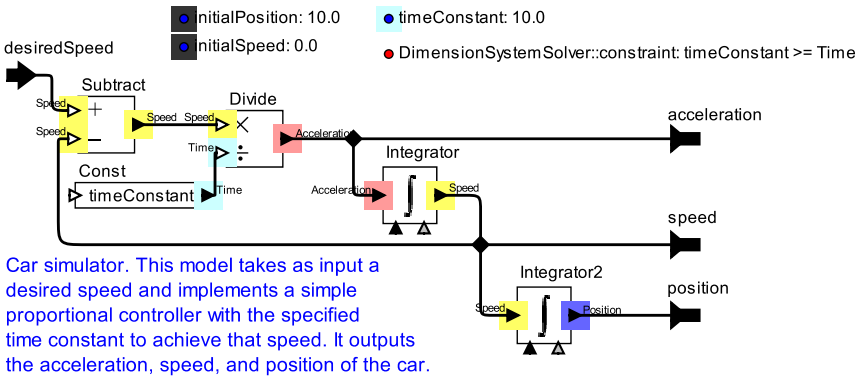

Fig. 9. Car model of figure 6 with properties resolved.

the model where messages between components have a constant value. This can be used to optimize the model automatically, or, more interestingly, to manage multiple models that represent product families. Specifically, variants of a model may result in different parts of the model being constant due to different parameterizations, which enables optimization of particular variants of the model without losing the generality of the master product family model.

6 Software Architecture

Our tool is an extension of the Ptolemy II type system that enables the definition of a concept lattice and the specification of constraints and acceptance criteria. The lowest level of the tool is a set of Java base classes for defining the lattice, constraints, and acceptance criteria. We have provided as well a set of model elements that can be incorporated with a Ptolemy II model that associate all of these objects with the model. Thus, a model designer can browse from a library of preconfigured property systems, and choose to use those that are useful.

Defining a property system requires a fair amount of work. A property system can be specific to a particular model, or it can be provided in a library of property systems for use with multiple models. Constraints that are specific to a particular model element, like the Integrator above, need to be part of the property system. We have developed an **adapter pattern** that facilitates associating constraints with preexisting components in a library. A key concern is that specifying constraints for model elements requires considerable expertise. We are exploring visual specifications of the concept lattice and constraints in order to improve usability. Another key concern is to be able to define reasonable default constraints that apply to modeling elements that are added after the property system is defined.

We provide a few generic mechanisms that make it easier to define property systems for complex models. For example, many models have modal behavior,

as illustrated in Figure 7. A modal model is a finite state machine (FSM) where each state may contain refinement models. The public interface (e.g. ports and parameters) of the modal model is shared across its refinement models. Each refinement defines the behavior of the modal model component when it is in that mode. A reasonable default strategy is that the constraints of the modal model should be the conjunction of the constraints of the refinements. While our framework permits overriding this default, most model builders will likely find it to be exactly what they want. An interesting extension would be to combine property analysis with model checking to get less conservative analysis.

Another generic mechanism we provide concerns arithmetic expressions. Figure 8, for example, contains a component labeled " Estimate Current Position," which is an instance of the Ptolemy II Expression actor, whose behavior is given by the expression shown in its icon. The constraints of Table 1 apply equally well to nodes of the abstract syntax tree (AST) of such expressions as to actors that add, subtract, multiply, or divide signal values. Hence, property inference and checking works automatically across such expressions. Again, we provide reasonable defaults for setting up the constraints, but the framework supports fine-grained customization to allow easy experimentation.

7 Assessment and Related Work

Much work in formal concept analysis attempts to extract an ontology from a set of object instances. It is more concerned with concept mining or clustering. Our property analysis, on the other hand, infers concept values for model objects based on a given ontology specification. Our focus, therefore, is on facilitating correct modeling by providing better model engineering tools that, like type systems, expose errors early in the design process and facilitate transformation.

Our work can also be viewed as providing a mechanism for incremental or partial construction of a metamodel. A traditional metamodel is more complete than our property systems need to be. A simple property system can be associated with a complex model and incrementally elaborated as the model evolves.

Our property systems are comparable to ontology modeling supported by OWL-protègè and EMF. These tools provide a flexible set of primitives to model complex ontologies. Like them, our lattice ontology description is based on the principle of modeling concepts and relationships. OWL leverages description logic for specifying relationship between classes of concepts. EMF specializes in a subset of relationships borrowed from UML to provide useful features such as model querying and model-to-text support. Our lattice ontology can be viewed as a specialization that restricts ontologies to a lattice structure and constraints to those compatible with efficient inference and checking. Our objectives are also similar to [21], but our lattice foundation ensures unique inference results, supports cyclic dependencies, and scales to large models.

There are a number of obvious extensions to this work. For example, our property system with the lattice in Figure 4 stops short of checking units, although limited forms of such checks are known to be possible [22]. Our ontology

includes concepts like "speed," but not "meters per second" or "miles per hour." An open question is the extent to which our lattice ontology approach can be extended to include units. Most unit analysis systems we are aware of check for consistent use of units at run time, not at compile time. We are aware of three exceptions: a static unit system in Ptolemy II created by Roland Johnson [unpublished], the SIunits library [23], which uses C++ templates, and SCADE [24]. Brown's approach in [23] relies on the type checking of C++. However, the C++ type system in general does not conform with our lattice structure (witness multiple inheritance), so such an approach may not yield unique solutions.

Schlick, et al. in [24] point out that unit checkers face a fundamental problem with "ambiguous units" like work and torque, both of which are Newton-meters. They suggest introducing "radial meters" to disambiguate the two, suggesting that associating more general ontology information with units is useful. Their mention of multiple disjoint uses of dimensionless numbers also reinforces this need for more general ontology information.

Another interesting obvious extension is to support infinite concept lattices. The Ptolemy II type system already does this, in order to support composite types such as arrays and records. Inference in such systems is known to become undecidable in general (witness dependent types), but practical heuristics lead to very usable inference algorithms, at least for type systems. One key question is whether such heuristics would work for domain-specific property systems. It is also challenging to find or invent mechanisms for model builders to define infinite lattices easily and specify constraints over them.

8 Conclusions

We have described a strategy for annotating models with semantic information and automatically performing inference and consistency checking. Our mechanism is scalable and customizable, and thus provides a foundation for research in domain-specific model ontologies and model engineering. Its mathematical foundation ensures that inference results are unique. A model builder can specify just a few semantic annotations, and the implications of these annotations throughout the model are automatically inferred. This will expose modeling errors early, will help designers to better understand their models, and help design teams to agree on interfaces between subsystems, on design concepts, and on terminology.

References

1. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-integrated development of embedded software. Proceedings of the IEEE **91**(1) (2003) 145–164
2. Jantsch, A.: Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation. Morgan Kaufmann (2003)
3. Lee, E.A.: Model-driven development - from object-oriented design to actor-oriented design. In: Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (The Monterey Workshop), Chicago (2003)

4. Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits, Systems, and Computers **12**(3) (2003) 231–260
5. Bock, C.: SysML and UML 2 support for activity modeling. Syst. Eng. **9**(2) (2006) 160–186
6. Rumbaugh, J.: The unified modeling language reference manual, second edition. Journal of Object Technology **3**(10) (2004) 193–195
7. OMG: System modeling language specification v1.1. Technical report, Object Management Group (2008)
8. Hardebolle, C., Boulanger, F.: Modhel'x: A component-oriented approach to multi-formalism modeling. In: MODELS 2007 Workshop on Multi- Paradigm Modeling, Nashville, Tennessee, USA, Elsevier Science B.V. (2007)
9. Pree, W., Templ, J.: Modeling with the timing definition language (tdl). In: Automotive Software Workshop San Diego (ASWSD) on Model-Driven Development of Reliable Automotive Services. LNCS, San Diego, CA, Springer (2006)
10. Herrera, F., Villar, E.: A framework for embedded system specification under different models of computation in SystemC. In: Design Automation Conference (DAC), San Francisco, ACM (2006)
11. Sander, I., Jantsch, A.: System modeling and transformational design refinement in ForSyDe. IEEE Transactions on Computer-Aided Design of Circuits and Systems **23**(1) (2004) 17–32
12. Goessler, G., Sangiovanni-Vincentelli, A.: Compositional modeling in Metropolis. In: EMSOFT, Grenoble, France, Springer-Verlag (2002)
13. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proceedings of the IEEE **91**(2) (2003) 127–144
14. Fritzson, P.: Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley (2003)
15. Lee, E.A., Xiong, Y.: A behavioral type system and its application in Ptolemy II. Formal Aspects of Computing Journal **16**(3) (2004) 210 – 237
16. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17** (1978) 348–375
17. Rehof, J., Mogensen, T.A.: Tractable constraints in finite semilattices. In: SAS '96: Proceedings of the Third International Symposium on Static Analysis, London, UK, Springer-Verlag (1996) 285–300
18. Xiong, Y.: An extensible type system for component-based design. Ph.D. Thesis Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720 (May 1 2002)
19. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer-Verlag, Berlin (1998) Translated by C. Franzke.
20. Bauer, H.: ACC Adaptive Cruise Control. Robert Bosch GmbH (2003)
21. Bowers, S., Ludäscher, B.: A calculus for propagating semantic annotations through scientific workflow queries. In: Workshop on Query Languages and Query Processing (QLQP), Munich, Germany, Springer (2006) 712–723 LNCS 4254.
22. Hayes, I.J., Mahony, B.P.: Using units of measurement in formal specifications. Formal Aspects of Computing Journal **7** (1995) 329–347
23. Brown, W.E.: Applied template meta-programming in SIunits: the library of unit-based computation. In: Workshop on C++ Template Programming, Tampa Bay, FL, USA (2001)
24. Schlick, R., Herzner, W., Sergent, T.L.: Checking SCADE models for correct usage of physical units. In: SAFECOMP. Volume LNCS 4166., Springer (2006) 358 371