This is a chapter from the book

System Design, Modeling, and Simulation using Ptolemy II

First Edition, Version 1.0

Please cite this book as:

Claudius Ptolemaeus, Editor,
System Design, Modeling, and Simulation using Ptolemy II, Ptolemy.org, 2014.
http://ptolemy.org/books/Systems.

15 Ontologies

Patricia Derler, Elizabeth A. Latronico, Edward A. Lee, Man-Kit Leung, Ben Lickly, Charles Shelton

Contents

An **ontology** in information science refers to an explicit organization of knowledge. An ontology can be organized into a graph as a set of **concepts** and the relations between those concepts. By constructing an ontology over a specific domain, a user is formalizing information of that domain in a way that can be shared with others. Models can have annotations added to them that express how they are used with respect to an ontology. Ontology-based annotations are a form of model documentation. Like type signatures, they can express the intended use of a model, but with respect to the domain of the ontology rather than to the type system.

A **static analysis** of a program or model is a check that can be run at compile time. Ptolemy II's type checker (see Chapter 14) is one example of a static analysis. It infers the data types used throughout a model and checks for consistency. In fact, the Ptolemy II type system is an ontology. It is an organization of knowledge about the data that a model operates on. The ontology checker described in this chapter also performs inference based on the annotations, and then checks consistency. But it is not constrained to checking data types. Instead, ontologies can be used to express static analyses from a variety of user-defined domains, including, for example:

- **units checking**: determining whether the units of data are consistent;
- **constant analysis**: determining what data in a model is constant, and what data varies in time;
- **taint analysis**: determining whether values in a data stream are influenced by an untrusted source; and
- **semantics checking**: determining whether the meaning of data produced by one component is consistent with the meaning assumed by another component that uses the data.

Such analyses can expose a variety of modeling errors.

Example 15.1: A portion of a model of a multi-tank fuel system in an aircraft (Moir and Seabridge, 2008) is shown in Figure 15.1. This model has three actors, where the ports are labeled with names that suggest the intended meaning and units of the data that are exchanged between actors. The model has three types of errors. It has units errors, where for example one component gives the level of a tank in gallons to a component that assumes that the level is being given in kilograms. (The latter is often a better choice, since amount of fuel in gallons varies with temperature, whereas the amount in kilograms does not.) It also has semantics

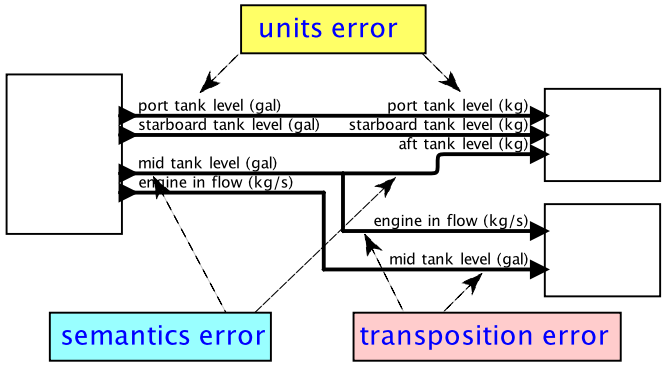

Figure 15.1: An illustration of some of the sorts of errors that can be caught by an ontology system.

errors, where a component gives the level of mid tank to a component that assumes it is seeing the level of the aft tank. And it has a transposition error, where a level and a flow are exchanged.

Such modeling errors are extremely easy to make and can have devastating consequences. This chapter gives an overview of how to construct ontologies and use them to prevent such errors.

15.1 Creating and Using Ontologies

The ontologies package provides an analysis that can be run on top of an existing model, so the first step is to create a Ptolemy II model on which we can run our analysis. In this section, we use a rather trivial model and a rather trivial ontology to illustrate the mechanics of construction of an ontology and the use of a solver. We will then illustrate a less trivial ontology that is practical and useful for catching certain kinds of modeling errors.

Example 15.2: Figure 15.2, shows a simple model with both constant and non-constant actors. The constant actors produce a sequence of output values that are all the same. For this model, we will show how to create a simple analysis that checks which signals in the model are constant. To do this, we will first define an ontology that distinguishes the concept of "constant" from "non constant." We will then define constraints for actors used by the model, and finally, we will invoke the solver.

Sidebar: Background on the Ontology Framework

The approach to ontologies described in this chapter was first given by Leung et al. (2009). They build on the theory of Hindley-Milner type systems (Milner, 1978), the efficient inference algorithm of Rehof and Mogensen Rehof and Mogensen (1996), the implementation of this algorithm in Ptolemy II (Xiong, 2002), and the application of similar mathematical foundations to formal concept analysis (Ganter and Wille, 1998).

An interesting extension of this basic mechanism, devised by Feng (2009), uses ontologies to guide model-based **model transformation**, where a Ptolemy II model modifies the structure of another Ptolemy II model. For example, the constant analysis described in Section 15.1 can guide a model optimization that replaces all constant subsystems with a Const actor. Also, Lickly et al. (2011) show how an infinite lattice can be used to not just infer that a signal is constant, but also to infer its value. Lickly et al. (2011) also show various other ways to use infinite lattices, including unit systems. They also show how ontologies work with structured types such as records.

The **Web Ontology Language** (**OWL**) is a widely-used family of languages endorsed by the World Wide Web Consortium (**W3C**) for specifying ontologies. OWL ontologies, like ours, form a partial order with a top and bottom element, but unlike ours, they are not constrained to be a lattice. Hence, the efficient inference algorithm of Rehof and Mogensen cannot always be applied. Nevertheless, a very useful extension of the mechanisms described in this chapter would be to export and import OWL ontologies. The **Eclipse Modeling Framework** (**EMF**) also specifies ontologies through the notion of classes and subclasses. Many Eclipse-based tools have been developed supporting it, so again it would be useful to build bridges.

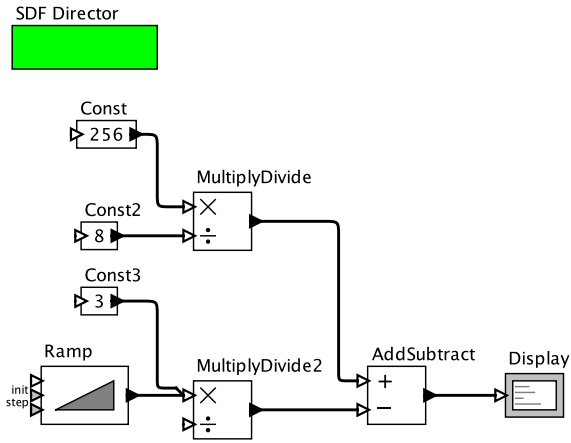

Figure 15.2: A simple Ptolemy model made of constant and non-constant actors and containing an ontology solver that can determine which signals are constant.

15.1.1 Creating an Ontology

In order to create the analysis, the first step will be to add the solver that will perform our analysis. In this case, we will drag in the **LatticeOntologySolver** actor to our model, as shown in Figure 15.3. This is where we will add all of the details of how our analysis works. These include the lattice that represents the concepts that we are interested in, and the constraints that actors impose on those concepts. In our case, the lattice will specify whether a signal is constant or not, and the constraints will provide information about which actors produce constant or non-constant signals.

As shown in Figure 15.3, if you open the LatticeOntologySolver, you get an editor with a customized library for building analyses. At a minimum, an analysis requires an ontology. Figure 15.4 illustrates the steps in constructing one. First, drag into the blank editor an **Ontology**. Open the ontology and drag **Concept**s into it from the library provided by the ontology editor.

First, we should assign meaningful names to our concepts. In Figure 15.5, we have renamed Top to NonConstant, Bottom to Unused, and Concept to Constant. We have also edited the parameters to the NonConstant concept to change its color to a light red, and to check the *isAcceptable* parameter, which visually removes the bold outline. These con-

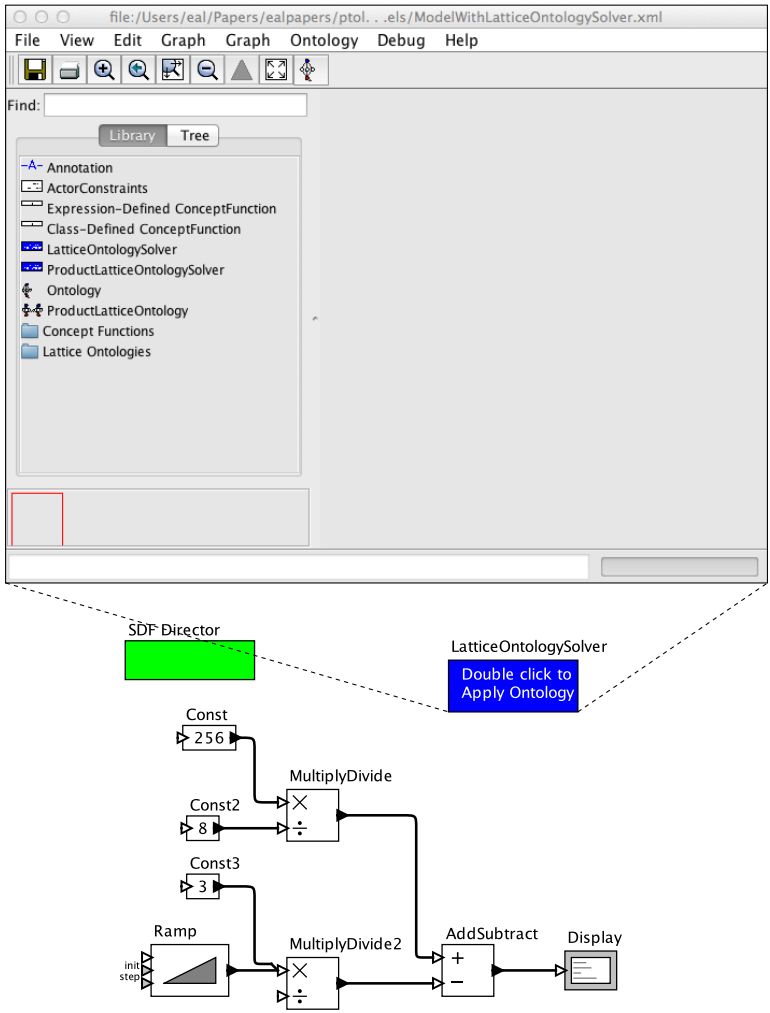

Figure 15.3: A model with a blank ontology solver.

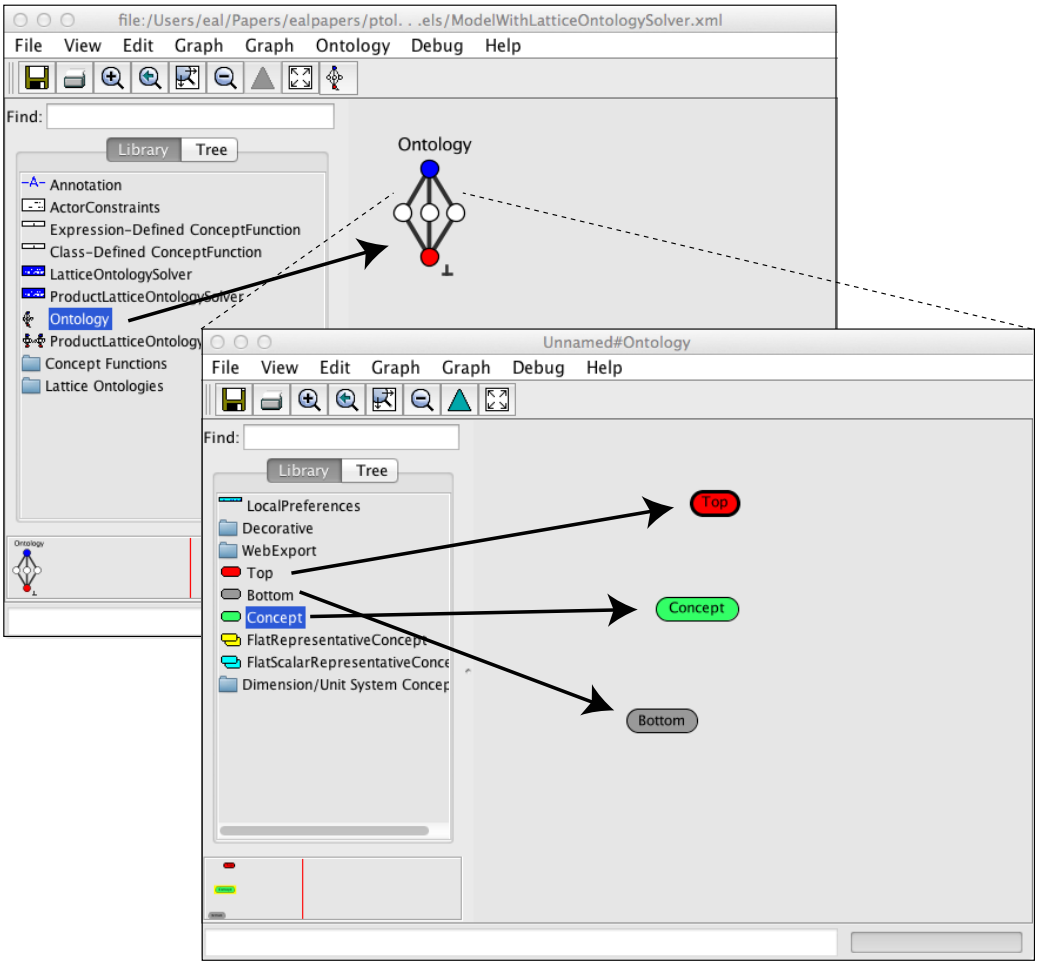

Figure 15.4: Steps in the construction of an ontology.

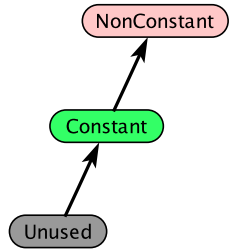

Figure 15.5: The lattice used for constant analysis.

cepts will be associated with ports in our model, and when *isAcceptable* is unchecked (the outline on the concept is bold), then it is an error in the model for any port to be associated with the concept. In our case, it is not an error for a port to be NonConstant, so we make the value of this parameter true.

Here, we include not only the concepts of Const and NonConst, but also explicitly include a notion of Unused. This concept will be associated with ports that are simply not participating in the analysis. We will use the NonConstant concept to represent any signal that may or may not be constant, so a better name might be PotentiallyNonConst or NotNecessarilyConst.

The last step in building an ontology is to establish relationships between the concepts. Do this by holding the control key (command key on a Mac) and dragging from the lower concept to the higher one. In this case, the relation between Constant and NonConstant is a **generalization relation**. The relations define an order relation between concepts (see sidebar on page 513), where if the arrow goes from concept a to concept b, then $a \leq b$. In this case, Constant $\leq$ NonConstant, and Unused $\leq$ Constant.

The meanings of these relations can vary with ontologies, but it is common for them to represent subclassing, a subset relation, or as an "is a" relation. In the subset interpretation, a concept represents a set, and concept A is less than another B if everything in A is also in B. The "is a" relation interpretation is similar, but doesn't require a formal notion of a set. E.g., if the concept A represents "dog" and the concept B represents "mammal," then it is reasonable to establish on ontology where $A \leq B$ under the "is a" interpretation. A dog is a mammal.

Example 15.3: In Figure 15.5, NonConstant represents anything that may or may not be constant. Hence, something that is actually constant "is a" NonConstant.

The interpretation of the bottom element, Unused in the example, is a bit trickier. Presumably, anything that makes no assertion about whether it is constant or not "is a" NonConstant. Indeed, since the order relation is transitive, this statement is implied by Figure 15.5. But why make a distinction between Unused and NonConstant? We could build an ontology that makes no such distinction, but in such an ontology, the inference engine would infer Constant for any port that imposes no constraints at all. This is probably an error. The bottom element, therefore, is used to indicate that the inference engine has no usable information at all. If a port resolves to Unused, then it is not playing the game. If we wish to force all ports to play the game, then we should set the *isAcceptable* parameter of Unused to false.

The Ptolemy II type system is an ontology, as shown in Figure 14.4. Here, the order relations represent subtyping, which in the case of Ptolemy II is based on the principle of lossless type conversion.

With the concepts as nodes and the relations as edges, the ontology forms a mathematical graph. The structure of this graph is required to conform with that a mathematical lattice (see sidebar on page 513). Specifically, the structure is a lattice if given any two concepts in the ontology, these two concepts have a least upper bound and a greatest lower bound. In this case, conformance is trivial. For example, the least upper bound of Constant and NonConstant is NonConstant. The greatest lower bound is Constant. But it is easy to construct an ontology that is not a lattice.

Example 15.4: Figure 15.6 shows an ontology that is not a lattice. Consider the two concepts, Dog and Cat. There are three concepts that are upper bounds for these two concepts, namely Pet, Mammal, and Animal. But of those three, there is no least upper bound. A least upper bound must be less than all other upper bounds.

If you build an ontology that is not a lattice, then upon invoking the solver, you will get an error message.

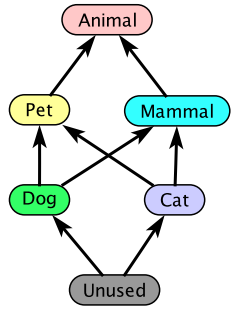

Figure 15.6: An ontology that is not a lattice.

15.1.2 Creating Constraints

Now that we have an ontology, to use it, we need to create constraints on the association between objects in the model and the concepts in the ontology. The most straightforward way to do this is with manual annotations in the model. With large models, however, this technique does not scale well. A more scalable technique is to define constraints that apply broadly to all actors of a class, and to specify default constraints that apply when no other constraints are specified. We begin with the manual annotations, because they are conceptually simplest.

Manual Annotations

The simplest way to relate objects in a model to concepts in the ontology is through manual annotations. Manual annotations take the form of inequality constraints. To create such constraints, find the **Constraint** annotation in the `MoreLibraries→Ontologies` library, and drag it into the model. Then specify an inequality constraint of the form `object >= concept` or `concept >= object`, where `object` is an object in the model (a port or parameter) and `concept` is a concept in the ontology.

Example 15.5: Figure 15.7 elaborates the model of Figure 15.3 by adding four annotations. Each of these has the form

```
port >= concept.
```

Specifically, the output ports of each Const actor are constrained to be greater than or equal to Constant, whereas the output port of the Ramp actor is constrained to be greater than or equal to NonConstant. The latter constraint, in effect, forces the port to NonConstant, since there is nothing greater than NonConstant in the ontology. The former constraints could be elaborated to force the ports to resolve to Constant by adding the complementary inequality, like

```
Constant >= Const.output.
```

However, this additional constraint is unnecessary. The solver will find the *least* solution that satisfies all the constraints, so in this case, since there are no other constraints on the output ports of the Const actors, they will resolve to Constant anyway.

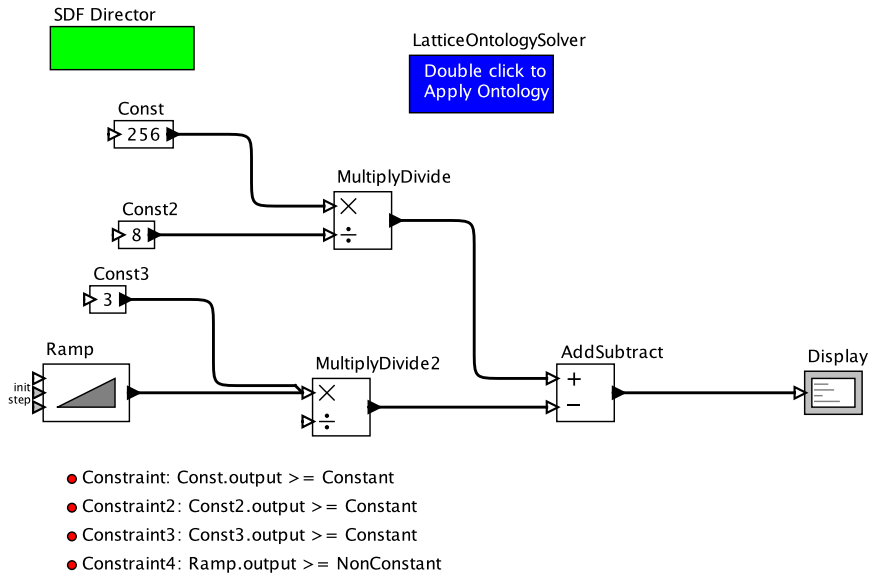

Figure 15.7: A model with manual constraints, using the ontology in Figure 15.5.

Once we have created all the constraints needed in our model, the next step is to run the analysis. The analysis can be run by right-clicking on the LatticeOntologySolver and selecting `Resolve Concepts`, as shown in Figure 15.8. Because this is a common operation, double-clicking on the LatticeOntologySolver will also run the analysis.

Example 15.6: Figure 15.8 includes the results of running the analysis. Notice that each port is annotated with the concept that it is now associated with. In addition, the port is highlighted with the same color specified for the concept in the ontology of Figure 15.5. The output ports of the Const actors, as expected, have resolved to Constant, and the the output of the Ramp has resolved to NonConstant. But more interestingly, downstream ports have also resolved in a reasonable way. The output of the MultiplyDivide is Constant (because both its inputs are Constant), and the the output of MultiplyDivide2 is NonConstant (because one of its inputs is NonConstant). These results are due to default constraints associated with actors, which as explained below, can be customized in an ontology.

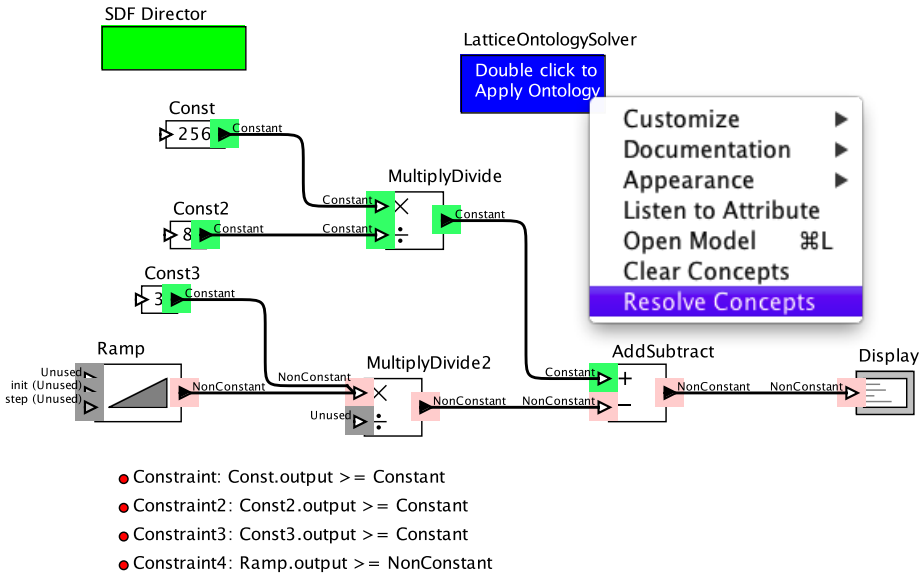

Figure 15.8: Running an analysis. [online]

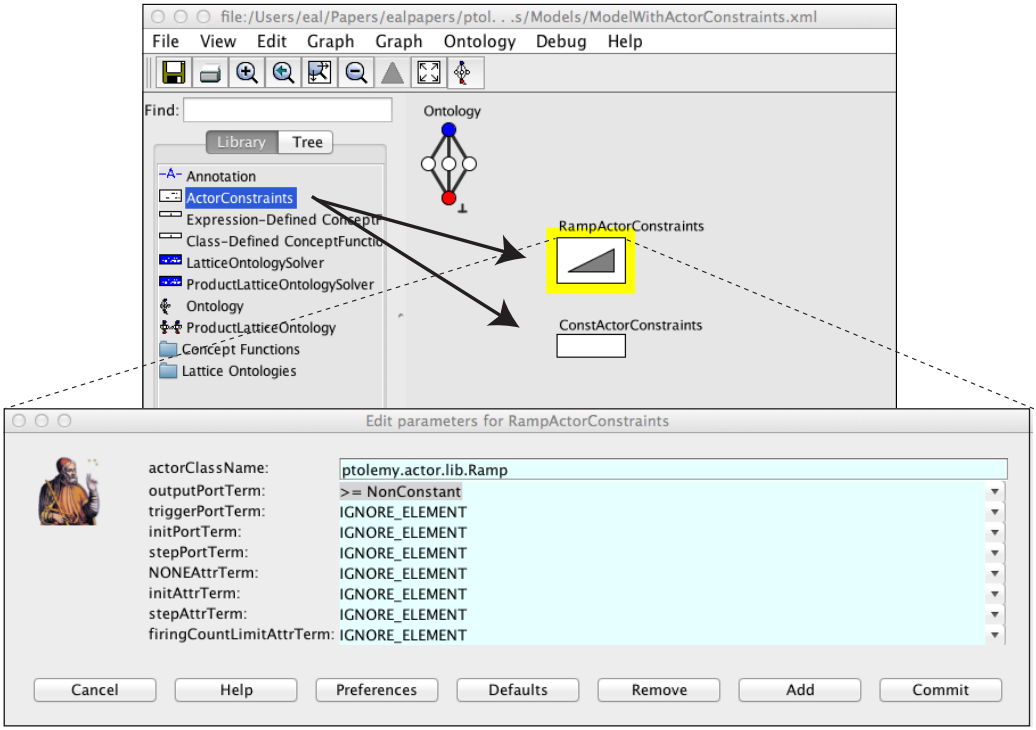

Figure 15.9: Adding actor constraints to an ontology.

Actor-level Constraints

The manual annotations in Figure 15.7 could become very tedious to enter in a large model. Fortunately, there is a convenient shortcut. As part of defining an ontology, we can specify constraints that will be associated with all instances of a particular actor class. Figure 15.9 shows how to do this. Inside the LatticeOntologySolver, we add one instance of **ActorConstraints** for each actor for which we want to specify default constraints.

Setting the *actorClassName* parameter of the ActorConstraints to the fully qualified class name of the actors to be constrained causes both the name and the icon of the instance of ActorConstraints to change to match the class of actors that it will constrain.

Example 15.7: Figure 15.9 shows two instances of ActorConstraints that have been dragged into the LatticeConstraintsSolver. The top one has the *actorClassName* parameter set to `ptolemy.actor.lib.Ramp`, the class name of the Ramp actor. (To see the class name of an actor, linger over it in Vergil.)

Once the class name has been set, upon re-opening the parameter dialog, a new set of parameters will have appeared, one for each port belonging to instances of the actor class, and one for each parameter of the actor.

Example 15.8: Figure 15.9 shows these parameters for the Ramp actor. Here we can see that constraints have been set that will force the *output* port to be greater than or equal to NonConstant, and that all other constraints have been set to `IGNORE_ELEMENT`. Once a similar constraint has been added to the ConstActorConstraints component, constraining the output ports of instances of Const to be `>= Constant`, then the four constraints at the bottom of Figure 15.7 are no longer necessary. They could be removed, and the results of the analysis will be the same as in Figure 15.8.

The constraints associated with a port or parameter can take any one of the following forms:

- `NO_CONSTRAINTS` (the default)
- `IGNORE_ELEMENT`
- >= *concept*
- <= *concept*
- == *concept*

By default, the ActorConstraints actor will set `NO_CONSTRAINTS` as the constraint of each port and parameter. This means that instances of the actor allow their ports and parameters to be associated with any concept. In this ontology, the association will always result in Unused, so we could equally well have left all the constraints at the default `NO_CONSTRAINTS`, except those for the output ports.

Default Constraints

Notice in Figure 15.8 that not only have the outputs of the Const and Ramp actors resolved to the appropriate concepts, but so have those of downstream actors, including MultiplyDivide and AddSubtract. How did this come about?

With respect to this analysis, both the MultiplyDivide actor and the AddSubtract actors behave the same way. Given only constant inputs, they produce a constant output, but given any non-constant input, they produce a (potentially) non-constant output. In terms of the ontology lattice (Figure 15.5), the output concept will always be greater than or equal to all of the input concepts. In other words, the output should be constrained to be greater than or equal to the least upper bound of all the inputs. It turns out that this type of inference behavior is a very common one. For this reason it is the default constraint for all actors. Actors that do not have explicit constraints will inherit this default constraint. This means that we can omit specifying any more ActorConstraints, since the global default constraint is sufficient for all of our remaining actors.

15.1.3 Abstract Interpretation

Identifying signals as either constant or non-constant is a particularly simple form of **abstract interpretation** (Cousot and Cousot, 1977). In abstract interpretation, instead of actually computing the values of variables, we classify the variables in more abstract terms, such as whether their values vary over time. Ontologies can be used to systematically apply more sophisticated abstractions, determining for example whether variables are always positive, negative, or zero. This can be used to expose errors in designs, and also to optimize design by removing unnecessary computations.

> **Example 15.9:** The model in Figure 15.10 produces a constant stream of zeros. Were we to apply the same Constant-NonConstant analysis as before to this model, we would be able to determine that the output is constant. But it would not tell us that the output is a constant stream of *zeros*.

To address this problem, we can use the more elaborate ontology shown in Figure 15.11. This ontology is used to abstract numeric variables as positive, negative, or zero-valued

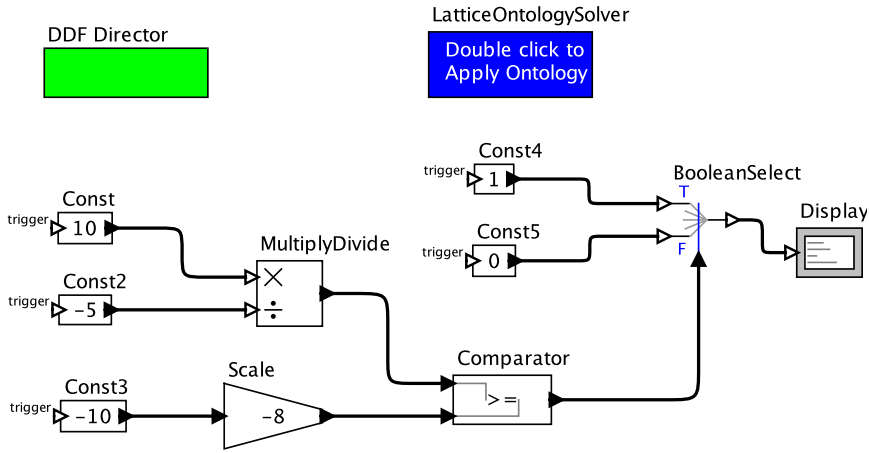

Figure 15.10: A model that produces only zeros.

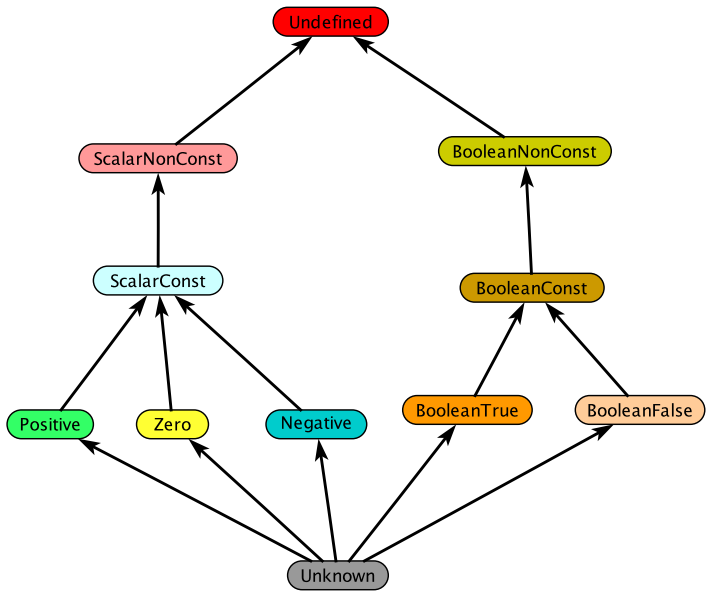

Figure 15.11: An ontology that tracks the sign of numeric variables and the value of boolean variables.

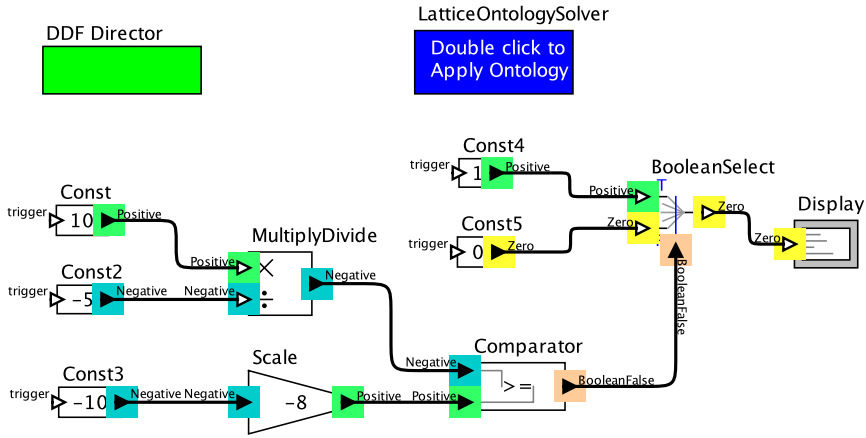

Figure 15.12: Result of applying an analysis based on the ontology in Figure 15.11 to the model in Figure 15.10. [online]

numbers, in addition to whether they are constant or non constant. For boolean-valued variables, if the variable is constant, then it also tracks whether the constant is true or false. With appropriate actor constraints, this ontology can be used to produce the result shown in Figure 15.12, which determines that the output is a constant stream of zeros.

15.2 Finding and Minimizing Errors

In this section, we discuss how to use an analysis to identify errors, and discuss tools available to help in correcting the errors. For this discussion, let us consider the ontology in Figure 15.13. This ontology models physical dimensions. Specifically, it distinguishes the concepts of time, position, velocity, and acceleration. We will show that with appropriate actor constraints, it can use properties of these dimensions in inference.

Example 15.10: Figure 15.14 shows a piece of a larger model that shows interesting inference of dimensions. This is a model of a car with a **cruise control**, where the input is a desired speed, and the outputs are acceleration, speed, and position. In this model, when velocity is divided by time, the result is acceleration. When

acceleration is integrated over time, the result is velocity. When velocity is integrated over time, the result is position. This analysis relies on actor constraints for arithmetic operations and integrators.

The ontology in Figure 15.13 explicitly includes the concepts Unknown and Conflict. The difference between Unknown and Conflict is subtle and deserves mention. Conflict

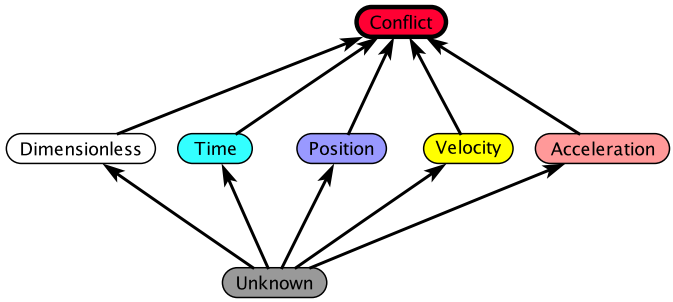

Figure 15.13: An ontology for analyzing physical dimensions.

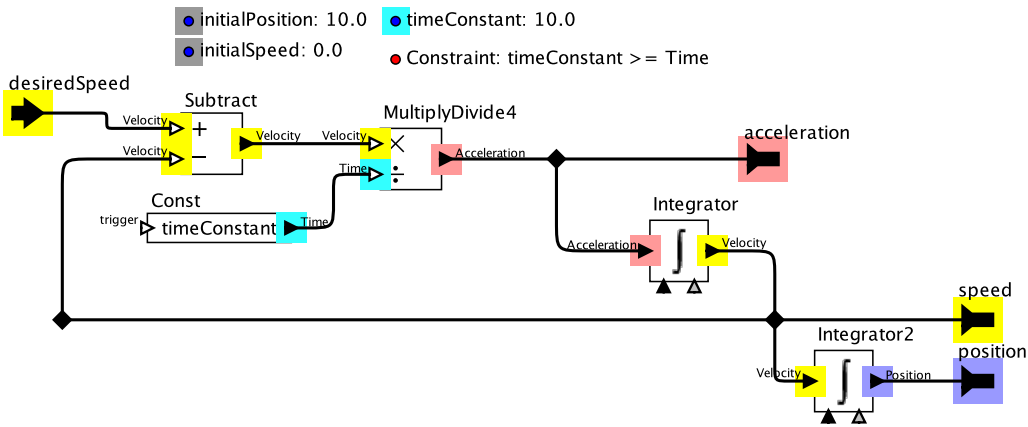

Figure 15.14: A piece of a larger model that shows interesting inference of dimensions. [online]

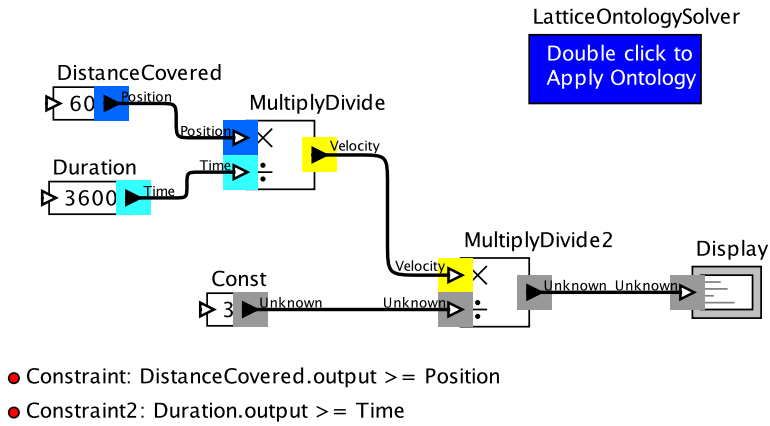

Figure 15.15: A model analyzing physical dimensions with too few annotations. Running the analysis reveals where additional annotations are needed. [online]

represents a situation where the analysis has detected that a given signal cannot have any of the dimensions; thus, the analysis has found an error in the model due to conflicting use of dimensions. For this reason, in this ontology, the *isAcceptable* parameter of Conflict is set to false, resulting in a bold outline in Figure 15.13. In addition, if any port resolves to Conflict, then running the analysis will report an error.

In the ontology in Figure 15.13, Unknown represents a case where the analysis cannot say conclusively anything about the given signal; this means that the property being analyzed cannot be proved with the given assumptions. Unknown in this case plays a similar role as Unused in Figure 15.5. When a port resolves to Unknown, this can point to there not being enough constraints in the model. This may or may not be an error, so *isAcceptable* is left at its default value of true.

Example 15.11: An example of an underconstrained model is shown in Figure 15.15. Here, the model divides a velocity by a time to get an acceleration, but the constraint specifying the dimension of the time value produced by the Const actor has been omitted. When running the analysis, we get results shown the figure. The lack of information propagates throughout the model. This can be fixed, of course, by adding a manual annotation to the model as shown in Figure 15.16.

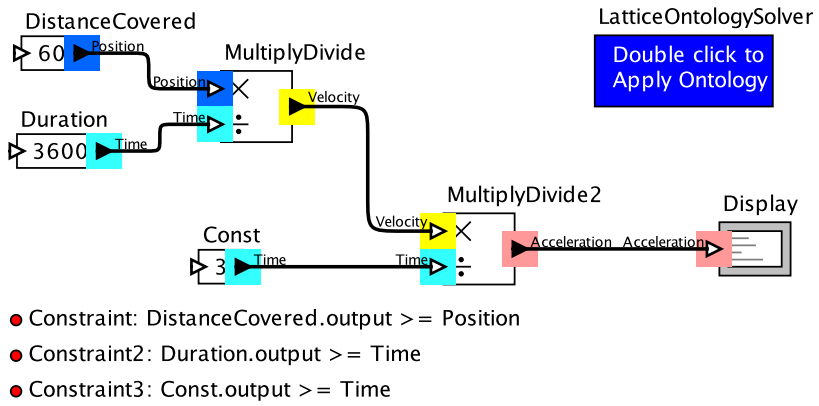

Figure 15.16: Adding an additional constraint allows for a complete analysis. [online]

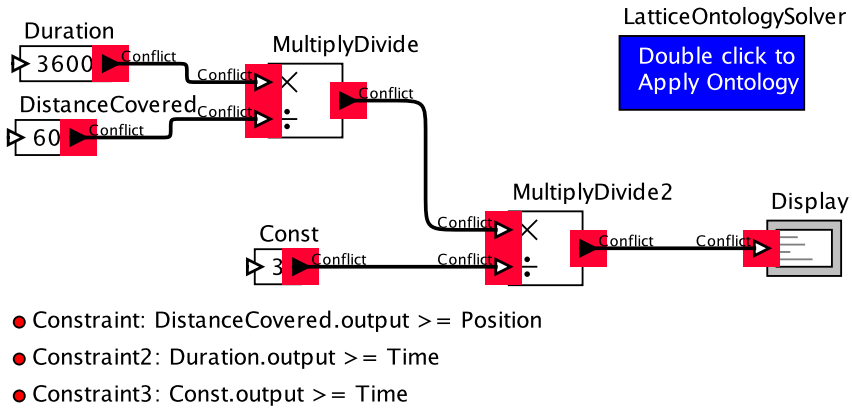

Figure 15.17: An example model with conflicting dimensions due to an error in the model. Running the analysis on this model shows that the whole model is in conflict. [online]

In general, overconstrained models can be more difficult to deal with.

Example 15.12: An example of an overconstrained model is shown in Figure 15.17. Here, the model builder has incorrectly divided a time by a position, where presumably the reverse operation was intended. The result is conflicts throughout the model.

The conflict in the previous example arises because of the ActorConstraints that are defined for the MultiplyDivide actor as part of the ontology (see Figure 15.9). Specifically, the ontology gives for the *outputPortTerm* the following expression:

```
>=
(multiply == Unknown || divide == Unknown) ? Unknown :
(multiply == Position && divide == Time) ? Velocity :
(multiply == Velocity && divide == Time) ? Acceleration :
(multiply == Position && divide == Velocity) ? Time :
(multiply == Velocity && divide == Acceleration) ? Time :
(divide == Dimensionless) ? multiply :
Conflict
```

This constrains the output concept as a function of the input concepts. If the *multiply* input is Position and the *divide* input is Time, for example, this expression evaluates to Velocity. If *divide* input is Dimensionless, then it evaluates to whatever *multiply* is. If *multiply* is Time and *divide* is Position, it evaluates to Conflict.

In the example of Figure 15.17, notice that conflicts propagate upstream as well as downstream. In this example, we have set the *solverStrategy* of the LatticeOntologySolver to `bidirectional`.* The default value of this parameter is `forward`, which means that when there is a connection from an output port to an input port, then the concept associated with the input port is constrained to be greater than or equal to the concept associated with the output port. When the parameter value is set to `bidirectional`, then the two concepts are required to be equal. With the `bidirectional` setting, constraints propagate upstream in a model just as easily as downstream. So although this is reasonable to

*Since double clicking on the LatticeOntologySolver runs the analysis, double clicking cannot be used to access the parameters. Instead, hold the alt key while you click to access the parameters.

do for dimension analysis, it makes it difficult to see which part of the model is causing the error.

By default, the solver finds the least solution that satisfies all the constraints.[†] Hence, the way that our analysis deals with conflicting information is to promote signals to the least upper bound of the conflicting concepts (or greatest lower bound when computing greatest fixed points).

The analysis is able to correctly detect the error, as shown in Figure 15.17, but there is a problem. Unlike the underconstrained case in which the error was relatively contained, in this case Conflict propagates throughout the model, making it very difficult to see where the source of the error is. In this simple example it is not too difficult to find the error, but as models grow, so does the difficulty in tracking down the source of an error of this type.

In order to address this problem, we have introduced an error minimization algorithm. This algorithm is implemented in the **DeltaConstraintSolver** actor, which is a subclass of LatticeOntologySolver. It can be found in the same `MoreLibraries→Ontologies` library. The DeltaConstraintSolver offers a `Resolve Conflicts` item in the context menu in addition to the `Resolve Concepts` that we used before.

Given a model in which at least one signal resolves to an unacceptable solution, the algorithm realized by `Resolve Conflicts` finds a subset of those constraints with the property that removing any additional constraint does not produce any error. Highlighting the result of running the analysis with these constraints will highlight only a subsection of the model that contains an error. In practice, this means in contrast to the full analysis, where the highlighted result shows many errors throughout the model as shown in Figure 15.17, the modified algorithm highlights only a single path through the model that contains an error, as shown in Figure 15.18. In this example, only the Duration, DistanceCovered, and MultiplyDivide actors (each of which is an instance of Const) are highlighted, since they are sufficient to cause the error. The unhighlighted actors (in this case the Const, MultiplyDivide2, and Display) are not necessary to cause the error, so the model builder can ignore them in trying to find the cause of the error.

In this case, the port that was highlighted with the erroneous concept (Conflict) was the location of the error, but in general, this need not be the case. The only guarantee that is made is that there is an error somewhere in the path of all highlighted signals. This

[†]This can be changed by changing the *solvingFixedPoint* parameter of the LatticeOntologySolver from the default `least` to `greatest`. In that case, the solver will find the highest solution in the lattice that satisfies all the constraints.

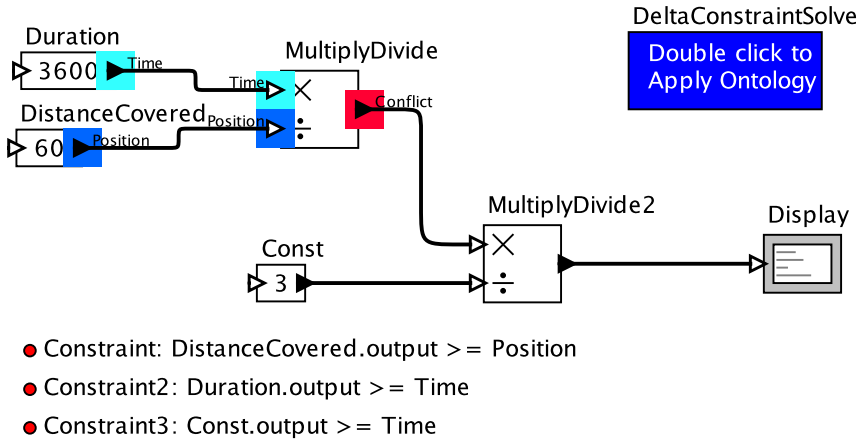

Figure 15.18: With our error minimization algorithm, finding errors is easier. [online]

means that in general, all of the signals that are highlighted may need to be checked in order to find the source of the error. The gain of this technique comes from the many unhighlighted actors in the model that can be completely ignored.

15.3 Creating a Unit System

In the previous section, we saw an analysis that checked that the dimensions of a system were being used in a consistent way, to avoid errors such as integrating a velocity and expecting to get an acceleration value out. A similar but more insidious type of error is when two signals have the same dimensions, but different units, such as feet vs. meters, or pounds vs. kilograms. Since this is a more subtle problem – results can deviate by only a small scaling factor – it is even more desirable to check these types of properties automatically. Section 13.7 describes a built-in unit system provided in Ptolemy II. But a units system is just another ontology, and the ontology framework can be used to create specialized units systems.

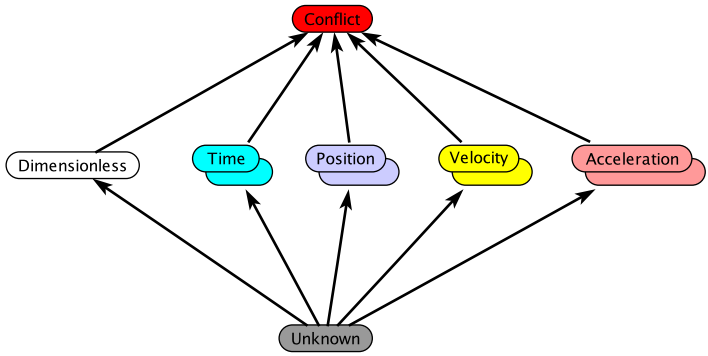

Figure 15.19: A unit system for physical units.

15.3.1 What are Units?

In order to discuss what units are, we first need to discuss how units differ from one another. There are two ways that units can differ. They can be different measures of the same quantity, like feet and meters, or they can be measures of fundamentally different quantities, like feet and seconds. Differences of the second type are exactly what is captured in the dimension ontology, so our approach to units will be similar, but extended to deal with different units within a single dimension.

Ptolemy II supports creating a **units system ontology** with the same freedom as other ontologies. Users can choose the units and dimensions that are appropriate to analyze the model at hand, and thus make the ontology only as complicated as it needs to be to perform the desired analysis. Since we have already discussed a physical dimension ontology that had concepts for Position, Velocity, Acceleration, etc., we will continue to use an example unit ontology where the units are drawn from the same dimensions, as shown in Figure 15.19. However, instead of building our ontology using only simple instances of Concept, we use instances of **DerivedDimension** and **Dimensionless**. These are concepts specialized to support units, and are selected from the `Dimension/Unit System Concepts` sublibrary provided by the ontology editor (see Figure 15.4).

As shown in Figure 15.19, DerivedDimension has a slightly different icon, a double oval, suggestive that it is in fact a **representative concept**, a single object that stands in for a family of concepts. DerivedDimension has some parameters that we set and parameters

that we must add to build a unit system. We outline what this looks like here, but to build a unit system from scratch, you will need to refer to the documentation.

15.3.2 Base and Derived Dimensions

The dimensions from which units are drawn are split into two types, **base dimensions**, and **derived dimensions**. Base dimensions are dimensions that cannot be broken down into any smaller components, such as Time, whereas derived dimensions can be expressed in terms of other dimensions. For example, Acceleration can be expressed in terms of Position and Time. Note that there is no restriction on which dimensions can be base dimensions or derived dimensions. There is no technical reason that a model builder could not define Acceleration to be a base dimension and derive Position, although it is not the natural choice in this situation. When defining a base dimension, the user only needs to specify the units within that dimension. This is done by adding parameters to the DerivedDimension concept.

Example 15.13: Figure 15.20 shows parameters that have been added to the Time dimension of Figure 15.19. Here, the user specifies the scaling factor of all units

secFactor:	1.0
hrFactor:	3600*secFactor
dayFactor:	24*hrFactor
sec:	{ Factor = secFactor }
ms:	{ Factor = 0.001*secFactor }
us:	{ Factor = 1E-06*secFactor }
ns:	{ Factor = 1E-09*secFactor }
minute:	{ Factor = 60*secFactor }
hr:	{ Factor = hrFactor }
day:	{ Factor = dayFactor }
yrCalendar:	{ Factor = 365.2425*dayFactor }
yrSidereal:	{ Factor = 31558150*secFactor }
yrTropical:	{ Factor = 31556930*secFactor }

Figure 15.20: An example of the specification of the Time base dimension.

within that dimension with respect to one another, using named constants as needed to make the calculations clearer.

The Time dimension is a base dimension. The definition of a derived dimension is slightly more involved, since a derived dimension must explicitly state which dimensions it is derived from. Derived dimensions must specify both how the derived dimension is derived from other dimensions, and how each of its individual units are derived from units of those other dimensions.

Example 15.14: Figure 15.21 shows an example of the specification of the Acceleration dimension of Figure 15.19. Here, the first lines show that an Acceleration is built from the Time and Position base dimensions, and that an acceleration has units of $position/time^2$. The rest of the specification shows how individual units of acceleration are related to units of position and time.

The main benefit of specifying units this way is that we can infer the constraints for multiplication, division, and integration, which are used in many actors.

```
dimensionArray:  { {Dimension = "LengthConcept", Exponent = 1}, {Dimension = "TimeConcept", Exponent = -2} }
LengthConcept:   Position
TimeConcept:     Time
m_per_sec2:      { LengthConcept = {"m"}, TimeConcept = {"sec", "sec"} }
cm_per_sec2:     { LengthConcept = {"cm"}, TimeConcept = {"sec", "sec"} }
ft_per_sec2:     { LengthConcept = {"ft"}, TimeConcept = {"sec", "sec"} }
kph_per_sec:     { LengthConcept = {"km"}, TimeConcept = {"hr", "sec"} }
mph_per_sec:     { LengthConcept = {"mi"}, TimeConcept = {"hr", "sec"} }
```

Figure 15.21: An example of the specification of the Acceleration derived dimension.

15.3.3 Converting Between Dimensions

In the case that units are used inconsistently, there is a error in the model. Not all units errors are the same, however. Units errors that result from interchanging signals with different dimensions point to a model whose connections must be changed, but unit errors from interchanging signals with different units of the same dimensions can be fixed by simply converting the units from one form to another. While it is technically possible to perform this type of conversion automatically, Ptolemy does not do this. Unit errors are an error in modeling, and model errors should never be concealed from model builders.

In keeping with this philosophy, a **UnitsConverter** actor[‡] must be explicitly added to a model in order to perform conversion between two units of the same dimension. This conversion happens both during the analysis, when the actor creates constraints on the units of its input and output ports, and during runtime, when the actor performs the linear transform from one unit to the other. Since the ontology knows the conversion factors between components, model builders need only specify the units to convert between, rather than the logic for conversion that would need to be specified in order to do conversion completely manually.

The parameters of the UnitsConverter actor, shown in Figure 15.22, include the *unitSystemOntologySolver*, which refers to the name of the LatticeOntologySolver of the units system analysis. (The name in this case is DimensionAnalysis.) Since it is only possible

[‡]which can be found in `MoreLibraries→Ontologies`

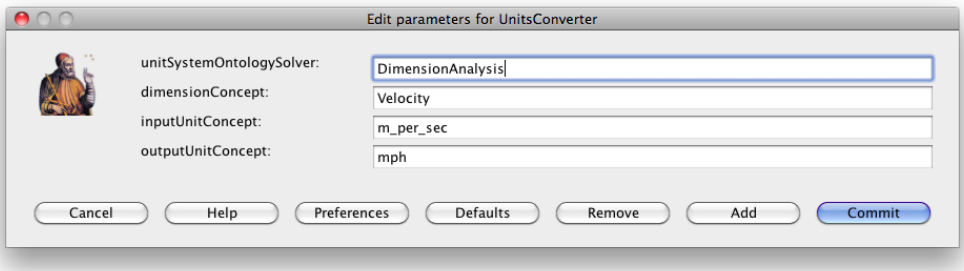

Figure 15.22: Using the UnitsConverter requires setting the name of the associated solver, as well as the input and output units.

to convert between units of the same dimension, the dimension is only specified once, in the *dimensionConcept* parameter, and the individual units of the input and output are specified in the *inputUnitConcept* and *outputUnitConcept* parameters respectively.

Example 15.15: Assume that we have a model that makes the unit error shown in Figure 15.23, where a velocity is provided in meters per second rather than miles per hour. By adding in a UnitsConverter as shown in Figure 15.24, and setting the parameters to convert from Velocity_mph to Velocity_m_per_sec as shown in Figure 15.22, we can create a model that both passes our units analysis and performs the conversion at runtime. Since the ontology analysis is not required to pass before running a model, the version of the model in Figure 15.23 without the UnitsConverter actor can still run. It produces an incorrect output value, however, since it treats the velocity value in Const as expressed in miles per hour as one in meters per second.

15.4 Summary

One of the key challenges in building large, heterogeneous models is ensuring correct composition of components. The components are often designed by different people, and their assumptions are not always obvious. Customized, domain-specific ontologies offer a powerful way to make assumptions clearer. Applying such ontologies in practice, however, can be very tedious, because they typically require the model builders to extensively annotate the model, decorating every element of the model with ontology information. The infrastructure described in this chapter leverages a very efficient inference algorithm that can significantly reduce the effort required to apply an ontology to a model. Far fewer annotations are required than is typical because most ontology associations are inferred.

Domain-specific ontologies and the associated constraints, however, can get quite sophisticated. The vision here is that libraries of ontologies, constraints, and analyses will be built up and re-used. This is certainly possible with the unit systems and dimension systems, but it also seems possible to construct libraries of analyses that are industry- or application-specific. Since these analyses are simply model components, they are easy to share among models within an enterprise.

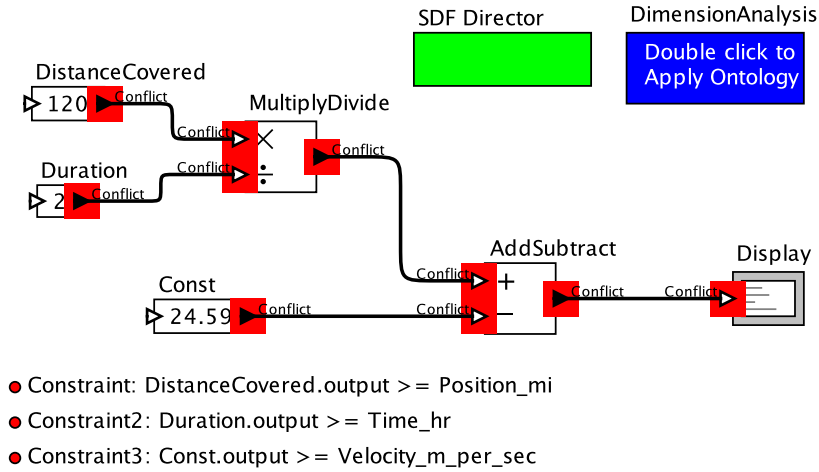

Figure 15.23: An example that adds a quantity in miles per hour to one in meters per second without conversion, resulting in conflicts throughout the model. [online]

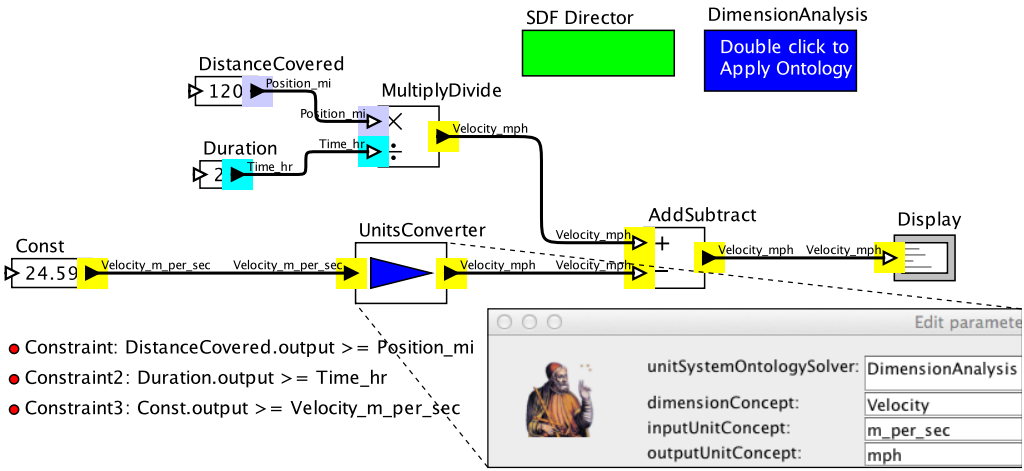

Figure 15.24: A model that uses the UnitsConverter actor to convert from meters per second to miles per hour. [online]

More interestingly, simple ontologies can be systematically combined to create more sophisticated ontologies, including ones where there are constraints that reference more than one ontology. For example, the ontology in Figure 15.11 could be factored into two simpler ontologies, once that refers to scalars and one that refers to booleans, and a product ontology could be defined in terms of these two simpler ontologies. The interested reader is referred to the Combining Ontologies chapter of Lickly (2012).