# A Practical Ontology Framework for Static Model Analysis

*Ben Lickly*
*Charles Shelton*
*Elizabeth Latronico*
*Edward A. Lee*

Electrical Engineering and Computer Sciences
University of California at Berkeley



Technical Report No. UCB/EECS-2011-33
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-33.html

April 26, 2011

Copyright © 2011, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

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 Founda- tion, the U.S. Army Research Office, the U.S. Air Force Office of Scientific Research, the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota.

# A Practical Ontology Framework for Static Model Analysis *

Ben Lickly
University of California, Berkeley
Berkeley, CA, USA
blickly@eecs.berkeley.edu

Charles Shelton
Bosch Research
Pittsburgh, PA, USA
Charles.Shelton@us.bosch.com

Elizabeth Latronico
Bosch Research
Pittsburgh, PA, USA
Elizabeth.Latronico@us.bosch.com

Edward A. Lee
University of California, Berkeley
Berkeley, CA, USA
eal@eecs.berkeley.edu

## ABSTRACT

In embedded software, there are many reasons to include concepts from the problem domain during design. Not only does doing so make the software more comprehensible to those with domain understanding, it also becomes possible to check that the software conforms to correctnesses criteria expressed in the domain of interest. Here we present a unified framework that enables users to create ontologies representing arbitrary domains of interest as well as analyses over those domains. These analyses may then be run against software specifications, encapsulated as models, checking that they are sound with respect to the given ontology. Our approach is general, in that our framework is agnostic to the semantic meaning of the ontologies that it uses and does not privilege the example ontologies that we present here. Where practical use-cases and principled theory exist, we provide for the expression of certain patterns of infinite ontologies and ontology compositions. In this paper we present two overarching patterns of infinite ontologies: those containing values, and those containing ontologies recursively. We show how these two patterns map on to use cases of unit systems and structured data types, and show how these can be used over cyber-physical systems examples drawn from automotive and avionic domains. Despite the range of ontologies and analyses that we present here, we see user-built ontologies as a key feature of our approach.

*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, the U.S. Army Research Office, the U.S. Air Force Office of Scientific Research, the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota.

## 1. INTRODUCTION

In building embedded software for cyber-physical systems, there are a host of potential problems as a result of interfacing components together. One of the leading causes of failure is mismatched assumptions between software components [16]. This occurs when the semantics of what one component expects to receive does not match the semantics of what another component produces. One way to check some of these errors is with a type system, which can check that the types of components are consistent with one another. This prevents such egregious examples as having one component produce output as a floating point number and the next component expect an integer, but it ignores an entire class of finer distinctions between signals whose type is the same but only differ with respect to some semantic property known to the model builder.

One type of semantic error that is particularly prevalent is that of mismatched units, which has been found to be a root cause of several high-profile disasters. Among these are the Air Canada Flight 143, which due to a miscalculation of fuel density confusing pounds and kilograms took off with less than half the fuel required [7], and the Mars Climate Orbiter, which crashed into the planet on descent due to a unit error between newtons and pound-force [11].

While traditional software projects often can encode domain information into their object-oriented type hierarchy, model-based engineering rarely is able to use such a large amount of domain information in the construction of their designs. And this is with good reason, because embedded systems are often constrained in terms of resources and unwilling to accept the run-time overhead that traditional object oriented systems, with dynamic dispatch and other practices imply. Rather, we find using an orthogonal analysis on top of existing models to be a preferable solution. This allows modelers to keep the structure and efficiency of existing designs, while allowing them to also leverage the advantages that come with including domain information into the software itself.

## 2. OUR APPROACH

We leverage the approach of Leung et al. [8] in which a model builder can explicitly specify the properties in which they are

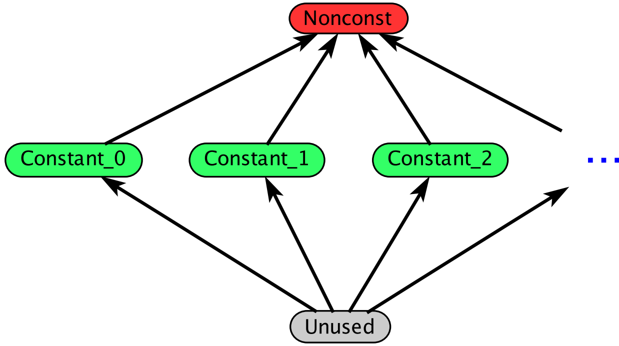



**Figure 1: An infinite flat lattice for doing constant propagation.**

interested by creating a lattice-based ontology. This ontology, along with a few manual annotations within a model-based specification can then be used in an efficient algorithm from Rehof and Mogenson [13], similar to Hindley-Milner type theory, in order to infer properties throughout the model.

We implement our analysis framework on top of Ptolemy II [3], an extensible open source model-based design tool written in Java. While Ptolemy II makes a good testbed for implementing and experimenting with new analyses, we also feel that the techniques we present here are broadly useful. For this reason, we aim to make our analysis framework orthogonal to the execution semantics of Ptolemy II, allowing it to be applied more easily to a broad selection of model-based design tools, such as TDL [12], ForSyDe [14], SPEX [9], ModHel'X [4], and Metropolis [1], as well as commercial tools like LabView and Simulink.

We see the dimension ontology given as an example in [8] to be a step in the right direction for solving unit issues, but ultimately insufficient. By only distinguishing between separate dimensions but not between different units of the same dimension, this dimension ontology is unable to discover the unit errors that lead to the problems presented in Section 1. Unfortunately, this limitation is not simply a case of ontology simplification, but an inherent shortcoming of expressing ontologies as a finite set of discrete concepts. This is because an ontology that expresses units rather than just dimensions fundamentally must represent somehow the scale and offset of separate units within a dimension, which cannot be contained in a simple finite lattice structure. Additionally, there are structured data types which provide useful abstractions for programmers, but whose properties do not fall neatly into the finite lattice restrictions given in [8]

In this work, we present generalizations of these two use-cases into a class of infinite ontology patterns that we have found useful and broadly applicable to semantic property analyses. We first present an overview of the general patterns, and then show their implementation as they apply to the unit system ontology presented here.

# 3. INFINITE ONTOLOGY PATTERNS

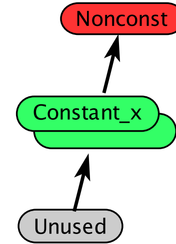



**Figure 2: Using a $FlatTokenInfiniteConcept$ to represent an infinite flat lattice.**

There are two main patterns that we utilize for allowing users to create potentially infinite lattices. The first type expresses an infinite number of incomparable elements that can be inserted into the lattice. This can be used to represent things like flat lattices with an infinite number of incompatible elements.

The other pattern expresses lattices that are self-referential, in which a lattice may recursively contain itself. A simple example of this is the array type of a type system. Since an array may contain elements of any type, including another array, the structure of the array sub-lattice is the same as the overall type lattice, recursively defining an infinite lattice.

## 3.1 Infinite Flat Lattice Pattern

The pattern that we utilize for creating an infinite flat lattice representative is simple. The user can select a special type of concept, called a $FlatTokenInfiniteConcept$ , and use it in her model in the same way she would use normal finite concepts, as seen in Figure 2. The only difference is that here the concept represents a potentially infinite set of concepts of the user's choosing. This pattern allows for a very intuitive approach to representing not only flat lattices, but also more complicated lattices that also contain infinite incomparable subparts.

One nice property of the infinite flat lattice pattern is that it does not increase the height of the lattice. The fixed point algorithm we use from Rehof and Mogenson [13] runs in time proportional to the height of the lattice, without regard to the overall size. This means that infinite flat lattices do not sacrifice inference efficiency in order to achieve their increased expressiveness.

### *3.1.1 Constant Propagation Analysis*

A simple example of an analysis that makes use of this type of lattice is constant propagation. Constant propagation is a static analysis often used in compilers that computes which variables in a program are constant, as well as their values. Usually, a lattice is used that has a separate concept for each constant element type, as well as an additional concept to represent a non-constant type. This produces the infinite flat lattice structure shown in Figure 1, represented in our software with a $FlatTokenInfiniteConcept$ as shown in Figure 2. The way that such a lattice is normally used is as follows: given a simple deterministic operation on two constant values, the constraint can simply perform the operation on the abstract values. Given an operation over a

| Component | Constraint |
|---|---|
| Addition | $\oplus(x,y) = \begin{cases} Unused & \text{if } x = Unused \\ & \text{ or } y = Unused \\ x + y & \text{else if } x < Nonconst \\ & \text{ and } y < Nonconst \\ Nonconst & \text{otherwise.} \end{cases}$ |
| Subtraction | $\ominus(x,y) = \begin{cases} Unused & \text{if } x = Unused \\ & \text{ or } y = Unused \\ x - y & \text{else if } x < Nonconst \\ & \text{ and } y < Nonconst \\ Nonconst & \text{otherwise.} \end{cases}$ |
| Multiplication | $\otimes(x,y) = \begin{cases} Unused & \text{if } x = Unused \\ & \text{ or } y = Unused \\ x \times y & \text{else if } x < Nonconst \\ & \text{ and } y < Nonconst \\ Nonconst & \text{otherwise.} \end{cases}$ |
| Division | $\oslash(x,y) = \begin{cases} Unused & \text{if } x = Unused \\ & \text{ or } y = Unused \\ & \text{ or } y = Constant_0 \\ x/y & \text{else if } x < Nonconst \\ & \text{ and } y < Nonconst \\ Nonconst & \text{otherwise.} \end{cases}$ |

**Table 1: Constraints for the constant propagation example**

non-constant value, however, we simply conclude that the resulting value is non-constant. There may be cases where non-constant inputs still have constant outputs, but this approximation is simple and sound, in that we will never conclude that a non-constant value is constant.

The constraints for the basic binary operations of addition, subtraction, multiplication, and division are given in Table 1, and mirror closely our operational notion of what these operations do (Note the special case for the division operation, since division by zero should be disallowed).

A simplified example of such an analysis is shown in Figure 3. This simple model has two types of source actors at the left: the Const actors each produce a single unchanging output throughout the execution, whereas the Ramp actor produces a time varying sequence. The Ramp actor in this simple model can represent any other non-constant sources that we may have to deal with such as sensors, network packets, or user input. Even in the presence of non-constant sources, however, the lower section of the model deals only with constant values, and the analysis computes, for example, that the output of the MultiplyDivide2 actor will always be the constant value 5600.

Using such an analysis allows model builders to see not only which signals in their models are constant, but also what the values of such constant signals are, which is often just as important. If the model builders were so inclined, they could use this information to simplify the model into a smaller optimized version with the same behavior but no run-time computation of constant values.

## 3.2 Infinite Recursive Type Patterns

The other infinite lattice pattern that we have observed to be useful is that of a self-referential recursive structure. The classic example is an array type, where the array type is not a final type in and of itself, but is parametrized with respect to the type of the elements of the array. In this way, a recursively defined hierarchy of array types can be built up starting with arrays of primitive types and then of arrays of arrays of primitives, and so on. In fact, all structured data types that can include data types inside of them share this property, including lists, records, sets, and more.

In these cases, the lattice that represents all of the possible types becomes not only infinite, but also infinite in height. This means that these cases lose some of the safety that we had with finite-height lattices, but there are benefits from these structured types as well. They allow us to represent a variety of useful patterns that are more varied than one may think at first. In addition, there are heuristics that allow us to deal with many cases decidably. In Section 5 we discuss specifically the design of infinite recursive lattices for supporting records of concepts, and these issues are discussed in more depth there.

# 4. UNIT SYSTEMS

One of the drawbacks of the original dimension analysis presented in [8] was that it could not check for inconsistencies arising from different units of the same dimension, such as having one component expect an input in feet coming from a component producing an output in meters. While it may technically be possible to add concepts and rules corresponding to each of the individual units in use in a particular model, the resulting ontology would be brittle and the resulting rules cumbersome. Using the infinite flat lattice pattern allows us to layer the information about units on top of a dimension lattice without complicating the basic structure. The way we do this is by replacing each individual dimension with a $FlatTokenInfiniteConcept$ that represents the scaling factor and offset of each unit in that dimension with respect to a representative unit. Our unit ontology also contains a $Dimensionless$ concept that is a special finite concept that represents model signals with no physical dimension and thus no units.

There is no limitation on what types of units can be represented in an infinite ontology. Figure 5 shows a lattice that contains dimensions that cover several base SI units for di-

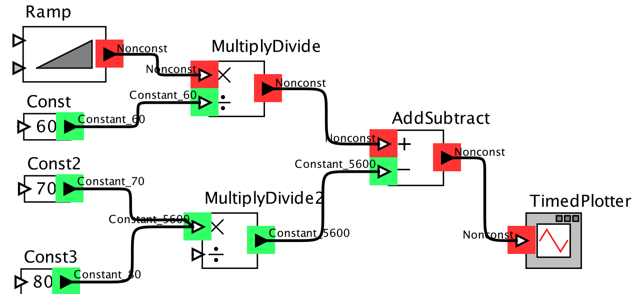



**Figure 3: A model on which constant propagation analysis has been applied.**

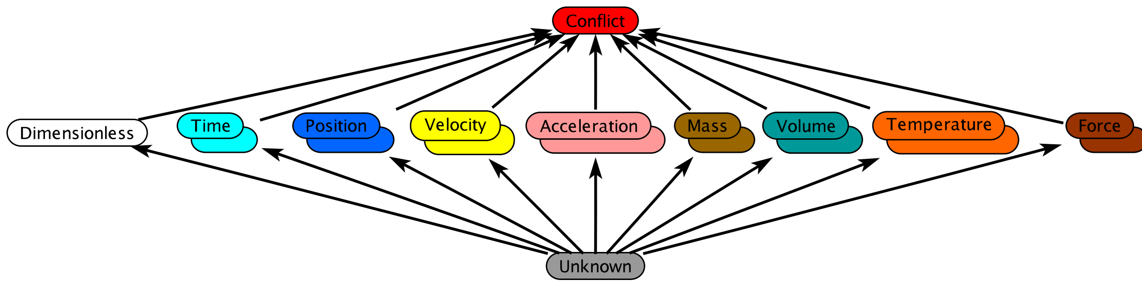



**Figure 5: A generic lattice for unit analysis.**

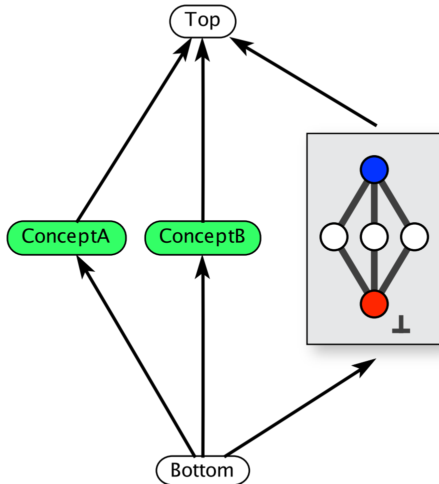



**Figure 4: An infinite recursive lattice can include references to itself.**

| | |
|---|---|
| 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 6: Attributes of the $Time$ base dimension.**

mensions such as Mass, Time, Position, and Temperature, plus a few combined units from the Velocity, Acceleration, Volume, and Force dimensions that are derived from the base units. Note that some of the units here have non-zero offsets, such as Celsius and Fahrenheit temperatures. Despite the difficulties with multiplying and dividing by units with non-zero offsets, there is no problem with expressing them, converting between them, and checking their consistent use.

Before we actually delve into the different units within a dimension, first let us note the approach that we take to distinguishing different dimensions. Like the dimension lattice and unlike most traditional unit systems [6] we explicitly enumerate all of the dimensions that we will be interested in. This means that any single unit ontology cannot hope to be comprehensive, but it also means that we are able to separately represent semantic dimensions that are composed of the same elementary units. This means that we could have an ontology that makes a distinction between distance and altitude, or work and torque, even though the underlying units are the same in both cases.

One of the features of our unit system infrastructure is that users may create arbitrary unit systems that do not necessarily correspond to SI units or any other existing fixed unit system. We allow this through the creation of two categories of dimensions to which units may belong: *base dimensions*, which cannot be broken down into smaller pieces, and *derived dimensions*, which can be expressed as products or quotients of other dimensions.

Base dimensions are the building blocks of our unit systems. As shown in Figure 6, within a given base dimension, all the units are expressed in terms of their scaling factors and offsets with respect to a specific unit, called the *representative unit*. For simplicity, we allow offsets to be omitted when they are zero. For example, if we chose to use $cm$ as our representative unit of position, then we could express the unit of a meter as $100 \times cm$ and of an inch as $2.54 \times cm$. This means that each individual unit is specified as a combination of the dimension to which it belongs as well as the scaling factor and offset from the representative unit of its dimension. As a form of shorthand, we allow the user to specify names for specific scaling factors, such as $cm$, $m$, or $inch$. These names must be qualified by the dimension to which they belong, leading to fully qualified unit names like $Position_cm$ or $Time_s$.

Derived dimensions are specified as a set of base dimensions and their corresponding exponents, as shown in Figure 7. Here, Acceleration is expressed as a derived dimension based on Position and Time, where the exponent of Position is 1 and the exponent of Time is $-2$. The units of derived dimensions are expressed in terms of units of base dimensions.

In this way, we can define as many units as we please within a dimension. It is important to note that the unit factors and offsets are only used for distinguishing units within a dimension, and not for canonicalizing all unit calculations. For example, a model with all units in English units will

| | |
|---|---|
| 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 7: Attributes of the *Acceleration* derived dimension.**

not need to convert any of its calculations to use metric units just because the representative units of the ontology are in metric. The analysis remains orthogonal to the actual execution semantics of the model.

Note that in order for this to work, we make the restriction that all of the units of derived dimensions are expressed in terms of base dimension units with zero offsets. This means that if kelvins are the only unit of temperature with a zero offset, then any derived dimension based on temperature will only be able to use temperatures expressed in kelvins in its computation. This intuitively makes sense, since the result of multiplying or dividing units with non-zero offsets depends on the values of those offsets.

In cases where there are unit mismatches, users can add units converters to their models. These actors leverage the information held in the units ontology in order to calculate the conversion from one unit to another. For this purpose, we have added a new actor for performing unit conversion. It takes two units from the same dimension as parameters, and uses the information in the units ontology to calculate the necessary scaling and offset to convert from the input unit to the output.

Note that other work with similar aims of adding unit information and static checking to programming systems includes packages for Ada [5], SCADE [15], and SystemC [10]. We value the utility in these efforts, but see our approach as fundamentally different. While other tools add explicit notions of units, our approach only adds enough infrastructure for end users to define their own units. This means that our tool allows model builders to create unit systems that are domain specific, or make semantic distinctions between units that must be the same in a general unit system.

## 4.1 Example Model: Adaptive Cruise Control

Here we present an example of a model used in a cyber-physical system, and then examine what types of analyses we may run on this model and how they can aid us in finding errors and better understanding our model. We use an example model that allows simulation of a system of two vehicles connected by a network of unknown reliability, where the following vehicle must use the information received on the network in order to determine a safe speed for itself. While this model clearly contains simplifications of real-world dynamics, we find it complicated enough to highlight real errors that occur in cyber-physical systems and the benefits of our approach.

Our example model, an adaptation of the example from [8],

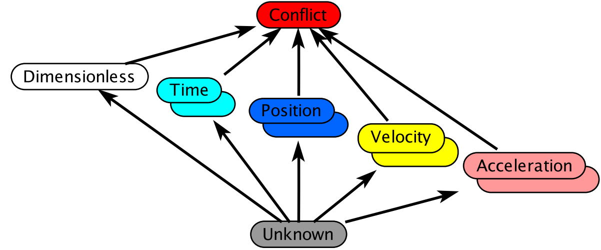


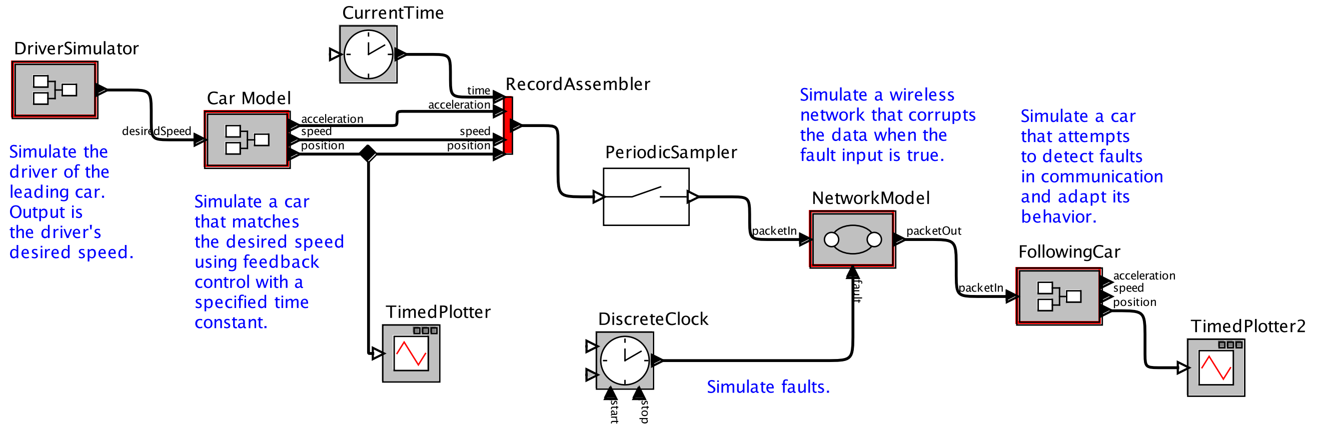

**Figure 8: A lattice for unit analysis of the two-car system.**

is shown in Figure 9(a) at the topmost level of hierarchy. It models a simple two-car system in which the leading car is driven by a human operator and sends its acceleration, speed, and velocity over a wireless link to the following car. The following car then uses the information received over the wireless link in order to determine its own speed, in a system of collaborative cruise control. Since the wireless link is assumed to be unreliable, the following car does sanity checks on the data it receives and falls back to a conservative control algorithm in case the data coming in on the link is deemed unreliable.

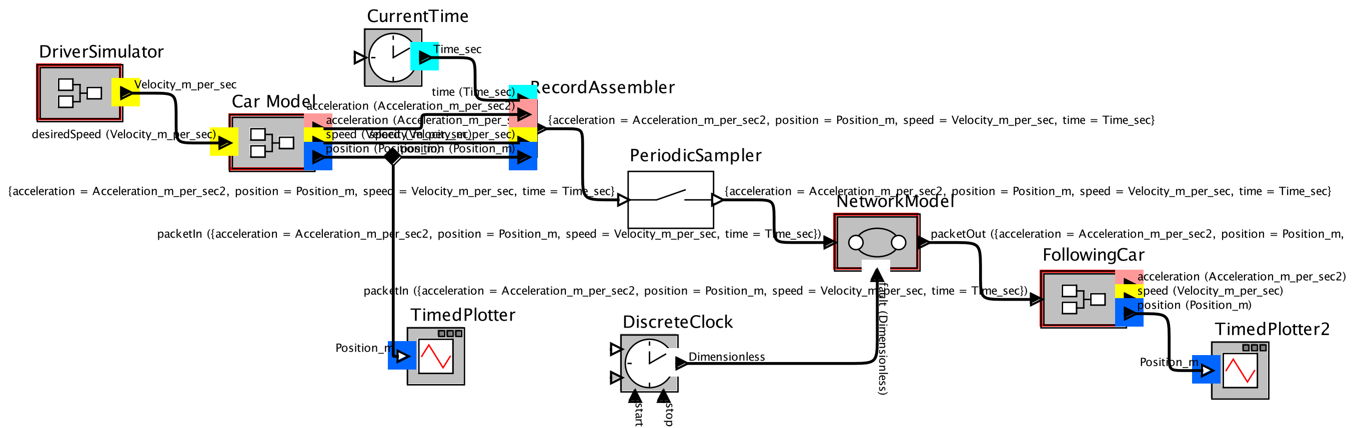
We take as a starting point the simple dimension analysis presented by Leung et al. in [8], but we take issue with the restrictions that they place on their ontologies that we see as impractical. Since their dimension analysis allows only a finite set of dimensions, it is not able to distinguish between units of the same dimension. Unfortunately, this rules out an entire class of error checking, as many common errors are the result of incorrect units within the same dimension.

We present our infinite unit lattice for this adaptive cruise control model in Figure 8. This has the same dimensions as the lattice presented in [8], but instead of each dimension consisting of only a single representative concept, each dimension is a $FlatTokenInfiniteConcept$ which can represent the potentially unbounded different combinations of scaling factors and offsets that different units of a dimension could have.

In order to be able to infer the resulting units throughout a model, it is important to specify constraints on how each actor transforms components. In our experience, many actors in a model will simply produce the same type of units that they accept, so it works best to give reasonable default constraints and then only specify the behavior of actors which differ from this behavior. For defaults, we simply allow the




(a) A model of a two-car system with adaptive cruise control.




(b) Completed unit resolution.

**Figure 9: Unit resolution of the adaptive cruise control example.**

| Component | | Constraint |
|---|---|---|
| Multiplication | $\otimes(x, y) =$ | $\begin{cases} Unknown & \text{if } x = Unknown \text{ or } y = Unknown \\ Position(xScale \times yScale) & \text{if } x = Time(xScale) \text{ and } y = Velocity(yScale) \\ & \text{ or } x = Velocity(xScale) \text{ and } y = Time(yScale) \\ Velocity(xScale \times yScale) & \text{if } x = Time(xScale) \text{ and } y = Acceleration(yScale) \\ & \text{ or } x = Acceleration(xScale) \text{ and } y = Time(yScale) \\ y & \text{if } x = Dimensionless \\ x & \text{if } y = Dimensionless \\ Conflict & \text{otherwise.} \end{cases}$ |
| Division | $\oslash(x, y) =$ | $\begin{cases} Unknown & \text{if } x = Unknown \text{ or } y = Unknown \\ Acceleration(xScale/yScale) & \text{if } x = Velocity(xScale) \text{ and } y = Time(yScale) \\ Velocity(xScale/yScale) & \text{if } x = Position(xScale) \text{ and } y = Time(yScale) \\ Time(xScale/yScale) & \text{if } x = Position(xScale) \text{ and } y = Velocity(yScale) \\ & \text{ or } x = Velocity(xScale) \text{ and } y = Acceleration(yScale) \\ x & \text{if } y = Dimensionless \\ Conflict & \text{otherwise.} \end{cases}$ |

**Table 2: Manual constraints for adaptive cruise control unit system example.**

output of an actor to be the least upper bound of its input constraints, as this allows actors with the same inputs and outputs to be inferred correctly, while also catching and reporting as conflicts cases where incompatible inputs are provided. In our example, the most interesting components that do not fall under the default least upper bound behavior are the division and multiplication actors, whose constraints are given in Table 2. In reality, multiplying or dividing by a unit with a non-zero offset will result in a conflict, since the semantics of such operations are not clearly defined. To simplify the presentation of constraints, however, we ignore offsets and present only behavior when offsets are zero. Other actors can then be derived from multiplication and division. An integrator, for example, has the same effect on units as a multiplication by a unit of time.

Note that while this facility for creating actor constraints is powerful, it is also somewhat cumbersome. Once we define the base and derived dimensions, we may desire that the behavior of a multiplication or division should be determined automatically. In every case we will want multiplying two units together to add the exponents of their component dimensions, and dividing two units to subtract the exponents of their component dimensions.

We have implemented this behavior as the default constraint for the built-in actors for multiplication and division, the MultiplyDivide and Scale actors. Additionally, we have applied the same default behavior to the multiplication and division operators in the Ptolemy expression language, allowing us to infer these same properties across Ptolemy expression actors. This allows us to express the constraints that work for all unit systems once, and then take advantage of them with all subsequent unit systems. The constraints for a general unit system look as follows.

Since we are ignoring offsets, we will represent units as $D(s)$ where $D$ is the dimension and $s$ is the scaling factor. The generic inference constraint for multiplication operations is given as follows:

$$\otimes(x,y) = \begin{cases} Unknown & \text{if } x = Unknown \text{ or } y = Unknown \\ D_z(scale_x \times scale_y) & \text{if } x = D_x(scale_x) \\ & \text{ and } y = D_y(scale_y) \\ & \text{ and } D_z = multiplyDim(D_x, D_y) \\ y & \text{if } x = Dimensionless \\ x & \text{if } y = Dimensionless \\ Conflict & \text{otherwise.} \end{cases}$$

Here $multiplyDim$ is a partial function that finds the new dimension that results from multiplying the two given dimensions. It can perform this calculation by simply adding up the exponents of the arguments of the dimensions passed to it.

The generic inference constraint for division operations is similar:

$$\oslash(x,y) = \begin{cases} Unknown & \text{if } x = Unknown \text{ or } y = Unknown \\ D_z(scale_x / scale_y) & \text{if } x = D_x(scale_x) \\ & \text{ and } y = D_y(scale_y) \\ & \text{ and } D_z = divideDim(D_x, D_y) \\ D_z(1/scale_y) & \text{if } x = Dimensionless \\ & \text{ and } y = D_y(scale_y) \\ & \text{ and } D_z = invertDim(D_y) \\ x & \text{if } y = Dimensionless \\ Conflict & \text{otherwise.} \end{cases}$$

Here $divideDim$ performs the expected analog to $multiplyDim$ in the previous example. Namely it calculates the dimension, if one exists, that results from taking the quotient of the given dimensions. In order to do this, it takes the difference of the exponents of the argument dimensions. The partial function $invertDim$ calculates the dimension with opposite signs for each of the exponents of its argument dimensions.

Note that we allow defining derived dimensions in terms of other derived dimensions, so both $multiplyDim$, $divideDim$, and $invertDim$ all must take this into account in order to calculate the unique set of base dimensions and exponents that make up their arguments.

## 4.2 Example Model: Fuel System

By no means are unit systems only useful for the standard dimensions presented here. In [2], Derler et al. present an example of a fuel system in a aircraft where multiple fuel tanks must orchestrate the movement of fuel throughout the craft while all communication occurs only over a bus with timing delays. A model of the system is shown in Figure 10. Due to the amount of communication happening between the fuel tanks, there are many connections between them. This can be a potential source of transposition errors for model builders, as it is easy to accidentally wire up the actors incorrectly.

While the finite dimension system could only distinguish between fuel levels and flows generally, a full unit system would allow a more exact analysis. In order to do so, we first break the units down into their simplest components: a fuel level is really a representation of volume, and a fuel flow is really a rate of change of volume over time. We chose to measure the tank capacities in liters, and the flows between tanks in liters per second. Building these up from the basic units of length and time gives the complete ontology shown in Figure 11.

Like in the adaptive cruise control example, we will use constraints on how the basic operations of multiplication and division affect our new units. As before, the dimensions will transform according to our intuitive notion of how multiplication and division affect dimensions, while the unit scaling factors will be either multiplied or divided appropriately.

Here, however, we are only interested in derived dimensions. The base dimensions of Time and Length are not important

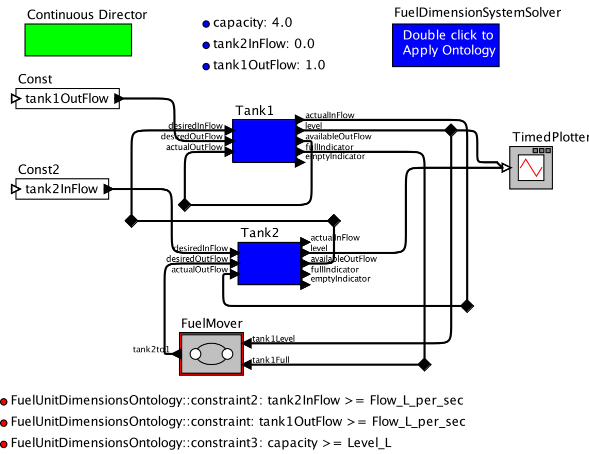



**Figure 10: Model of a two-tank aircraft fuel system.**

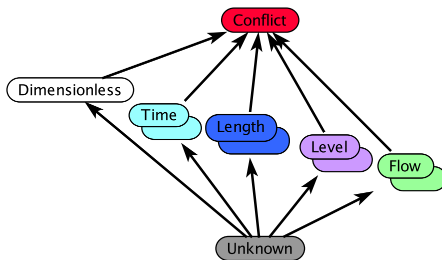



**Figure 11: A lattice for unit analysis of a fuel system.**

in this particular model, as all of the signals in the model measure either a *Level*, *Flow*, or are *Dimensionless*. The *Level* dimension is actually a measure of volume, so we derive this from the *Length* base dimension, and the *Flow* dimension is a rate of change of the *Level* dimension over the *Time* dimension. The completed analysis is shown in Figure 12, with the levels colored purple, the flows colored green, and the dimensionless communications colored white.

## 4.3 Unit Conversions

While the most important step to preventing disasters that result from inconsistent units is to find errors with inconsistent units, there is also utility to correcting those errors to transform erroneous models into correct ones. Because we think that being aware of the units in use is important for designers, we make the deliberate decision not to introduce a feature for unsupervised automatic unit conversion in the case of errors. Instead, we allow the model designer to explicitly add a UnitConverter actor to the model, as shown in Figure 13. This allows conversion from one unit to another within the same dimension, and the UnitConverter can take care of the arithmetic for doing the conversion. It does this by looking up the scaling factor and offsets for the units being converted from the unit ontology. The functionality that the actor then performs on receipt of an input value is to first convert it into the representative unit type and then from the representative unit into the output unit.

One caveat to note is that the UnitsConverter makes the model behavior dependent on the ontology definition, which

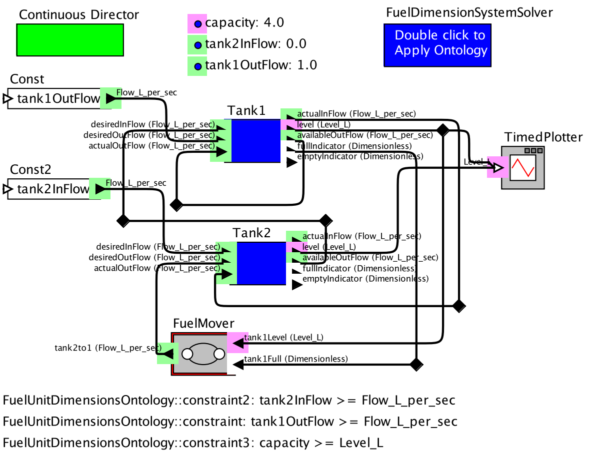



**Figure 12: The result of inferring units over the fuel system model.**

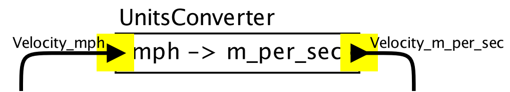



**Figure 13: Using a UnitsConverter to convert from mph to m/s.**

is a unique property of this actor. We think that the benefits and convenience of the UnitsConverter make this worthwhile, but model designers who want to preserve the separation of analysis and behavior can create an equivalent of the UnitsConverter actor by manually computing the conversion between units and specifying the corresponding unit constraints.

## 4.4 Domain specific unit systems

Thus far, all distinct units of measurement, such as those corresponding to SI units, have all had distinct dimension concepts in the ontologies. In some domain specific unit systems, however, a user may want to allow a different set of distinctions. In fact, much of the power of our unit system stems from the fact that we allow distinction between arbitrary concepts. This means users can model distinct concepts from their domain even if they are traditionally considered to have the same units.

Imagine, for example, that the car in our adaptive cruise

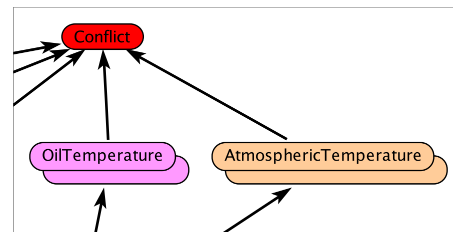



**Figure 14: One way to model two semantically distinct temperatures separately.**

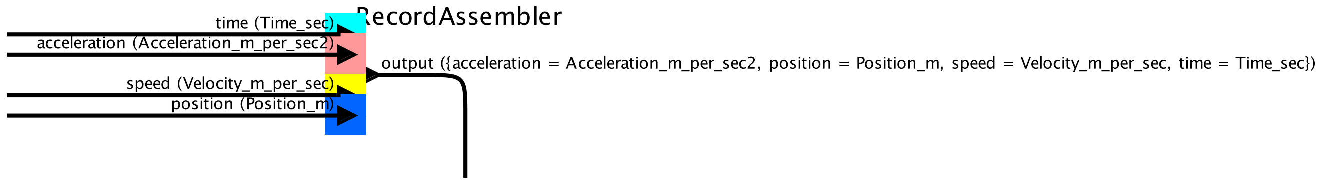



**Figure 15: Unit resolution over the RecordAssembler actor inferring a record of concepts.**

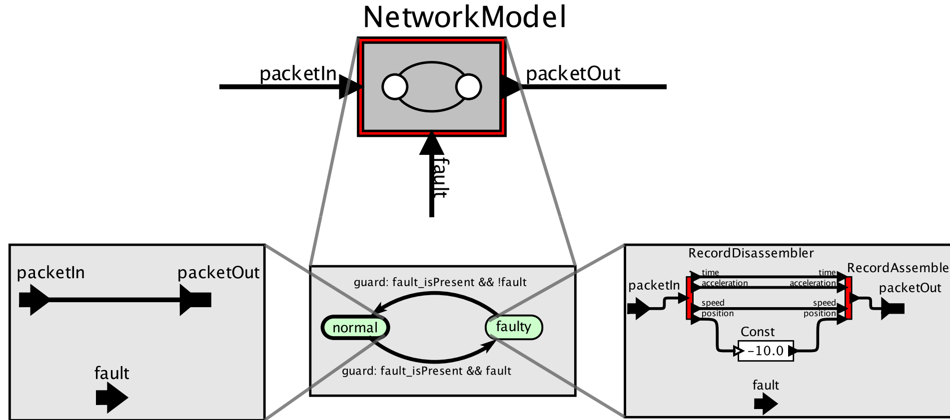



**Figure 16: The interface of the network model becomes much simpler with records.**

control example had sensors for both oil temperature and atmospheric temperature outside the car. Even though both of these sensor readings may be temperature measured in degrees Celsius, they have a very different semantic meaning in the model, and it may be important that these separate semantic meanings are maintained by the unit system. In our approach, a user can specify that these two temperatures have different semantic meaning by simply creating separate dimensions for them within the lattice. In Figure 14 we can see how this would be accomplished. Since the least upper bound of $OilTemperature$ and $AtmosphericTemperature$ concepts in this case is $Conflict$, our default constraints will show that units from these dimensions are incompatible. Using this revised lattice, adding an oil temperature reading to an atmospheric temperature reading would cause a conflict, alerting the user to an error.

We see this type of user-specified semantic distinction as a broadly useful feature. One can imagine aeronautical systems that must keep their notion of distance traveled separate from their notion of altitude, or secure banking systems that must keep the currency units belonging to one customer separate from another. Even units that seem straightforward, such as a joule of work and a newton meter of torque are dimensionally equivalent and must be explicitly distinguished in order to maintain their semantic distinction.

## 5. RECORDS OF CONCEPTS

Another weakness of the example presented in [8] was that it was unable to use the structured data types of Ptolemy to simplify the model. The reason for this is that by restricting their ontologies to be finite, they are unable to express concepts that may contain concepts inside of them. This means they are unable to leverage useful abstraction mechanisms like the Ptolemy II record types.

A record type is a datatype that provides a mapping from strings, called *keys* into values of any type. In Ptolemy, users can create records and break them down into their component parts with the RecordAssembler and RecordDisassmbler actors, respectively. In our example, it would make sense for the data that is sent over the network to be encapsulated into a record rather than modeling each field separately. We can change our model to do so easily in Ptolemy, but doing so exposes a shortcoming in our unit ontology.

Since the output of a RecordAssembler is composed of many separate pieces of data, no one unit type would make sense. It would be possible to add a separate concept specifically for records, but this would make it impossible to get back to the original units used when reversing the process at a RecordDisassembler. What is really needed is a family of records corresponding to all possible combinations of units. Since records may potentially contain themselves (consider, for example, one RecordAssembler whose output is connected to the input of another), this is an instance of an infinite recursive type pattern from Section 3.2.

Since the structure of records is quite common, we provide a general mechanism by which users can add records to any ontologies, and RecordAssemblers and RecordDisassemblers have default constraints that construct and deconstruct these records of concepts in the expected way. Figure 15 illustrates how several input signals that have different

dimensions and units are transformed by a RecordAssembler actor into a record output signal that resolves to a record concept output unit composed of the input units.

Due to the use of record concepts, the packets sent between the vehicles of our adaptive cruise control model can be simplified significantly. The Network simulator, for example, can be simplified as shown in Figure 16. If we chose to model the network differently, with a more abstract behavior, for example, that occasionally dropped packets rather than corrupting them, we could create a network model that was oblivious to the structure of the packets which it carried. This makes models more abstract and reusable, and is an important workflow that we aim to support.

One of the dangers of infinite recursive patterns like those used for record concepts is that they can create infinite height lattices, which can in theory create situations where inference may not terminate. We follow the design of the Ptolemy II type system, which deals with similar problems in supporting structured data types [17]. They deal with this problem by placing limits in certain specific cases on the depth of recursive nesting allowed. Since the run-time semantics of Ptolemy are bound by these restrictions, it makes sense that any static checks, like ours, should reflect the same behavior. The main difference between the record types of Ptolemy and the record concepts in our work is that the type lattice of Ptolemy is fixed and known a priori, allowing specialization for exactly the structured types that Ptolemy supports. We aim for a more general approach that supports records of concepts, but also allows user-created extensions of other similar classes of infinite concepts.

# 6. CONCLUSION

Here we have presented a comprehensive system for allowing ontology-based analysis of actor-oriented models. Unlike previous work, our framework supports useful patterns of infinite ontologies, such as ontologies that contain values and ontologies that contain themselves recursively. One important class of analyses that we have concentrated on is unit systems. Our framework allows user-specified unit systems that include notions of base dimensions and derived dimensions. It specifies reasonable default constraints that models how these units are related, freeing the user from having to specify individual constraints for many common operations.

In contrast to existing unit analysis approaches which conflate the meaning of all quantities using the same unit as being of the same dimension, we allow users to specify dimensions arbitrarily. We see this as useful in cases where there are different domain meanings that happen to be captured with measurements having the same units.

Finally, our infrastructure is general and does not prejudice the specific ontologies or even the types of infinite ontologies presented here. In addition to enabling users to create new ontologies and analyses, we contend that new types of infinite ontologies can and should be added to make analyses more powerful and complete.

# 7. REFERENCES


[1] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *Computer*, 36(4):45 – 52, April 2003.

[2] Patricia Derler, Edward A. Lee, and Alberto L. Sangiovanni-Vincentelli. Addressing modeling challenges in cyber-physical systems. Technical Report UCB/EECS-2011-17, EECS Department, University of California, Berkeley, Mar 2011.

[3] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127 – 144, January 2003.

[4] Cécile Hardebolle and Frédéric Boulanger. ModHel'X: A component-oriented approach to multi-formalism modeling. In Holger Giese, editor, *Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science*, pages 247–258. Springer Berlin / Heidelberg, 2008.

[5] Paul N. Hilfinger. An Ada package for dimensional analysis. *ACM Trans. Program. Lang. Syst.*, 10:189–203, April 1988.

[6] Michael Karr and David B. Loveman, III. Incorporation of units into programming languages. *Commun. ACM*, 21:385–391, May 1978.

[7] Don Lawson. *Engineering disasters : lessons to be learned*, pages 221–229. Professional Engineering Publishing Limited, 1 Birdcage Walk, London, UK, 2005.

[8] Jackie Man-Kit Leung, Thomas Mandl, Edward A. Lee, Elizabeth Latronico, Charles Shelton, Stavros Tripakis, and Ben Lickly. Scalable semantic annotation using lattice-based ontologies. In *12th International Conference on Model Driven Engineering Languages and Systems*, pages 393–407. ACM/IEEE, October 2009.

[9] Yuan Lin, Robert Mullenix, Mark Woh, Scott Mahlke, Trevor Mudge, Alastair Reid, and Krisztián Flautner. SPEX: A programming language for software defined radio. In *In Software Defined Radio Technical Conference and Product Exposition*, pages 13–17, 2006.

[10] T. Maehne and A. Vachoux. Supporting dimensional analysis in SystemC-AMS. In *Behavioral Modeling and Simulation Workshop, 2009. BMAS 2009. IEEE*, pages 108 –113, September 2009.

[11] James Oberg. Why the mars probe went off course. *IEEE Spectr.*, 36:34–39, December 1999.

[12] Wolfgang Pree and Josef Templ. Modeling with the timing definition language (tdl). In Manfred Broy, Ingolf H. Krüger, and Michael Meisinger, editors, *Model-Driven Development of Reliable Automotive Services*, volume 4922 of *Lecture Notes in Computer Science*, pages 133–144. Springer-Verlag, Berlin, Heidelberg, 2008.

[13] Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semilattices. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 285–300, London, UK, 1996. Springer-Verlag.

[14] Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23:1, s. 17-32, 2004.

[15] Rupert Schlick, Wolfgang Herzner, and Thierry Le Sergent. Checking scade models for correct usage of physical units. In Janusz GÃşrski, editor, *Computer Safety, Reliability, and Security*, volume 4166 of *Lecture Notes in Computer Science*, pages 358–371. Springer Berlin / Heidelberg, 2006.

[16] Ajay Tirumala, Tanya Crenshaw, Lui Sha, Girish Baliga, Sumant Kowshik, Craig Robinson, and Weerasak Witthawaskul. Prevention of failures due to assumptions made by software components in real-time systems. *SIGBED Rev.*, 2:36–39, July 2005.

[17] Yang Zhao, Yuhong Xiong, Edward A. Lee, Xiaojun Liu, and Lizhi C. Zhong. The design and application of structured types in Ptolemy II. *Int. J. Intell. Syst.*, 25:118–136, February 2010.