Static Model Analysis with Lattice-based Ontologies

Ben Lickly

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-212
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-212.html

November 20, 2012

Static Model Analysis with Lattice-based Ontologies

by

Ben Lickly

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Edward A. Lee, Chair
Professor George Necula
Associate Professor Alexandre M. Bayen

Fall 2012

Static Model Analysis with Lattice-based Ontologies

Abstract

Static Model Analysis with Lattice-based Ontologies

by

Ben Lickly

Doctor of Philosophy in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Edward A. Lee, Chair

This thesis demonstrates a correct, scalable and automated method to infer semantic concepts using lattice-based ontologies, given relatively few manual annotations. Semantic concepts and their relationships are formalized as a lattice, and relationships within and between program elements are expressed as a set of constraints. Our inference engine automatically infers concepts wherever they are not explicitly specified. Our approach is general, in that our framework is agnostic to the semantic meaning of the ontologies that it uses.

Where practical use-cases and principled theory exist, we provide for the expression of infinite ontologies and ontology compositions. We also show how these features can be used to express of value-parametrized concepts and structured data types. In order to help find the source of errors, we also present a novel approach to debugging by showing simplified errors paths. These are minimal subsets of the constraints that fail to type-check, and are much more useful than previous approaches in finding the cause of program bugs. We also present examples of how this analysis tool can be used to express analyses of abstract interpretation; physical dimensions and units; constant propagation; and checks of the monotonicity of expressions.

Contents

List of Figures

List of Tables

Acknowledgments

Welcome to my acknowledgments. Congratulations on such a thorough reading. If you're reading this page in earnest, I ought to thank you too. Thank you! And while we're at it; Thanks Mom! Thanks Dad! Thanks Shaomei! Thanks Edward!

My work in this thesis grew out of the PtHOMAS project, originally started by Thomas Mandl, Thomas Feng, Man-kit Leung, and Edward Lee. Along with Charles Shelton, Elizabeth Latronico, and Stavros Tripakis, the first paper [26] forms the background for all of the work of this thesis.

I want to thank my collaborators Charles Shelton, Elizabeth Latronico, and Thomas Mandl from Bosch Research, and Dai Bui, Man-kit Leung, and Thomas Feng and Stavros Tripakis. Charles Shelton and Elizabeth Latronico from Bosch research provided tremendous help with infinite lattice, unit systems, and lattice compositions of Chapter 9, as well as the paper [28] upon which that chapter is based.

Dai Bui provided close collaboration during the implementation of algorithm for error minimization of Chapter 8, and professor Koushik Sen provided much useful feedback for this approach.

During my stay at his lab in the summer of 2010, Professor Kwangkeun Yi showed me many novel aspects of static program analysis, including the monotonicity analysis that was the inspiration for Chapter 10. I must thank him for graciously accepting me as part of his lab, and providing me with a fruitful environment for research.

I would like to thank my Qualifying Exam committee, Professors Paul N. Hilfinger, George Necula, Alexandre M. Bayen, and Edward A. Lee, who took the time to view a preliminary version of the main ideas of this work. Their helpful criticism and feedback greatly shaped and improved this thesis.

I would like to thank all of the members of my lab, especially Isaac Liu, Dai Bui, Christos Stergiou, Shanna Forbes, Patricia Derler, Stavros Tripakis, Hiren Patel, Eleftherios Matsikoudis, Thomas Feng, Michael Zimmer, and Chris Shaver, all of whom let me bounce ideas off of them at one point or another and help get me unstuck.

As always, I would like to thank my advisor, Edward A. Lee, for his direction, enthusiasm, advice, and support. I certainly wouldn't be here without him.

Chapter 1

Introduction

The goal of this work is to improve model engineering techniques by providing a framework for semantic annotations. Semantic annotations help in several ways. First, and most importantly, checking consistency across such annotations can help to expose errors early in the design process. Second, having semantic annotations can engage a designer in a deeper understanding of the model. Third, deciding on a set of annotations helps to standardize semantic information across a development team. This can help prevent misunderstandings. Annotations can be provided manually by the designer or inferred algorithmically. A model may have multiple sets of annotations, each specific to a particular use case domain.

These annotations can be applied to a wide variety of problems. A modeler can use semantic annotations to verify that physical units in a model are always used consistently, or run a taint detection check that all inputs from an external source have been sufficiently sanitized. In addition to catching errors, semantic annotations can help modelers learn about their models, by inferring high level details. A modeler can quickly learn which parts of a large product line simulation are configured or unconfigured, or statically infer in a model performing signal processing the values of signals that remain constant.

To illustrate the key idea, consider a simple modeling component, the Integrator. Integrators were the most fundamental components of analog computers, which were prevalent before the widespread availability of digital computers. Today, integrators are still commonly used in software such as Simulink for modeling physical dynamics and control system design. Such a component might be represented graphically as shown in Figure 1.1. The inputs and

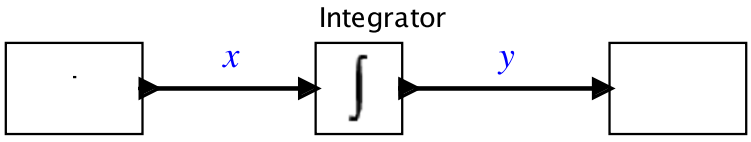


Figure 1.1: An Integrator component reads a continuous value from x and outputs the integral of that over time to y.

outputs of this component are represented as ports, depicted as small black triangles, with the input port pointing in to the component and the output port pointing out. These ports mediate communication between components. Components are composed by interconnecting their ports, and our goal is to ensure that such composition is consistent with the designer's intent.

The Integrator component has some particular properties that constrain its use. First, its input and output ports receive and send continuous-time signals, approximated in a software system by samples. Second, the samples will be represented as an IEEE 754 double precision floating-point number. Third, if the input represents the velocity of a vehicle measured in meters per second, then the output represents the distance the vehicle travels; if the input represents acceleration, then the output represents the vehicle's velocity. Fourth, the physical units measuring the output must be the same as the units measuring the input divided by one second; no mixing of imperial and metric units is allowed. Fifth, the output value may vary over time even if the input does not.

The type system in languages like C or Ptolemy II are sufficient to check for correct usage with respect to the second property, the data type of the ports. This is widely used to check for incompatible connections of components whose types are not compatible. A behavioral type system, such as that implemented in Ptolemy II [25], can check for correct usage with respect to the first property, the structure of the signals communicated between components. Ontology analysis is useful as a configurable and extensible mechanism for performing checks and inference with respect to properties like the third through fifth.

We refer to the third through fifth properties as **semantic types**, and these are typically domain specific. The fact that a model operates on signals representing "velocity" and "acceleration" is a consequence of the application domain for which the model is built. Thus, unlike in type theory, in our case it is essential for the model builders to be able to construct their own domain-specific **ontology**. Here, we present a framework for doing that quickly and easily for a modeler who does not necessarily have extensive knowledge of type theory. This infrastructure is presented as a proof of concept for how this type of analysis can be applied. There is a need for program analyses that are more formal and complete than heuristic based code quality tools, but with less computational overhead than traditionally heavyweight static-analysis and reasoning tools. Built into this infrastructure, we include features to make debugging faulty models easier, to increase the expressiveness and create more powerful analyses, and to automatically check that analyses are built correctly.

We implement our analysis framework on top of Ptolemy II [11], 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 aim to make our analysis framework orthogonal to the execution semantics of Ptolemy II, allowing it to be applied more broadly. It would be straightforward to apply these same techniques to other model-based design tools, such as TDL [37], SysML [49], ForSyDe [43], SPEX [29], ModHel'X [15], and Metropolis [5], as well as commercial tools like LabView and Simulink. Even though Ptolemy II is an actor-oriented modeling tool, many of the techniques can be applied more broadly. Because the analyses presented here make no assumptions about the execution semantics of the models

that they analyze, there is potential to extend and generalize these techniques to functional and object-oriented programs.

Part I

Background

Chapter 2

Ontologies / Knowledge Representation

In Philosophy, Ontology is the study of being, and what it means to exist. [42]

Gruber was the first to define what an ontology means in the context of computer science, where it has taken on a very different meaning. In [14], he defines an ontology as "an explicit specification of a conceptualization." More simply, an ontology in information science refers to an explicit organization of knowledge. One of the driving uses of ontologies in computer science has been in knowledge representation for automated reasoning about vast data sets found on the Internet; the Resource Description Framework (RDF) [31] and later the Web Ontology Language (OWL) [13] were created to describe such a need.

RDF defines a standard data model for representing structured information, which at the lowest level can be represented as a set of statements, each with a subject, a predicate, and an object. For example, an RDF specification could express the concept that "Socrates is a man" by having "Socrates" be the subject, "Man" be the object, and "is a" be the predicate.

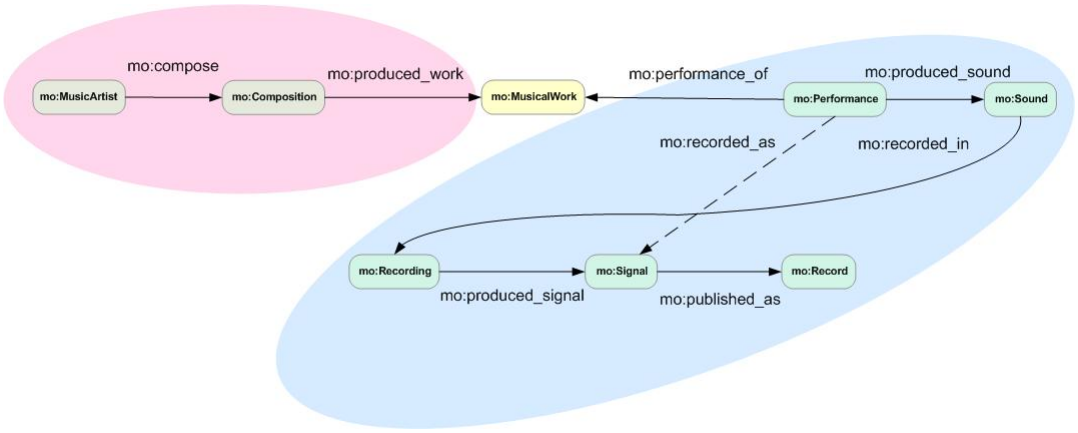


Figure 2.1: An example RDF ontology describing musical works.

One way to visualize RDF information is as a graph, where the subjects and objects become nodes of the graph, and the predicates become directed edges. A non-trivial example is the detailed ontology for the domain of music is available from `http://musicontology.com/`, a small portion of which is reproduced in Figure 2.1. This ontology can be used to describe many semantic aspects of music creation, recording, and reproduction. Here, the musical work in the middle is created by the creative forces on the left (in pink), and captured by the technology on the right (in blue). Although RDF itself refers to the data model itself, the most common format is an XML representation called RDF/XML.

OWL is built on top of an RDF/XML representation for the concepts of an ontology and the relationships between the concepts. Additionally, OWL defines a formal semantics that allows for making and answering queries about ontologies. OWL ontologies can be created in editors such as Protégé [21] from which queries can also be posed. There are multiple dialects of OWL that make different trade-offs. Some dialects make restrictions in order to make it easier to answer queries, whereas other dialects concentrate on being more expressive, with the most expressive dialect, "OWL Full" able to express queries that are undecidable. For the "OWL DL" subset of OWL, closely related to description logic, queries are more tractable, and many reasoners exist, such as FaCT++ [48], Pellet [45], and HermiT [33].

2.1 Ontologies as Documentation

The idea of using ontologies in order to document software is not a new one. Yonggang Zhang [53] [54] presents an approach to using ontologies as a way of comprehending computer programs. This builds on program comprehension languages like UML class diagrams [2], which creates a visual syntax for concisely expressing the class structure of object-oriented software systems. While these types of ontologies are useful for programmers, they tend to discuss only the concepts from a software engineering perspective, making them less flexible and useful for talking about properties of software that are not broadly applicable across software sources.

In [56], Zhou, Chen, and Yang present an approach where they combine an ontology describing software architecture with an ontology of domain concepts into a new ontology. They then propose that this combination ontology can be used to help those with domain knowledge to understand more about the program being documented. The focus of this approach is commendable: bringing in domain information and connecting it to the software is a helpful feature for making software more comprehensible for those with domain understanding, but it is not clear how their domain ontology should be constructed or by whom. We try to address this as well, by presenting a framework in which model builders can quickly create simple domain-specific ontologies. These need only to be powerful enough to analyze the model under consideration, which can make them much simpler than ontologies that aim to be all-encompassing.

The Kepler project [1], a tools for constructing scientific workflows, contains more concrete example of how ontological information can be used to document a program. Here,

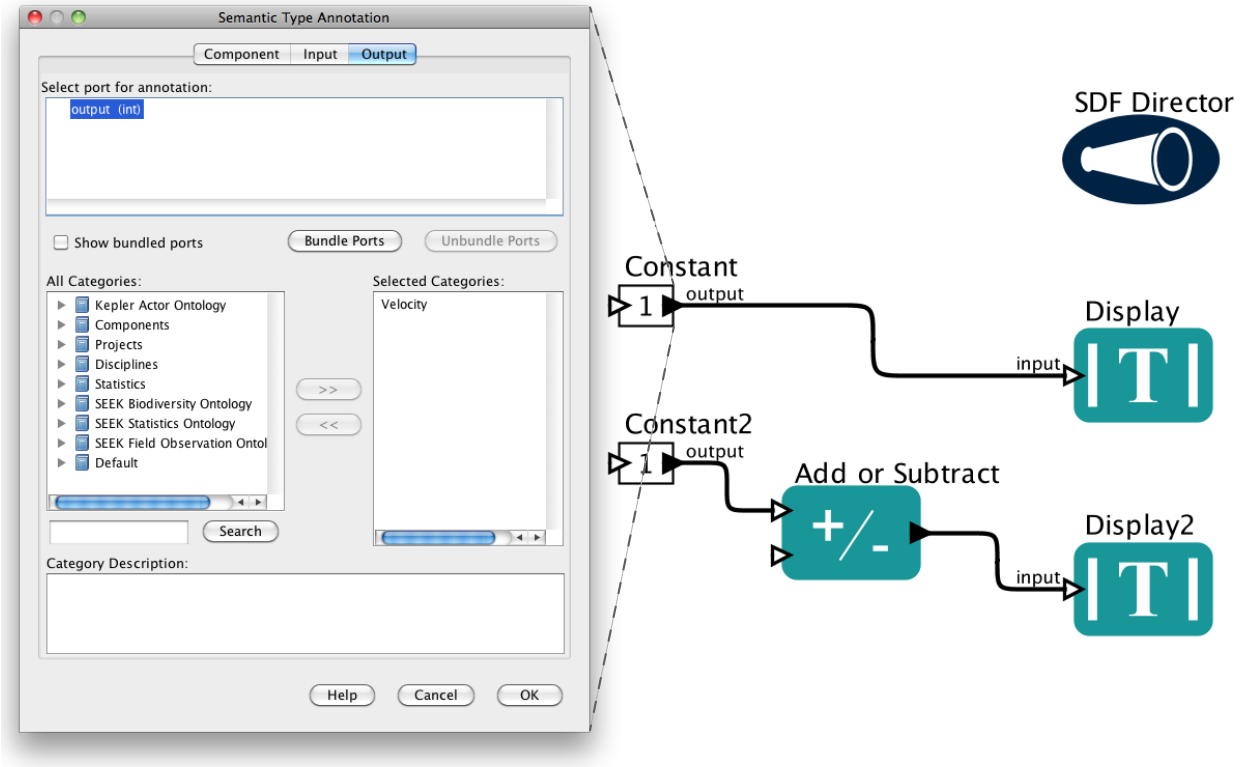


Figure 2.2: An example Kepler model using the Semantic Type annotation package. Any model element, including actor ports, can be annotated with ontological information.

dataflow workflows and their components can be tagged with *semantic type annotations* from an ontology [38]. OWL ontologies can be imported, and then concepts from these ontologies can be used to tag ports and actors of a workflow model. Additionally, actors that are linked together can have those compositions checked automatically with a semantic type checker.

Figure 2.2 shows an example of a model annotated with semantic types. Using the dialog shown on the left, any element of a Kepler model can be annotated with concepts from an ontology, including ports of actors. Later, ports connected together can be checked for compatibility. Though not visible from the dataflow diagram, in this example *Constant* and *Constant*2 have been annotated with semantic type annotations on their outputs and *Display* and *Display*2 have been annotated on their inputs. If the type of the upstream actor's output is a subtype of the type of the downstream actor's input, then this composition is valid. This case would be guaranteeing that the output produced would always be a subset of the valid inputs that could be consumed. Figure 2.2, however, shows an invalid composition, in which the *Constant* actors produce a type that is inconsistent with what the *Display* actors consume. Thus, this connection should result in an error. Figure 2.3 shows the summary of the type checker for this model. Here, the checker is able to determine an error on the connection between *Constant* and *Display* since they are directly connected to

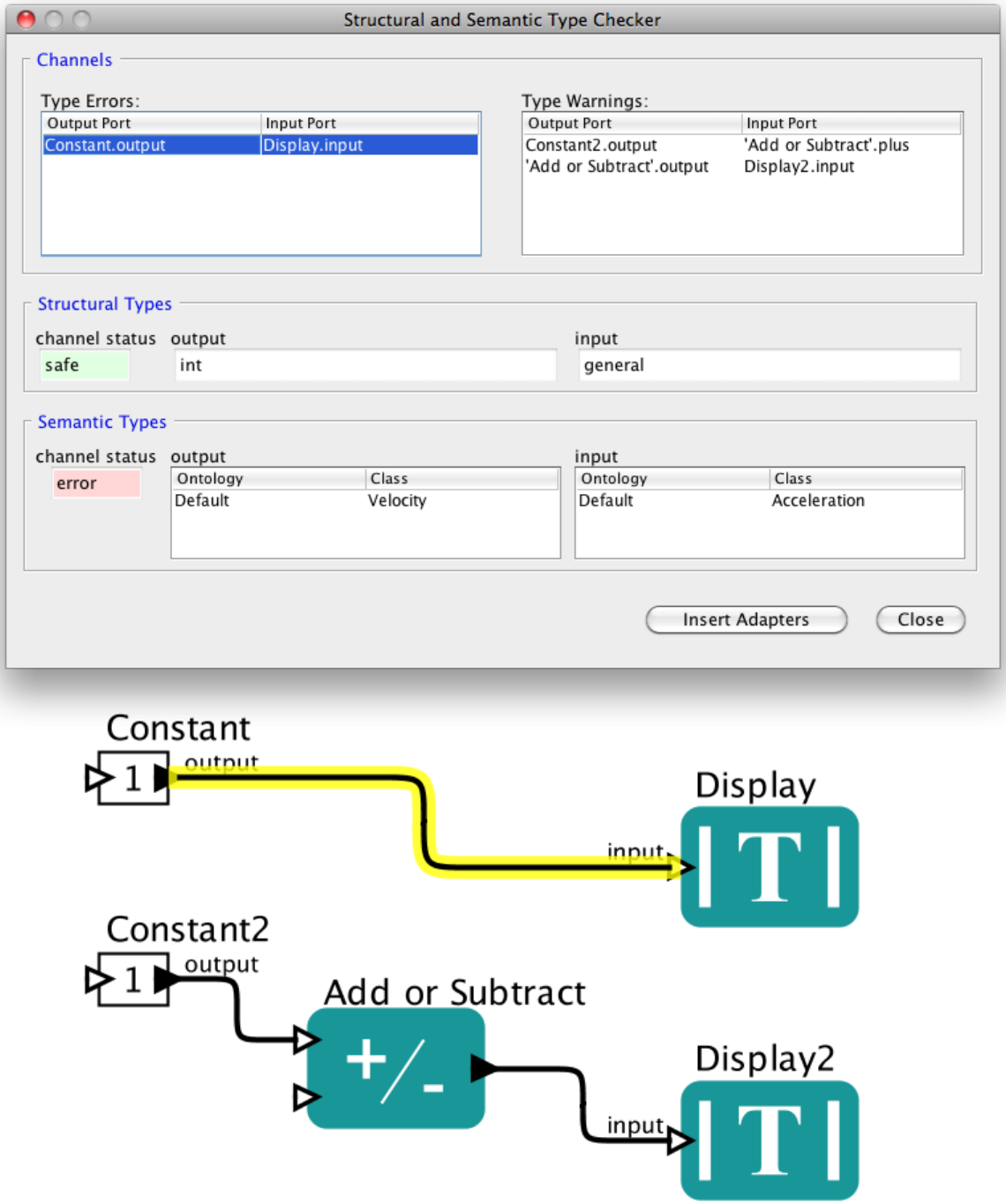


Figure 2.3: Kepler can find and report semantic type errors on adjacent ports.

each other.

While this is sufficient to check the annotations on two components connected together directly, this means that every component in a model must be annotated in order to get complete checks. Since the semantic types do not include a method for type inference, the process of manually tagging large models is cumbersome and error-prone. This means that in practice there is a small practical limit on the size of models that can be annotated with this approach. In this thesis, we address this issue by presenting a customizable method for specifying inference of ontology concepts, allowing builders to specify ontologies as well as how those ontological concepts should be inferred in their models. This allows builders to make fewer manual annotations but still be able to completely annotate large real-world models.

Chapter 3

Static Analysis

Static program analysis is the process of deriving meaningful properties of computer programs without running them. This is often necessary because testing a program exhaustively is impossible or infeasible. Static program analysis is most often used in embedded and safety-critical systems, where ensuring program correctness before it is deployed is the most important, but static program analysis can also be deployed to verify the security, correctness, or coding style of a program.

3.1 Heuristics-based tools

One usage of static program analysis is as a way for programmers to learn more about the programs that they work on. In this type of static analysis, rather than focusing proving program properties, the analysis merely reports useful information to the programmer. The programmer can then use this information to make decisions about the code. Some of the oldest such analyses can be found in lint [17], a program that checks C source files for error-prone or non-portable constructs.

3.2 Heavyweight Tools

Another branch of static analysis deals with more heavyweight approaches. The most famous of these is solving the Boolean satisfiability problem (SAT), which asks whether a Boolean formula with conjunctions, disjunctions, and negations is satisfiable. This problem was proved to be NP-complete, which would suggest that it is likely that arbitrary instances of the problem can not be solved efficiently. Interestingly, many advances have been made in recent years at solving these problems more efficiently for certain cases, and SAT solvers are used as a basis for many other types of analyses. These types of approaches are useful, but often their applicability is limited by their relatively high computational costs. This means that they are more useful for performing final validations than as part of a process where fast feedback is required.

3.3 Sound and Efficient Analyses

Type Checking and Inference

Probably the most widely used type of static analysis is the type-checking of statically typed programming languages. These are commonly used in programming languages in order to prevent certain types of errors. Statically typed programming languages are so widely used that they can take on a broad spectrum of forms. These can range from relatively simple traditional C-style types that dictate how bits in memory should be viewed, to complex dependent type systems that can prove interesting program properties such as memory safety [7] and even program termination [51].

Compiler Optimizations (control-flow and data-flow analysis)

In addition to checking the correctness of programs with respect to their type systems, compilers also often perform analyses in the process of automatically optimizing programs [34]. These include things like determining whether discrete pointers may alias to the same object, or analyzing the data-flow structure of variables to propagate constant values throughout code and determine regions of code to be unreachable. Additionally, it has been noted that these types of compiler optimization problems can be formulated generally as fixed-point problems over semi-lattices [20], in a similar way to the formalization of the analyses in this work.

Abstract Interpretation

Abstract interpretation [8] is a technique that statically analyzes programs by abstracting the possible values that program objects can take as sets. These sets contain may contain a superset of the real possible values, but by performing the conservative approximations, many computations are possible that would not be decidable for concrete values. By choosing these sets correctly, sound scalable abstract interpretation analyses can prove useful program properties. The ideas of this work draw heavily on abstract interpretation, but here we do not require our analyses to consider only abstractions of the values computed. Our analyses may also consider information that is completely orthogonal to the values computed by the program when it is run. This gives our analysis the ability to be used as documentation of aspects outside the program semantics, in addition to the possible use for traditional static analysis.

Chapter 4

Interface Disasters

In building embedded software for cyber-physical systems, there are a host of potential problems when interfacing components with one another. One of the leading causes of failure is mismatched assumptions between software components [47]. This occurs when the semantics of what one component expects to receive does not match the semantics of what another component produces.

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 [22], and the Mars Climate Orbiter, which crashed into the planet on descent due to a unit error between Newtons and pound-forces [36].

Another type of semantic error can occur when components are reused without checking that all of their requirements are met. One example of this occurred on the first test flight of the Ariane 5 rocket system [23]. Code reused from the Ariane 4 rocket assumed that the horizontal bias of the vehicle could be fit into a 16-bit signed integer, which was not protected for efficiency reasons. The Ariane 5 had a much higher horizontal velocity, and used a 64-bit floating-point number to represent the horizontal bias. This fundamental disconnect in the meaning and range of "horizontal bias" was not communicated effectively during design or testing, and the rocket disintegrated 39 seconds after takeoff.

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 examples such 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 other semantic property known to the model builder.

While traditional software projects often encode domain information into their object-oriented type hierarchy, executable actor models tend to be structured around scheduling and concurrency decisions rather than semantic domain information. 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. A better approach is to allow for annotations that are orthogonal to the execution semantics of the system, such as the annotations for non-functional properties provided in the UML Profile MARTE [12]. 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.

Chapter 5

Mathematical Background

5.1 Order Theory and Lattices

Partial Orders

A **partial order** is just an ordering, like $\leq$, in which not all elements are necessarily ordered. Some examples of partial orders are the subset operation on sets ($A \leq B$ if A is a subset of B), and the prefix operation on strings ($s_1 \leq s_2$ if s_1 is a prefix of s_2).

More formally, a partial order consists of a binary relation ($\leq$) defined on a given set P, such that $\leq$ is:

- Reflexive (i.e. $x \leq x$)
- Transitive (if $x \leq y$ and $y \leq z$, then $x \leq z$), and
- Anti-symmetric (if $x \leq y$ and $y \leq x$, then x and y must refer to the same element).

We can also define the familiar operators of $=$, $<$, $\geq$, $>$, and $\neq$ in terms of $\leq$. For example, $\neq$ can be defined as $x \neq y$ if and only if $x \not\leq y$ or $y \not\leq x$.

Within a partial order P, an **upper bound** of some set of elements S is an element that is greater than or equal to every element in the set. Formally, $x \in P$ is the upper bound of S if

$$\forall s \in S, s \leq x.$$

The **least upper bound** of a set of elements S in a partial order P is the unique element in P, if it exists, that is less than or equal to all other upper bounds of S. The least upper bound is often called the **supremum** of the set, or the **join** of the set, and is written $\bigvee S$. The **lower bound** of a set of elements is closely related, referring to an element less than or equal to every element in the set. In a partial order P, $x \in P$ is the lower bound of a set S if

$$\forall s \in S, x \leq s.$$

If it exists, the **greatest lower bound** of a set of elements S is the lower bound greater than or equal to all other lower bounds. The greatest lower bound is often called the **infimum** of the set, or the **meet** of the set, and is written $\bigwedge S$. Least upper bounds and greatest lower bounds can also be defined between pairs of elements and written as binary operators. Given a pair of elements x and y, we write their join and meet as:

$$x \vee y = \bigvee \{x, y\}$$
$$x \wedge y = \bigwedge \{x, y\}.$$

A **directed set** is a non-empty set of elements S in a partial order for which every pair of elements has a least upper bound in S. Formally, a set S is directed if

$$\forall x, y \in S, x \vee y \in S.$$

Complete Partial Orders and Complete Lattices

Lattices are partial orders that have nice properties that make them particularly useful for representing information in static analyses and type theory. A lattice is a partial order that has one additional restriction. A **lattice** is a partial order for which every pair of elements has both a unique least upper bound (called the **join** and written $x \vee y$) and a greatest lower bound (called the **meet** and written $x \wedge y$).

This means that, for example, the partial order shown in Figure 5.1a is a lattice, while the partial order shown in Figure 5.1b is not. In regard to our examples, the subset relation creates a lattice, because the least upper bound can be found with union, and the greatest lower bound can be found with intersection. The prefix ordering on strings, however, is not a lattice. This is because the least upper bound does not always exist between strings. For example, given two strings for which neither is a prefix of the other, it is impossible to construct a new string of which they are both a prefix.

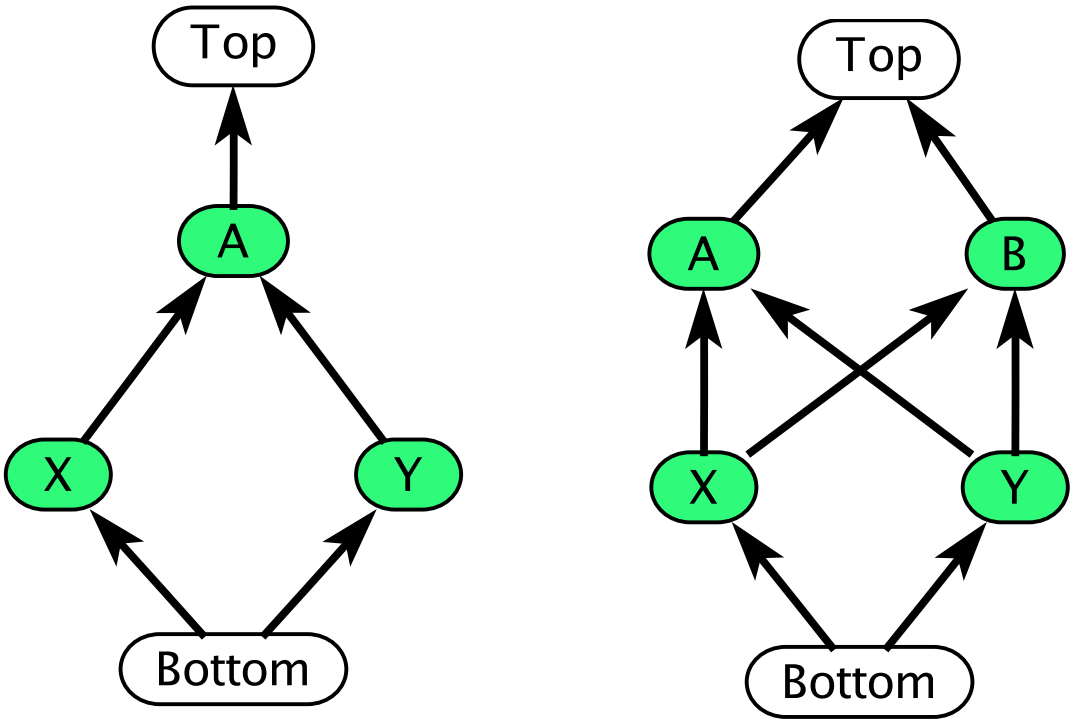
A **complete lattice** further requires that every subset of P have a join and a meet in P. Every complete lattice has a top element and a bottom element, which can be trivially found by taking the join or meet of the entire set. The top element is typically written as $\top$ and the bottom element $\bot$.

Complete Partial Orders

A **complete partial order** or *CPO*, is another type of partial order that is useful for representing program semantics. Every complete lattice is also a CPO, so properties of CPOs will also be useful when using complete lattices.

Formally, a complete partial order P is a partial order with the following two properties.

1. P has a bottommost element, $\bot$.
2. Every subset of P that is directed has its upper bound in P.



(a) A partial order that is a lattice. (b) A partial order that is not a lattice.

Figure 5.1: Example Partial Orders

Cartesian Products

The **Cartesian product** of two sets is the set of all pairs of elements from those sets. For example, the Cartesian product of the sets $Rank = \{A, K, Q, J\}$ and $Suit = \{Heart, Diamond\}$ would be

$$Rank \times Suit = \{(A, Heart), (A, Diamond), (K, Heart), (K, Diamond), \\ (Q, Heart), (Q, Diamond), (J, Heart), (J, Diamond)\}.$$

A **projection** refers to the process of extracting one of the elements of the pair in a Cartesian product. Given an element of a Cartesian product, call it (a, b), we say that it's left projection is a and right projection is b.

For sets that form partial orders, there is a natural ordering on their Cartesian product called the **product order**. For two elements of the Cartesian product $A \times B$, (a_1, b_1) and (a_2, b_2), we say that $(a_1, b_1) \leq (a_2, b_2)$, if and only if $a_1 \leq a_2$ and $b_1 \leq b_2$

The product order preserves the ordering of the elements of A given a fixed B, and vice versa. Additionally, it preserves the properties of CPOs and lattices, so that given A and B that are lattices, it is guaranteed that $A \times B$ ordered by the product order will also be a lattice.

5.2 Monotonic Functions

A **monotonic function** is a function that preserves the order of its inputs. In mathematics over real valued numbers, this refers to a function that never has a negative slope, such as: $f(x) = 3x - 4$.

In general, monotonic functions can be defined over arbitrary partial orders. Formally, given partial orders X and Y, a function $f : X \to Y$ is called monotonic if $\forall x, y \in X$:

$$x \leq y \implies f(x) \leq f(y).$$

Monotonic functions are very useful, since they always preserve the order of their inputs. This property allows one to prove certain properties are preserved when applying monotonic functions. When a monotonic function is applied to each element in an increasing sequence, for example, the resulting sequence is also increasing.

5.3 Fixed points

The fixed point of a function is a value for which the output is the same as the input. For example, $f(x) = x^2$ defined over the real numbers has fixed points at 0 and 1, whereas $g(x) = x + 1$ has no such fixed point.

For functions defined over partial orders, it makes sense to refer to a **least fixed point** or **greatest fixed point**. A least fixed point of a function, if it exists, is simply the unique fixed point that is less than or equal to all other fixed points. Functions which have no fixed points or which have multiple incomparable fixed points have no least fixed point.

Kleene Fixed Point Theorem

The Kleene fixed-point theorem specifies circumstances under which the existence of a least fixed point is guaranteed, and also provides a constructive method to find that fixed point.

Given a monotonic function $f : P \to P$, defined over a CPO P, f will always have a least fixed point. Moreover, this fixed point is equal to the upper bound of the set of repeated applications of f to $\bot$.

Formally, the least fixed point of f is equal to $\bigvee \{x_i\}$ where x_i is defined as

$$x_i = \begin{cases} \bot, \text{ if } i = 0 \\ f(x_{i-1}), \text{ otherwise.} \end{cases}$$

Part II

Basic Ontology Analysis

Chapter 6

Elements of an Ontology Analysis

6.1 Lattice-based Ontologies

In a traditional type system, data types can be thought of as being elements of a complete lattice. A simple example of a type system lattice is illustrated in Figure 6.1. Each node represents a data type, and the arrows between them represent an ordering relation. This relation can be interpreted as an "is a" relation or as a "lossless convertibility" relation. For example, a value of the *Int* type can be converted losslessly to a *Long* or a *Double*, but a *Long* value cannot be converted to a *Double* nor vice versa.

In this work, ontologies are represented as a complete lattice structure. For this reason, we also refer to them as **concept lattice**. The concepts of the ontology are represented as the elements of the lattice, and the ordering relationship is an "is-a" relationship. For concepts x and y, $x \leq y$ in the lattice order means that an x "is a" y. Another way to think about this is that the $\leq$ relation defines an order on the number of different ways an element of the model can be used. This is because an element that is used in fewer different ways can have a more specific concept, and one that is used in more different ways must have a more general one.

For example, the lattice shown in Figure 6.2 represents an ontology for checking which signals have fixed constant values as opposed to time-varying values. In this ontology, the concept *Nonconst* represents model elements that cannot be proven to be constant. In reality, some of these elements could be constant. It is simply the most general type of concept. For some, a clearer name for this concept might be *General*, since it can potentially refer to elements that are either constant or non-constant. In this ontology, a *Const* model element is a particular type of *Nonconst* element that is known to never have more than one value. An *Unused* element is a special type of *Const* element that is known to never be used.

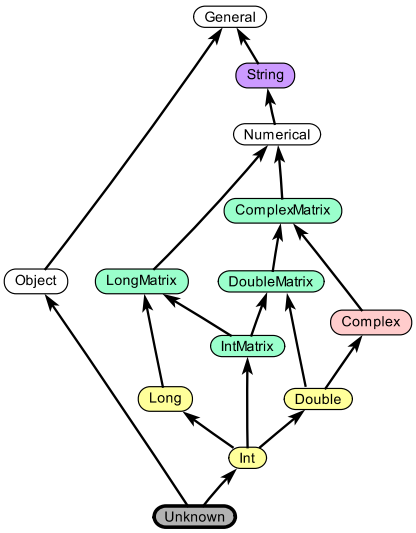


Figure 6.1: A type lattice modeling a simplified version of the Ptolemy II type system.

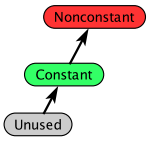


Figure 6.2: A concept lattice modeling signal dynamics.

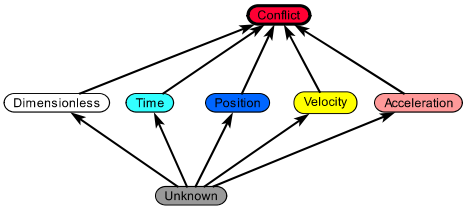


Figure 6.3: A lattice ontology for dimensions Time, Position, Velocity, Acceleration, and Dimensionless.

6.2 Ontology Analysis

In order to make an ontology useful for analyzing a given model, we must make a connection between the ontology and the model in question. One way to connect concepts from an ontology to a model would be to simply exhaustively annotate the model, but this process would be cumbersome and error-prone. A better way is to specify *how* the ontology relates to the model, and then run an algorithm that can **infer** the concepts of objects throughout the model. In order to propagate information throughout a model or program, it is important to understand how the elements of the program interact with the concepts of the ontology. In order to do this, we let each component of the model specify a set of **constraints**.

An **ontology analysis** consists of a concept lattice along with these constraints. When solving these constraints, we would like to make as specific assertions as possible. In particular, if there are multiple possible solutions to the constraints, we would prefer solutions that resolve model elements to concepts that are lower in the ontology lattice. This corresponds to the **least solution**, if one exists. After running an inference algorithm, the end result of an ontology analysis will be a mapping of elements of the model to the domain given by the ontology. In addition to its graphical actor-oriented modeling paradigm, Ptolemy II also includes a functional expression language [24]. Although they cannot be labeled and colored, abstract syntax tree nodes of the Ptolemy expression language can also be resolved to concepts from an ontology. Thus we will use the phrase **modeling element** to refer to anything that can resolved to a concept from a concept lattice, including ports of actors, parameters of models, and abstract-syntax tree (AST) nodes of expressions.

The type lattice of Figure 6.1 is an example of a concept lattice, as are Figures 6.2

and 6.3. We will first explain what an ontology analysis is informally through an example. In particular, we will illustrate how to use the *dimension* concept lattice (Figure 6.3) to check for correct usage of an Integrator component as discussed previously.

Consider a very simple model with three components as shown in Figure 6.4(a). Component C provides samples of a continuous-time signal to the integrator, which performs numerical integration and provides samples of a continuous signal to component B. Suppose that we associate the input x of the Integrator with a concept p_x in the concept lattice L. We wish to catch errors, where, for example, component C sends position information to the Integrator, and component B expects velocity information. This is incorrect because position is the integral of velocity, not the other way around. We can construct an ontology analysis that systematically identifies such errors.

The concept lattice for this ontology analysis is shown in Figure 6.3. To complete the ontology analysis, we need to encode the constraints imposed by the integrator. To do this, we leverage the mathematical properties of a complete lattice.

Suppose we have a model that has n model elements with concepts. In Figure 6.4(a), we have two such elements, x and y, and their concepts are $(p_x, p_y) \in L^2$, where L is the concept lattice of Figure 6.3. An ontology analysis for this model defines a monotonic function $F\colon L^2 \to L^2$ mapping pairs of concepts to pairs of concepts. Intuitively, F takes a set of known concepts and takes a step of inference. The restriction that F is monotonic ensures that we can never retrace our steps in the inference process. A **fixed point** of such a function is a pair (p_1, p_2) where $(p_1, p_2) = F(p_1, p_2)$. The Kleene fixed-point theorem tells us that any monotonic function over a finite lattice has a unique least fixed point defined as follows, for some natural number n:

$$(p_1, p_2) = F^n(\bot, \bot) \ . \tag{6.1}$$

This simply means that iteratively applying the function F until no further concepts can be inferred is a terminating constructive algorithm. We define the **inferred concepts** of a model to be this least fixed point. Note that this least fixed point is not a single concept from the lattice but a set of concept values. This set of concepts can be ordered with the product order to give a coherent partial ordering. The least fixed point associates with each model element a concept in the lattice, which is the inferred concept for that model element. Overall, this process allows us to start with a small set of concepts, and either infer a consistent set of concepts throughout the model, or find that the model is illegal.

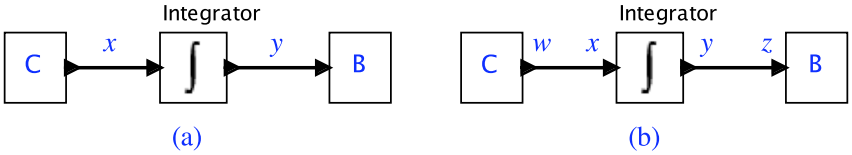


Figure 6.4: Models using an Integrator, where (a) labels connections and (b) labels ports.

Even for the simple Integrator example above, manually defining the function F for the entire example is somewhat complicated and error prone. Later, we will show that there is an alternate way to define it implicitly in an elegant and modular way. To reflect the constraints of the integrator, the function is

$$F(p_x, p_y) = \begin{cases} (\top, \top) & \text{if } p_x = \top \text{ or } p_y = \top \\ (p_x \vee A, p_y \vee V) & \text{else if } p_x = A \text{ or } p_y = V \\ (p_x \vee V, p_y \vee P) & \text{else if } p_x = V \text{ or } p_y = P \\ (p_x \vee D, p_y \vee T) & \text{else if } p_x = D \text{ or } p_y = T \\ (p_x \vee p_y, p_x \vee p_y) & \text{otherwise} \end{cases} \tag{6.2}$$

where $\bot, T, P, V, A, D$, and $\top$ are the elements of the lattice in Figure 6.3: Unknown is $\bot$, Error is $\top$, Time is T, and so on.

The least fixed point of this function is $(p_x, p_y) = (\bot, \bot)$, found after a single application of F in (6.1), meaning we do not have enough information to draw conclusions about the concepts associated with x and y.

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

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

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

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

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

which has least fixed point $(p_x, p_y) = (\top, \top)$, meaning that we have created a modeling error. In practice, model builders should not directly give the function F. This is because it can be inferred from constraints on the components.

We are now closer to formally defining an ontology analysis. An **ontology analysis** for n modeling elements consists of a concept lattice P, a monotonic function $F\colon P^n \to P^n$, and optionally a collection of **acceptance criteria** that define whether the least fixed point yields an acceptable set of concepts. We next show how the monotonic function F can be implicitly defined in a modular way by giving constraints associated with the components. We also explain the need for acceptance criteria and how they are used.

6.3 Constraints and Acceptance Criteria

Rehof and Mogensen [39] give a modular and compositional way to implicitly define a class of monotonic functions F on a lattice by a collection of inequality constraints. They also provide an efficient algorithm for finding the least fixed point of this function. The algorithm has been shown to be scalable to very large number of constraints, and is widely used in type systems, including that of Ptolemy II, which we leverage. Specifically, for a fixed concept lattice L, this algorithm has a computational upper bound that scales linearly with the number of inequality constraints, which is proportional to the number of model components, or the model size.

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

$$p_x \leq p_y \ . \tag{6.3}$$

Such an inequality constrains the concept value of y to be higher than or equal to the concept value of x, according to the ordering in the lattice. Of course, two inequality constraints can be combined to form an **equality constraint**,

$$p_x \leq p_y \text{ and } p_y \leq p_x \Rightarrow p_x = p_y \tag{6.4}$$

because the order relation is antisymmetric.

In Figure 6.4(a), we implicitly assumed an equality constraint for the output of C and the input of the Integrator. We could equally well have assumed that each port was a distinct model element, as shown in Figure 6.4(b), and imposed inequality constraints $p_w \leq p_x$ and $p_x \leq p_w$. However, if our analysis requires it, we can also specify constraints in only a single direction, such as $p_x \leq p_w$ without $p_w \leq p_x$. These are used when defining single-directional analyses, which are often useful and fully supported.

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

$$f(p_1, \ldots, p_n) \leq p_x \tag{6.5}$$

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

$$f_s(p_1, \ldots, p_n) = V$$

where V represents Velocity. Hence, to express that component B in Figure 6.4(b) assumes its input is Velocity, we simply assert the constraint

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

which of course just means

$$V \leq p_z \ . \tag{6.6}$$

However, by itself, this constraint does not quite assert that the only valid value of p_z is V, since it only encapsulates an inequality in a single direction and does not preclude the possibility that $p_z = \top$. In this analysis, it is a modeling error for a signal to resolve to $\top$. Expressing that directly, however, cannot be accomplished through monotonic function constraints as shown in Equation 6.5. We would like to assert that $p_z < \top$, but this type of inequality is not allowed in Rehof and Mogensen's framework. Hence, to complete the specification, we can specify **acceptance criteria** of the form

$$p_i < l \tag{6.7}$$

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

$$p_z < \top \ , \tag{6.8}$$

which when combined with (6.6), means $p_z = V$, or z is Velocity. We can also declare an acceptance criterion that for each model element i with concept p_i,

$$p_i < \top \ , \tag{6.9}$$

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

An arbitrary collection of monotonic function constraints, say $\{f_i(\ldots) \leq p_i\}$, defines a monotonic function $F\colon L^n \to L^n$ as follows:

$$F(p_1, \ldots, p_n) = (p_1 \vee f_1(\ldots), p_2 \vee f_2(\ldots), \ldots, p_n \vee f_n(\ldots)) \ . \tag{6.10}$$

A fixed point of this monotonic function gives assignments to $(p_1, \ldots, p_n)$ that satisfy all of the inequality constraints.

Acceptance criteria do not become part of the definition of the monotonic function F, and hence have no effect on the determination of the least fixed point. Once the least fixed point is found, the acceptance criteria are checked. If any one of them is violated, then we can conclude that there is no fixed point that satisfies all the constraints and acceptance criteria. We declare this situation to be a modeling error. Some analyses are used only as a form of documentation and not as a way of uncovering modeling errors, and these analyses need not use acceptance criteria. Acceptance criteria are only necessary for models that are capable of rejecting certain models as erroneous.

Constraints of the Integrator include one given in the form of (6.5) as

$$f_I(p_y) \leq p_x \quad \text{where} \quad f_I(p_y) = \begin{cases} \bot & \text{if } p_y = \bot \\ V & \text{if } p_y = P \\ A & \text{if } p_y = V \\ D & \text{if } p_y = T \\ \top & \text{otherwise} \end{cases} \tag{6.11}$$

This constraint is a property of the Integrator and is used together with other constraints to implicitly define the monotonic function F. In contrast to defining F directly, as in (6.2), this method of defining constraints is modular, making it easy to add components to the model. It is also more intuitive, as the constraint (6.11) directly describes how to infer the input concept of the Integrator from the outputs.

To see how this works in Figure 6.4(b), suppose we assume constraints (6.6) and (6.11). Combined with $p_y = p_z$, these imply that $A \leq p_x$. Our inference algorithm finds the least fixed point to be $p_w = p_x = A$ and $p_y = p_z = V$. This solution meets the acceptance criterion in (6.9). We leave it as an exercise for the reader to determine that if instead of (6.6) we require $A \leq p_z$, then the least fixed point is $p_w = p_x = p_y = p_z = \top$, which fails to meet acceptance criterion (6.9). This would be a modeling error because the output of the Integrator cannot represent Acceleration in our ontology.

In summary, an **ontology analysis** is a concept lattice, a set of constraints in the form of (6.3) or (6.5), and a (potentially empty) set of acceptance criteria in the form of (6.7). The constraints come from component definitions, an interpretation for connections between components, and annotations made on the model by the model builder.

Chapter 7

Ptolemy II implementation

Ptolemy II [11] is a simulation tool that we use for experimenting with actor-oriented design and models of computation. It allows users to create and run hierarchically heterogeneous actor-oriented models. Our analysis framework is implemented on top of Ptolemy II, allowing model builders to create ontology analyses that they can then use to check their Ptolemy models.

In order to do so, users must draw (or reuse) a concept lattice, specify the constraints that their model has on that ontology, and package those up into an ontology analysis within their model. As a running example for this section, we will use an analysis for checking the physical dimensions of a model. The model itself simulates a system of cars that communicate over a wireless network to cooperatively control their speed in a cruise control type mechanism. To design such a control system of the following car, many different computations must be computed in parallel, representing different dynamic quantities such as velocity, acceleration and position. For our purposes, the exact details of the model are not as important as the fact that it relays many different signals dealing with physical dynamics. In a complicated model users can easily make mistakes, such as miswiring components together, or confusing the directionality of an integrator. An analysis that could check and alert for these types of mistakes would provide a useful sanity check for the model.

7.1 Ontology Analysis: LatticeOntologySolver

The first element that a model builder should notice is the LatticeOntologySolver attribute, which represents a single complete ontology analysis. A user can simply drag and drop this from the library of components into a model.

There are a few ways in which the LatticeOntologySolver can be configured. The first and most important way is that the solver direction can be configured. This distinguishes between "forward" analyses, that infer information about the data sinks in a model from the sources, "backward" analyses, that do the reverse, and "bidirectional" analyses that try to infer information as broadly as possible by inferring information in both directions. Typical

type inference tends to be a "forward" analysis, in that type information is known about the data values at the sources and is then propagated to the downstream actors. Many other analyses, such as dimension/unit analysis, constant propagation, and taint analysis share this property, and thus "forward" analysis is the default.

An ontology analysis is made up of a concept lattice, and constraints. Within the LatticeOntologySolver, it is possible to create both through the graphical interface.

7.2 Concept Lattice: Ontology

After placing an Ontology into their analysis, a user can edit it graphically. This leverages the infrastructure in the Ptolemy finite state machine editor to let users draw directed acyclic graphs. Rather than represent states and transitions, however, here the graph elements represent concepts in a concept lattice and their relationships. The ontology editor also has a few specializations for drawing concept lattices: the component library on the right hand side has been replaced with a concept-lattice specific one, and the debug menu contains a sanity checker. This simply checks that that they structure of the concept lattice is indeed a lattice.

Figure 7.1 shows an example of how one could construct the concept lattice for our "dimension system" analysis. This ontology has distinct concepts for all of the dimensions in our model, which in this case consists of dimensions of physical dynamics like velocity and acceleration. Here, concepts for conflicting constraints and unknown dimensions are explicitly included, both to make the structure a lattice and to explicitly distinguish between the case of overspecification and underspecification.

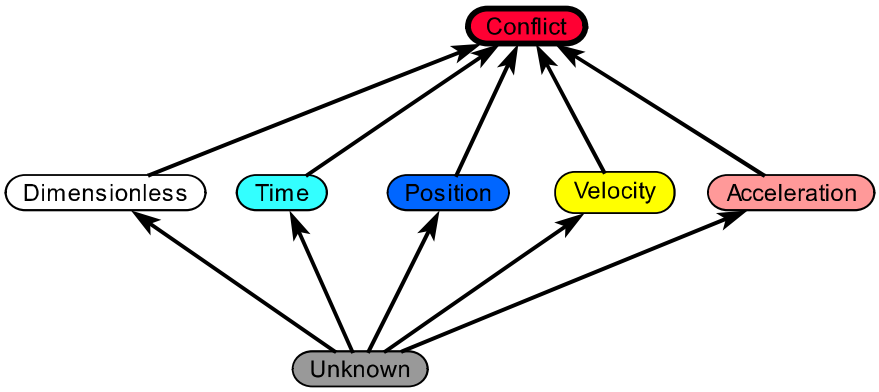


Figure 7.1: An ontology analysis representing dimensions.

After dragging concepts into the graph and renaming them, there are a couple ways that they can be configured. The first is that a concept can be set to be an **unacceptable solution** by turning off the `isAcceptable` parameter. This is the way of specifying the acceptance criteria as explained in Section 6.3. This is equivalent to adding an accep-

tance criterion for each model element specifying that it must be less than the unacceptable concept:

$$\forall x, p_x < C$$

In order for this to be valid, unacceptable parameters must be at the top of the lattice, like **Conflict** in the Dimension Lattice from figure 7.1. This property is checked automatically by selecting the "Check Concept Graph" option in the ontology editor.

The other configuration option for individual concepts is that their colors can be changed. This color is also used to show the result of running the ontology analysis, by highlighting which concepts have been inferred for each model element in the model.

7.3 Constraints

The most involved part of creating an ontology analysis is specifying the constraints. Constraints are needed for specifying relationships of concepts around and between actors, and each one may have a monotonic function. Since every actor, expression, and connection in a model should be checked, there will be a number of variables and constraints proportional to the size of the model. Specifying each of these constraints individually would be prohibitively time-consuming for all but the most trivially small models. Instead, we use a tiered approach to constructing constraints where different levels of constraints are specified in different places.

Default Constraints

Default constraints are the most general type of constraint, and are used to specify the way that actors should infer by default if not given any more specific instructions. In most cases, default constraints can be determined from the type of the analysis being performed. Generally, forward analyses should have outputs of actors inferred from their inputs, backward analyses should have inputs inferred from outputs, and bidirectional analyses should have constraints for both directions of inference. When doing a least fixed point, the most straightforward way to propagate ontology information from one model element, call it x, to another, called y, is to simply require that the concept of x be greater than the concept of y. This means that by default, actors preserve the concept information that flows through them. For actors that need to change this flow of information, class-level constraints must be used.

Class-level Constraints

The next most general type of constraints are class-level constraints: constraints that tell how a single class of actors behave. Often, an entire class of actors will behave the same way, and it would be a waste of effort to specify those throughout the model. A better way

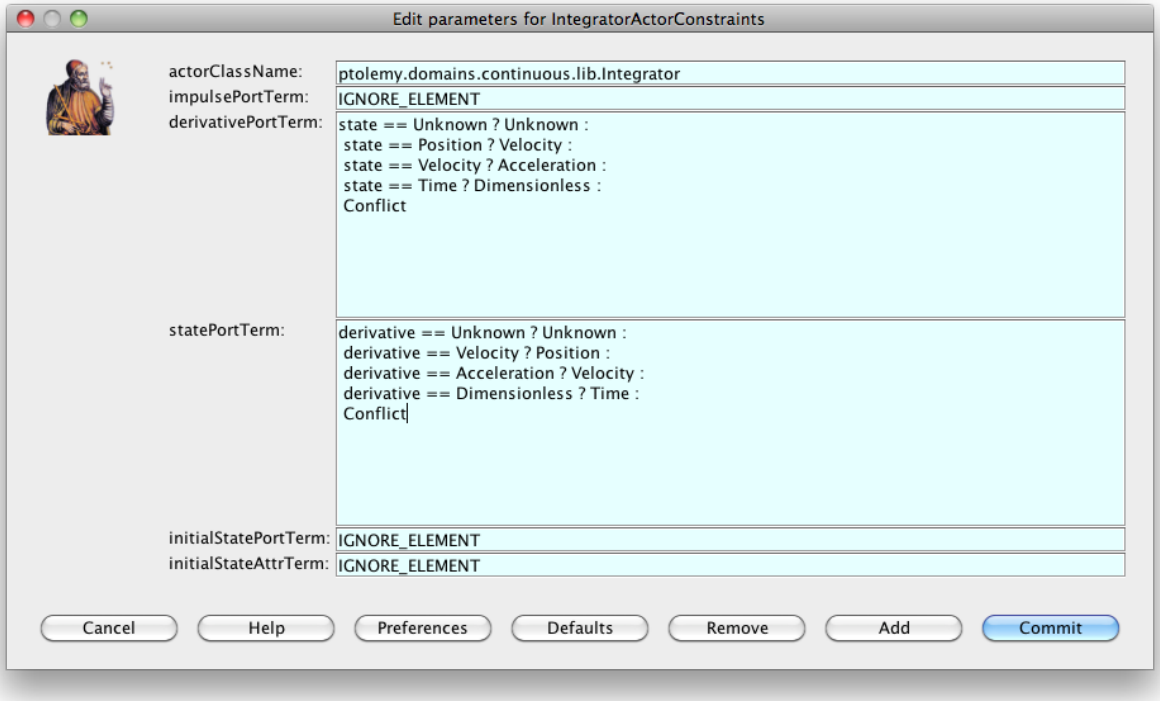


Figure 7.2: Specifying class-level constraints with the ActorConstraints attribute.

is to specify once in the ontology analysis how that particular class of actors works, and let the individual constraints for each actor be generated from that.

To use a class-level constraint, a model builder can use `ActorConstraints` attribute. Each instance of this actor can be parametrized by the name of the class of actor to which it will apply. After configuration, the icon for the ActorConstraints actors changes to mimic that of the type of actor being constrained. Then, the user can specify constraints for each port and parameter of the actor.

For least fixed point calculations, the constraints will be of the form $port \geq expression$, where *port* is the name of a port or parameter of the given actor, and *expression* is a monotonic expression of the other ports of the actor or concept values. Since the port is given in the interface of the ActorConstraints attribute, and the $\geq$ can be determined from the type of the fixed point, only the *expression* needs to be defined. The way to express these constraints is by writing them as Ptolemy II expressions. As in instance-specific constraints, these expressions are augmented so that they may refer to concept values from the concept lattice for this analysis. Additionally, class-level constraint expressions may also refer to ports or parameters of the actor as variables, which refers to the inferred concept of those model elements. If no constraint on a particular port is desired, it can be omitted by either setting the expression to $NO\_CONSTRAINTS$ or to $IGNORE\_ELEMENT$.

$NO\_CONSTRAINTS$ means that this actor does not create any constraints on its output ports, and will allow them to be anything. $IGNORE\_ELEMENT$ means that the port in question should be omitted from the analysis entirely and not inferred to have any concept. This is useful for ports like trigger ports in certain actors that are often left unconnected in certain models of computation.

We can utilize our domain knowledge of how individual components of our model work in order to determine what their constraints should be. For example, if we are given an integrator component with an input of acceleration, then we know its output should be a velocity; and if its input is a velocity, we know its output should be a position. We can use characteristics like this to infer the concepts throughout our model given a small set of initially known concepts.

Let us call the variable corresponding to the input of the integrator as *derivative* and the output as *state*. Then the characteristics of the integrator can be encoded in the following constraints:

$$\begin{aligned} d(p_{state}) &\leq p_{derivative} \\ int(p_{derivative}) &\leq p_{state} \\ p_y &\leq p_{state} \\ p_{state} &\leq p_y \end{aligned}$$

where d and int are defined as follows:

$$d(p_{state}) = \begin{cases} \bot & \text{if } p_{state} = \bot \\ V & \text{if } p_{state} = P \\ A & \text{if } p_{state} = V \\ D & \text{if } p_{state} = T \end{cases}$$

$$int(p_{derivative}) = \begin{cases} \bot & \text{if } p_{derivative} = \bot \\ P & \text{if } p_{derivative} = V \\ V & \text{if } p_{derivative} = A \\ T & \text{if } p_{derivative} = D \end{cases}$$

The d and int functions encode our knowledge about how integration and differentiation work on our dimension system, and allow for intuitive definition of the constraints.

Figure 7.2 shows an example of how such constraints can be specified in Ptolemy using the `ActorConstraints` attribute. In this example, the monotonic function constraint on integrators is expressed as a Ptolemy expression. Each port or parameter of the integrator actor can have an expression describing its monotonic function constraint. The inequality itself can be inferred from the type of fixed point being calculated, so it is sufficient to only specify the monotonic function. Expressions here may refer to concepts from the concept lattice as constants, such as $Unknown$, $Velocity$, and $Conflict$. They may also refer to other ports and parameters, which refer to the concept value that those model elements have been

resolved to. The output port of an `Integrator` actor is named *state* and the input port of is called *derivative*. This means that a monotonic constraint for inferring the output from the input will be called *statePortTerm* and may refer to the concept of the input as a variable called *derivative*.

Constraints on Ptolemy Expressions

In addition to specifying constraints for classes of actors, the same technique can be used to specify constraints on Ptolemy expressions themselves, allowing for analyses to infer through Expression actors, as shown in Figure 7.3. In order to do this, class-level constraints can be created for abstract syntax tree (AST) nodes of the Ptolemy expression language. Within the ontology analysis editor, there are several concept functions corresponding to the different types of operations available in the Ptolemy expression language. Specifying constraints with these attributes works allows constraints to be specified

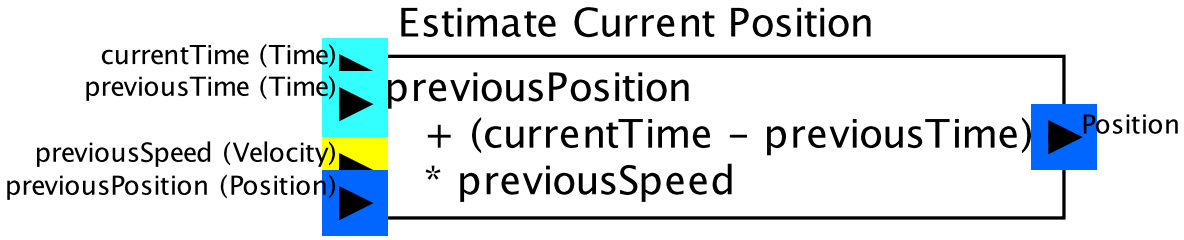


Figure 7.3: An analysis being performed over an Expression actor.

Specifying Constraints in Java

In addition to the methods of specifying constraints through the graphical editor in the Ptolemy expression language, it is also possible to write class-level constraints in Java. This uses class-level adapters in Java that correspond to a single ontology and actor or AST node type, which may then manually construct the constraints that the model element imposes. This specification allows construction of constraints that access lower-level information, such as the value of parameters or data internal to the actor.

Instance-specific Constraints

Instance-specific constraints specify a single constraint and are the least scalable. Still, there are some cases where it is impossible to infer the concepts of a model element just from the type of actor that it is or how it is used in a model. For example, a sensor component may have some semantic meaning about the dimensionality of the data that it produces that is

known only to the model builder. Instance-specific constraints give model builders a way to specify this type of information.

Unlike the more general constraints, instance-specific constraints are not specified within the ontology analysis itself. Since they are dependent on the model being analyzed, they are specified as annotations in the model. To use an instance-specific constraint, a model builder can use the `OntologyAnnotation` attribute. This is an attribute that can be placed into the model and used to explicitly specify a single instance-specific constraint. The syntax is the same as the Ptolemy expression language [24], where terms can refer either to model elements or concepts from the ontology lattice of the associated analysis. The scoping rules for the constraints allow them to refer to model elements at the same level of hierarchy, as well as concept values from the associated concept lattice.

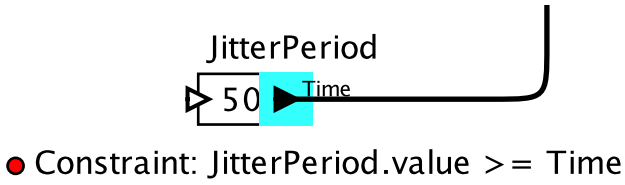


Figure 7.4: An instance-specific constraint telling specifying the dimension of a Const actor.

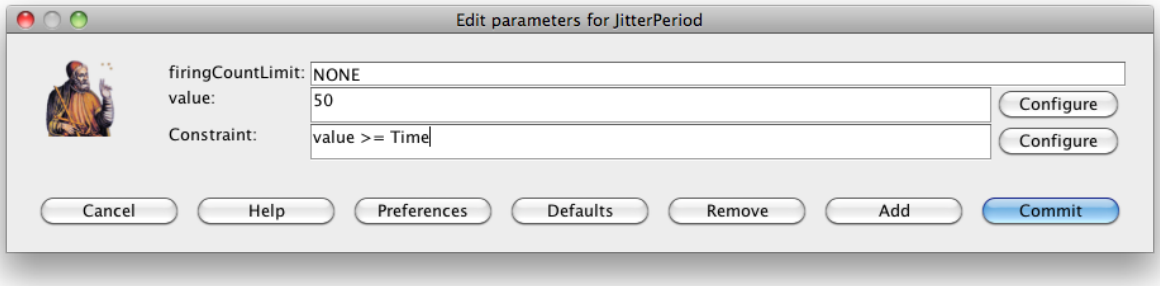


Figure 7.5: Another way to specify instance-specific constraints

Figure 7.4 shows an OntologyAnnotation attribute that specifies an instance-specific constraint for a dimension analysis. This constraint specifies that value of the JitterPeriod actor has the **Time** dimension. In some cases, having the constraint and the actor be separate in the model can add clutter and make maintenance more difficult. If an actor name changes, for example, then constraints must be updated as well to refer to the correct model elements. Another way to specify the same constraint that gets around this problem is by adding the constraint to the actor itself, rather than to the containing model. The interface for this can be seen in Figure 7.5. The semantics of the constraint expression is

the same, but the scope is different. Since the constraint is now contained within the actor being constrained, it is no longer necessary to refer to the actor's name. This makes it much easier to reuse and modify the constraint.

Each instance-specific constraint is associated with a particular ontology analysis, and this can be accessed by configuring the constraint. In the case where there are multiple different ontology analyses in the same model, this is required to specify which constraints belong to which analyses. In the case that there is only a single analysis for a model, this connection can be automatically detected and need not be manually specified.

7.4 Running the Analysis

Once the ontology analysis is specified completely, the analysis can be run on a model. The result of running the analysis is that every model element ends up associated with some concept from the concept lattice. If any of the acceptance criteria are not met, this is deemed a modeling error, and the analysis alerts the user of the error. Otherwise, the resulting resolved concepts are presented to the user. Note that since the analyses are orthogonal to the design of the model, there is no limit on how many different ontology analyses can be defined and run on one model.

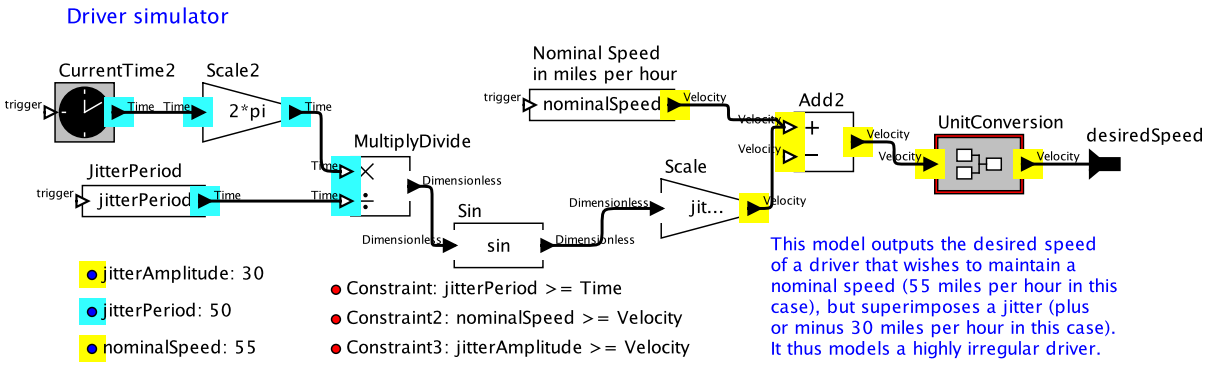


Figure 7.6: A view of part of a model after running the dimension analysis.

Figure 7.6 shows a sub-portion of the car simulation that simulates the human driver driving the leading car in the system. The driver tries to maintain a constant speed, but there is jitter, making the actual speed vary. After the dimension analysis has been performed, the dimensions of each of the modeling elements are highlighted to show the solution to the constraints. Here, we find that this model consists of times, velocities, and dimensionless quantities. If inconsistencies had been found during resolution, they would be highlighted as well as conflicts. Since conflict is not an acceptable solution for this ontology, this would also have been reported to the user. In this example, however, there are no errors.

Part III

Advanced Features

Chapter 8

Minimizing Errors

In order for ontology analyses to be useful, they must provide a useful dynamic for interacting with programmers. One way that an ontology analysis is used is to aid with the understanding of correct programs: the result of an analysis shows how the correct program is structured with respect to the concepts of the ontology. Another equally important goal, however, is to help understand the structure of incorrect programs. Making it clear to users where their errors come from is critical to helping them benefit from the provided analysis.

8.1 Motivating example

Figure 8.1 shows the top level of the model presented in Chapter 7, simulating two cars that implement a cooperative cruise control algorithm. A person drives the leading car, and the following car has its speed controlled automatically. The leading car sends information about its current velocity and position over a potentially faulty communication network, and the following car then tries to determine whether the information it receives is faulty or correct. The following car tries to follow the leading car as closely as it can, but it must not hit the leading car.

In this figure, the model has been successfully analyzed using the dimension analysis, and throughout the model signals have correctly been resolved to their physical dimensions of positions, times, velocities and accelerations. This shows us that all of these physical dimensions used in this model are used consistently. There are no bugs in its use of dimensions.

In the cases where the model uses dimensions in an inconsistent way, the analysis should alert the user of the error. Figure 8.2 demonstrates an error in the car simulator. This component implements a feedback control algorithm that tries to reach a given desired speed by applying a force proportional to the difference in the actual and desired speeds. The feedback control algorithm requires dividing the difference in speeds by a time constant to get the acceleration to be applied. The integral of this gives the actual velocity and the integral of that gives the actual position. Figure 8.2 demonstrates an error where the user failed to integrate the signal before connecting the feedback loop. This corresponds

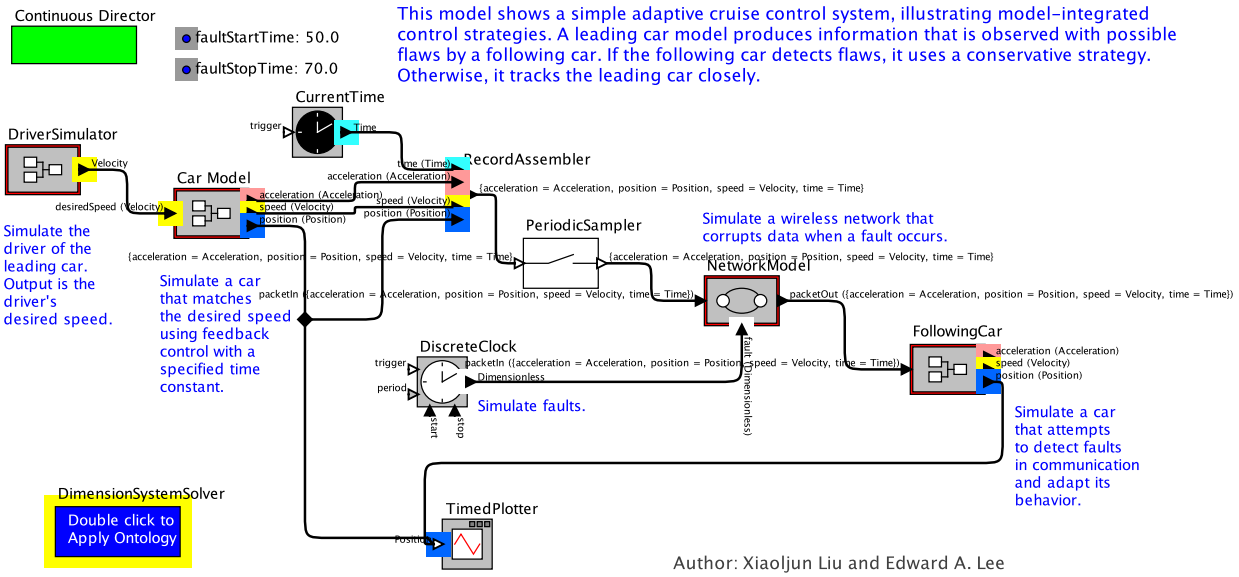


Figure 8.1: A model of the two car system with concepts inferred successfully.

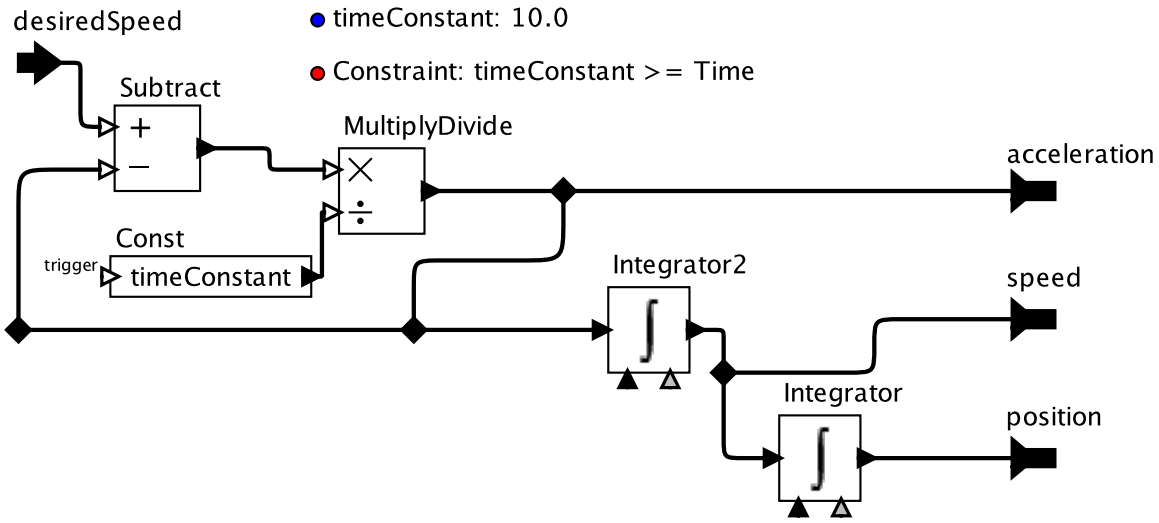


Figure 8.2: An erroneous feedback control algorithm.

to trying to subtract the actual acceleration from the desired speed, and does not make sense dimensionally. Fortunately, the dimension analysis is able to catch this type of error to the user. Unfortunately, the details of the final analysis are not particularly helpful for pinpointing the error, as can be seen in Figures 8.3 and 8.4.

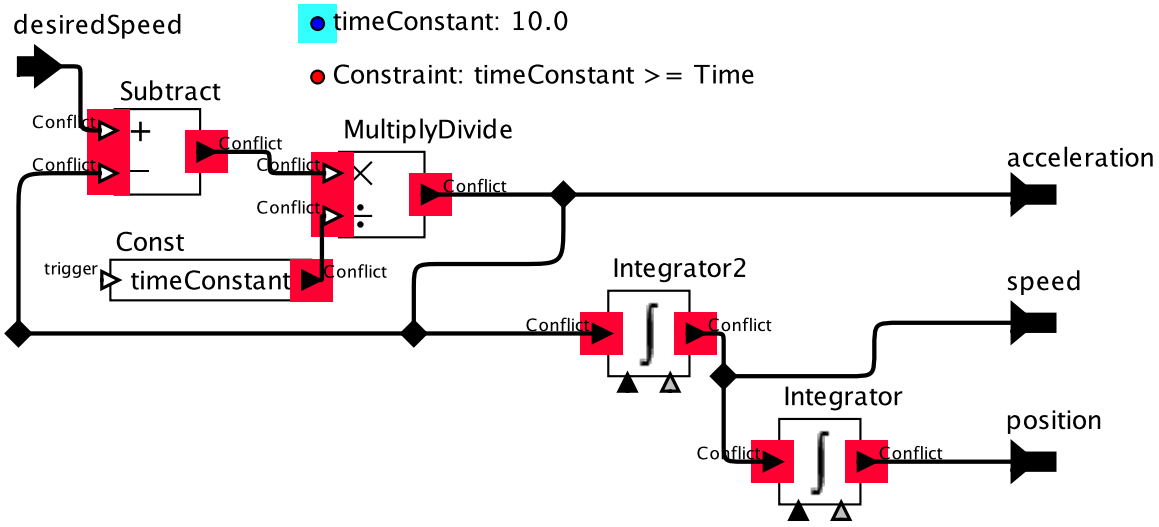


Figure 8.3: A dimension analysis reports a model as erroneous.

In order to infer as much information about the model from as few annotations as possible, the dimensionality analysis is a bidirectional analysis. This is what allows it to infer the dimensions of the source of an actor from the sink, in addition to the reverse. But one of the negative consequences of this is that information about conflicting dimensions is also propagates equally broadly.

The fact that so many signals in the model have resolved to invalid concepts makes it is difficult for users to know where the sources of errors are. We need a mechanism to isolate each error in the model into a small trace so that users can easily figure out and fix the error. Like a type system, we may not be able to catch every error at once, but the user can iteratively use this approach to fix errors until there are no errors remaining in the model.

Solving inequality constraints

Let us consider in more detail the constraints used in the dimension analysis, as presented in Chapter 6. Solving the system of constraints can be done in two steps: first, finding the least solution to the inference constraints, and then checking that the solution satisfies the acceptance criteria. By finding the least solution, we can be assured that if it does not satisfy the acceptance criteria, then no other solution will. Finding the least solution to the inference constraints can be seen a least fixed point problem. The function whose least fixed

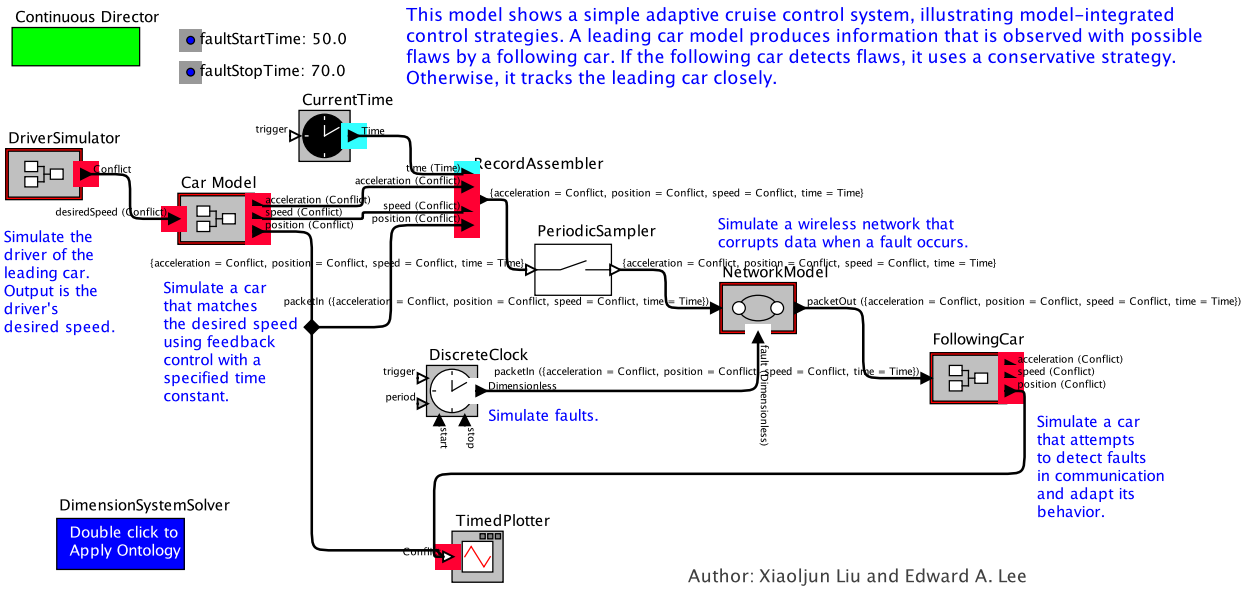


Figure 8.4: An error in one component can propagate throughout an entire model.

point we calculate has domain and range of all of the variables in the model, and is implicitly defined from the inference constraints as

$$F(x_1, \cdots, x_n) = (x_1 \vee g_1(\cdots), \ldots, x_n \vee g_n(\cdots)) \ ,$$

where the g_i functions are defined as

$$g_i = \bigvee_{\{\phi \leq x_i\}} \phi \ .$$

This simply means that each variable must move up the lattice in each iteration, and must also satisfy all inference constraints in a fixed point. In practice, however, we do not need to compute this entire function at once, and we can find the same fixed point by iteratively applying each constraint individually, as shown in the algorithm from Rehof and Mogensen, shown in Algorithm 1. Note that the fixed point depends only on the inference constraints, and not on the acceptance criteria. After reaching the fixed point, it simply checks whether this solution satisfies the acceptance criteria to see if it is a valid solution.

When the solution is invalid, this means that the original constraints were not satisfiable. With regard to the theory, it makes sense to define the result of the invalid solution to that of the least fixed point. Practically, however, we see many situations where this solution is not useful for users. In particular, if the model builder has created a model with type conflicts, this can result in all of the signals in the model resolving to the top element of the lattice. With regard to the theory, this is exactly right, but in practice, this is rarely helpful.

$x_i \leftarrow \bot, \forall i$;
while $\exists\{\sigma \leq x_i\} \in inferenceConstraints$ *with* $\sigma \not\leq x_i$ **do**
  $x_i \leftarrow x_i \vee \sigma$;
end
if $\exists\{\sigma \leq c\} \in acceptanceCriteria$ *with* $\sigma \not\leq c$ **then**
  raise Error ;
end

Algorithm 1: Algorithm for checking satisfiability and inferring concepts. Note that the result remains the same regardless of the order in which constraints are iterated.

8.2 Problem Definition

We aim to address this problem by automatically identifying and simplifying specification errors, and presenting the simplified form to the user. We want to show the user as little extraneous information as possible in order to make it as easy as possible to pinpoint where the error is coming from. We would like to preserve the interface that we have now, which colors concepts in the model according to what they were resolved to, and, in the case of error, provides a dialog that says which constraints were violated.

In case the constraints are inconsistent, rather than show all the constraints that are violated, we would like to present only a subset of them to the user. Ideally, we would like to find a minimal subset of them that reproduce that inconsistency.

There are two possible notions of minimality we can use. The stronger condition is the *global minimum*. A subset $S' \subseteq S$ is globally minimal if S' is inconsistent, and $\forall S'' \subseteq S$ with S'' inconsistent, $|S''| \geq |S'|$. That is, it is the smallest possible set of constraints in the model that are inconsistent.

A *local minimum* is a weaker condition. This simply means that removing any constraint would create a satisfying constraint set. Formally, a subset $S' \subseteq S$ is globally minimal if S' is inconsistent, and $\forall S'' \subset S$, S'' is consistent.

8.3 Solution

Since finding the global minimum will not scale well to large models, we settle for the more computationally tractable option of finding a local minimum. We leverage the delta debugging technique [52] from Zeller and Hildebrandt. There, the idea is to significantly reduce the size of a large failure tests to a small isolated part so that developers can more easily trace out errors. This can be also applied in many other cases, including ours, provided the problem meets a certain constraint of monotonicity, which can be stated as follows: If an error is absent in a given case, then it must also be absent in a strictly smaller case. Informally, this means that given more input can only add errors to a system, and not remove them. In terms of our constraints, this means that given two sets of constraints where one is a subset of the other, if the smaller set of constraints resolves to an error, larger

set must also resolve to an error. Formally, given constraint sets S_1 and S_2, where $S_1 \subseteq S_2$, then if there exists an acceptance criteria A for which A accepts the result of S_1, then A must also accept the result of S_2. This is true for our system.

Proof. Assume to the contrary that we had sets of constraints S_1, S_2, and an acceptance criteria A for which $S_1 \subseteq S_2$ and A rejects the result given by Algorithm 1 on S_1 but accepts the result of S_2.

Consider the least fixed point solution of S_2 which A accepts, call it s_2. By Algorithm 1, this means that every constraint in S_2 must be satisfied. Since the constraints of S_1 are a subset of those of S_2, all the constraints of S_1 are satisfied by s_2. Thus, s_2 is a fixed point of S_1. Now consider the least fixed point solution of S_1, call it s_1. Since it is a least fixed point, it must be no more than any other fixed point, including s_2. Thus we have that $s_1 \leq s_2$. Now consider the structure of A, which is $p_x \leq C$ for some model element x and constant C. Because $s_1 \leq s_2$, the assignment of p_x in s_1 must be less than or equal to the assignment of p_x in s_2, and thus also less or equal to C. Thus, A must also accept S_1. □

In some ways, our constraint system is a better environment for using delta debugging because of this monotonicity property. Based on this, our binary pruning algorithm to minimize the set of erroneous constraints, is shown in Algorithm 2.

```
blockSize ← constraints.size()/2 ;
while blockSize ≥ 1 do
    foreach block ⊂ constraints with |block| = blockSize do
        if ERROR ∈ resolve(constraints \ block) then
            constraints ← constraints − block ;
            blockSize ← min{constraints.size()/2, blockSize} ;
            continue while loop ;
        end
    end
    blockSize ← blockSize/2 ;
end
```

Algorithm 2: Our error reduction algorithm. *resolve*() performs the inference described in Algorithm 1.

In this approach, we try to find a block of constraints that can be removed without changing the errors. If we cannot do so, we retry with the block size halved, until the block size is eventually reduced to zero. When this is the case, we know that removing any constraint from our set will cause the errors to disappear. Thus, the set of constraints our algorithm returns is a local minimum, guaranteed to contain an erroneous constraint.

Complexity

Given a fixed lattice, the algorithm for inferring concepts runs in time linear to the number of constraints. Thus the complexity to find a local minimum is $\Theta(n^2)$ where n is the number of constraints in the whole model.

Our algorithm exhibits worst-case complexity when no constraints can be removed from the original constraint set without removing the error. This is because in this case the algorithm is not able to remove any blocks of constraints, instead being forced to search through the entire constraint set one by one.

If the original constraint set is minimal, our error reduction algorithm will reduce the block size $\log(n)$ times, halving it each time until it reaches 1. For each block size b, the inferring algorithm will run n/b times in $\Theta(n-b)$ time each. Summing up these steps, we get

$$\sum_{i=1}^{\log n} \frac{n}{2^i}(n-2^i) = n^2 - n\log n - n \ .$$

Therefore, the complexity of the whole algorithm is $\Theta(n^2)$.

8.4 Experimental Results

Table 8.1 shows the results for different test cases that contain a single error. From the table, we can see that our tool is able to remove a large proportion of the constraints in the original model. The resulting set of constraints is generally small enough that a user can much more easily identify the source of error.

Table 8.1: Debugging test cases' results.

Lattice	Test	# original constraints	# reduced constraints
Dimension system	1	453	27
	2	397	40
	3	453	11
Product line configuration	1	608	25
	2	537	29

In contrast to the results obtained in our original solver, such as shown in Figure 8.6a, the new algorithm's results are striking. The result of the new algorithm applied to the same error can be seen in Figure 8.6b. Here, the minimal error trace highlights only the actors involved in the error, and makes the mistake obvious.

In other examples, the minimal error traces are a bit more difficult to interpret. Figure 8.7 shows the component containing the error in the cooperative cruise control example from Figure 8.4. The new algorithm finds that this component is the source of the error, as it

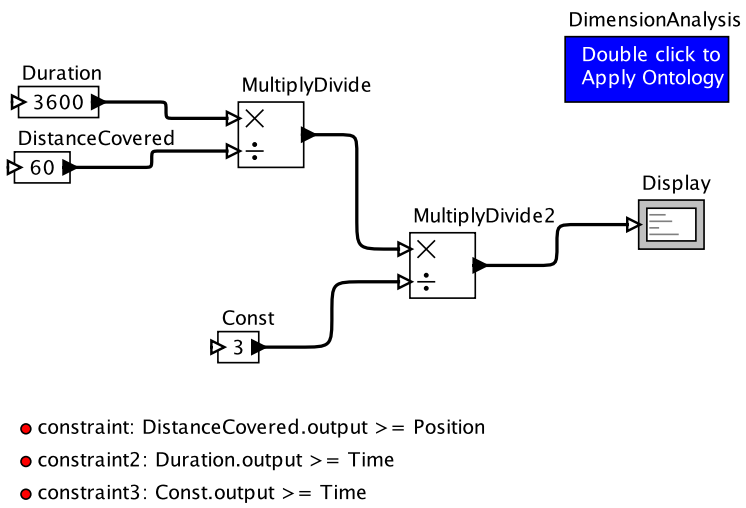


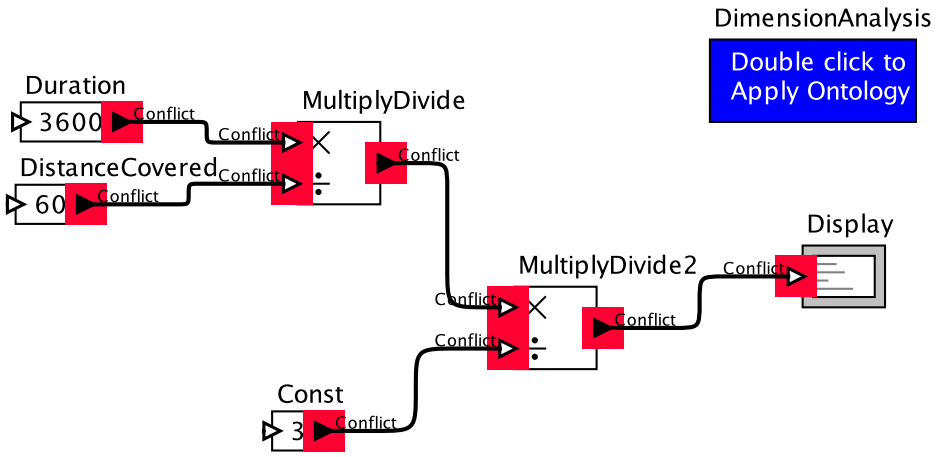
Figure 8.5: A simple model with a dimensionality error.

contains the conflict in the minimal error trace. Here, there is a path outside this level that allows the inference of the desiredSpeed port to be *Velocity*. Using our knowledge of how subtraction works, this allows us to infer that the output and other input of the Subtract actor are of the same dimension. Since this is a bidirectional analysis, all the ports connected to a relation will have the same dimension, meaning that the input to the first integrator should also have the *Velocity* concept. The integral of *Velocity* is *Position*, but the integral of *Position* is not defined, giving us an error. Removing any step in the link from the desiredSpeed input to the second integrator would remove the conflict, making this error path minimal. Even though this is a relatively complicated example, it still makes it much easier to find the error than the original fixed point from Figure 8.4.

8.5 Related Work

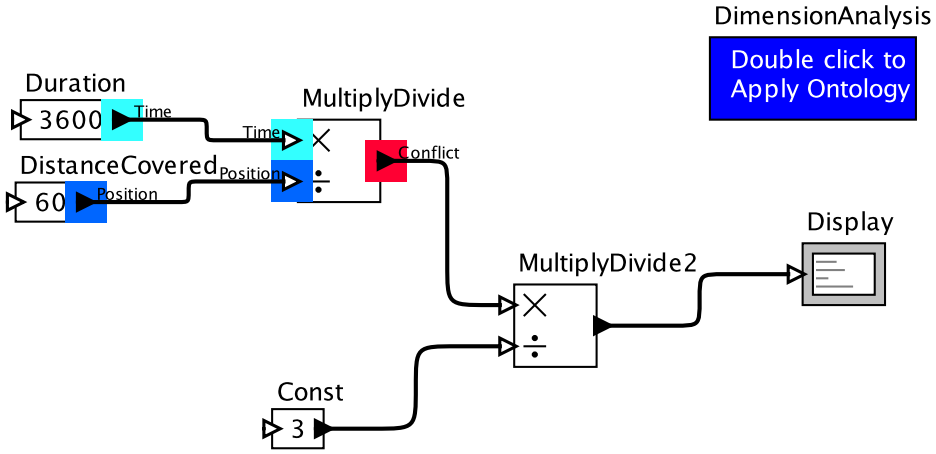
Our solution is obviously based on the Delta Debugging technique[52] for test case minimization, of which many modifications exist. Misherghi extends this approach to hierarchical delta debugging [32], which works on structural data like XML. In this case, local minima are often not good approximations, but they can leverage additional syntactic knowledge about the input to compensate. This trades off generality for quality in a particular use case. At a high level, this could be the strategy we use to improve our results. In terms of our solution, this particular approach would seem to address cases where certain constraints are not monotonic, but have a specific hierarchical structure that can be leveraged.

For textual programs, Weimer et al. [50] propose genetic algorithms for automatically finding patches for error programs. Griesmayer et al. work [41] focuses on automatic repair of Boolean C programs using SMT solvers. We suppose that similar techniques could be



- constraint: DistanceCovered.output >= Position
- constraint2: Duration.output >= Time
- constraint3: Const.output >= Time

(a) An ontology analysis catching an error in the model with the original algorithm.



- constraint: DistanceCovered.output >= Position
- constraint2: Duration.output >= Time
- constraint3: Const.output >= Time

(b) The new algorithm finds minimal error paths.

Figure 8.6: Different ways to catch an error.

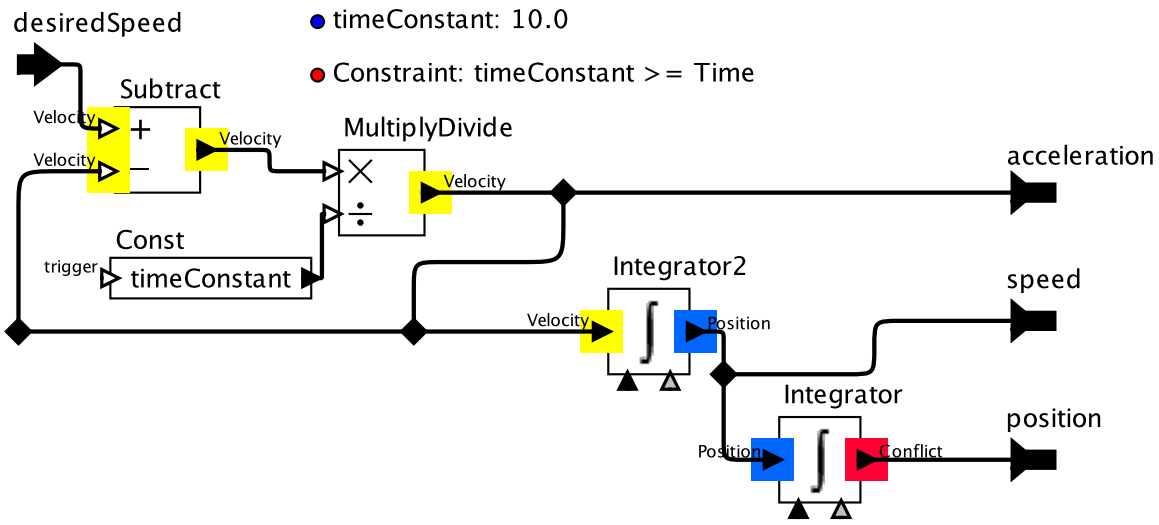


Figure 8.7: An error trace after running debugging tool.

applied to our system to more intelligently suggest fixes to the most probable errors.

The program slicing techniques in [46] isolate the parts of a program that can have an effect on a particular location in the program. These can also be used to narrow down the possible sources of an error in a program.

8.6 Conclusion

Here we have presented a tool that is able to infer concepts in actor-oriented models. It displays all of the resolved concepts in the case that inference completes successfully, and shows a minimal trace that causes the error in the case that inference raises an error.

One of the biggest strengths of our approach is also its greatest weakness. That is that it does not distinguish between different types of constraints. This is a strength, since it means that the approach is very general, is relatively simple to implement, and will work on models regardless of the types of constraints that they contain. It is also a weakness, however, because there is information that we are not taking advantage of. In particular, our tool has multiple different types of constraints: there are general constraints given by the solver, there are class-level constraints given for a particular class of actors, and there are annotation-based constraints given on an individual basis by special annotations in the model. It is much easier for an end user to change the annotation-based constraints than the solver constraints, and because of this, it is probably more likely to find mistakes in annotation-based constraints.

We could imagine an error-resolution procedure that took these facts into account and

was more likely to remove constraints that were from annotations. We could also imagine a tool that in some cases is able to guess exactly which constraint is erroneous, and make a suggestion explicitly to the user about what to change. This is potential future work.

Chapter 9

Infinite Ontologies and Ontology Composition

The dimension ontology presented in Chapter 6 is a nice way to catch certain types of dimension errors, but is ultimately insufficient for describing full units. By distinguishing between 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 Chapter 4. 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 must represent the scale and offset of separate units within a dimension, which cannot be contained in a simple finite lattice structure. Additionally, real programs make use of structured data types which provide useful abstractions, but whose properties do not fall neatly into the finite lattice restrictions given in [27].

In this chapter, 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.

9.1 Infinite Ontology Patterns

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.

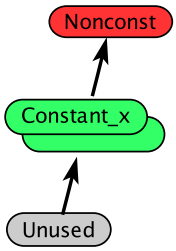


Figure 9.1: Using a `FlatTokenInfiniteConcept` to represent an infinite flat lattice.

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 9.1. 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 sub-parts.

One nice property of the infinite flat lattice pattern is that it does not increase the height of the lattice. The resolution algorithm we use from Rehof and Mogensen [40] 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.

Constant Propagation Analysis

A simple example of an analysis that makes use of this type of lattice is constant propagation, which 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 can be seen as an abstract interpretation [9] of the signals in a program in which all potentially non-constant signals are all abstracted away. This produces the infinite flat lattice structure shown in Figure 9.2, represented in our software with a `FlatTokenInfiniteConcept` as shown in Figure 9.1. 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 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.

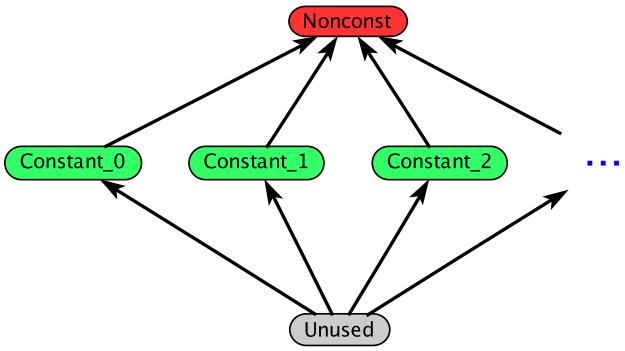


Figure 9.2: An infinite flat lattice for doing constant propagation.

Component		Constraint	
Add	$\oplus(x,y) =$	$Unused$	if $x = Unused$ or $y = Unused$
		$x + y$	else if $x < Nonconst$ and $y < Nonconst$
		$Nonconst$	otherwise.
Subtract	$\ominus(x,y) =$	$Unused$	if $x = Unused$ or $y = Unused$
		$x - y$	else if $x < Nonconst$ and $y < Nonconst$
		$Nonconst$	otherwise.
Multiply	$\otimes(x,y) =$	$Unused$	if $x = Unused$ or $y = Unused$
		$x \times y$	else if $x < Nonconst$ and $y < Nonconst$
		$Nonconst$	otherwise.
Divide	$\oslash(x,y) =$	$Unused$	if $x = Unused$ or $y \leq Constant\_0$
		x/y	else if $x < Nonconst$ and $y < Nonconst$
		$Nonconst$	otherwise.

Table 9.1: Constraints for the constant propagation example

The constraints for the basic binary operations of addition, subtraction, multiplication, and division are given in Table 9.1, 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 usage of a constant propagation analysis is shown in Figure 9.3. This simple model has two types of source actors at the left: the Const actors each produce an unchanging output throughout the execution, whereas the Ramp actor produces a time varying sequence. The Ramp actor can represent any other non-constant sources that exist in real systems such as sensors, network packets, or user input. Even in the presence of non-constant sources, however, subsections of the model may be constant. In Figure 9.3, for example, the analysis computes 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 constant values of constants signals are; in many cases, this is just as important. If 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.

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 that is parametrized with respect to the type of the elements of the array. In this way, a recursively defined hierarchy of array

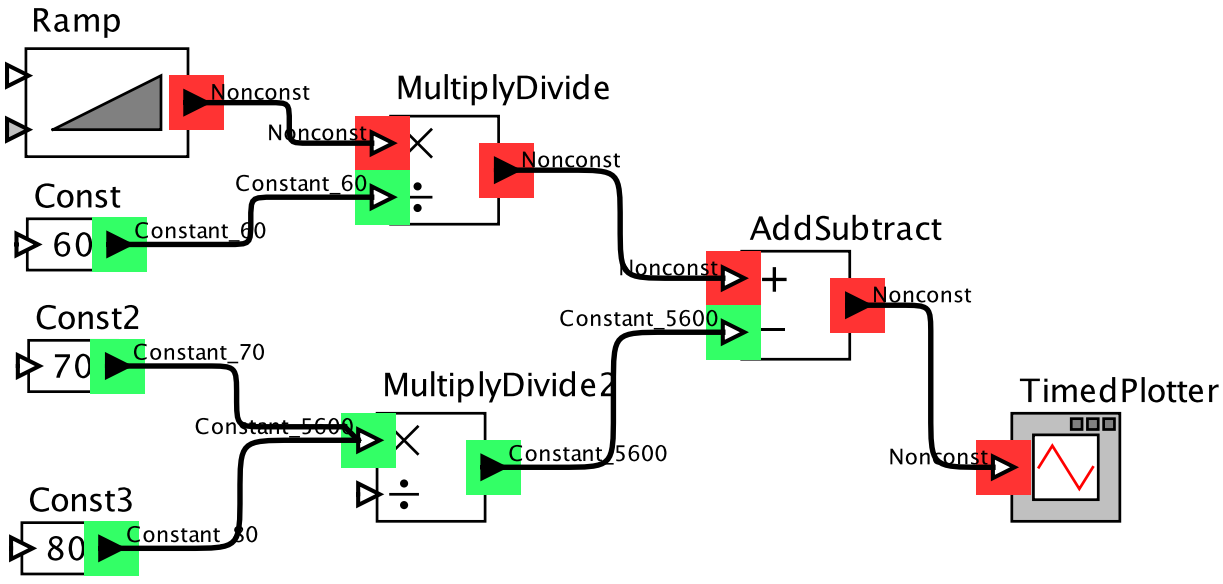


Figure 9.3: A model on which constant propagation analysis has been applied.

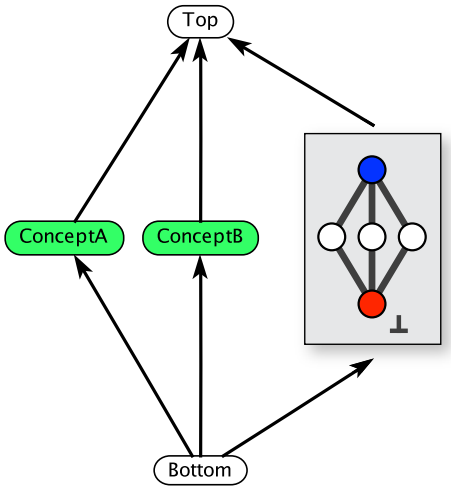


Figure 9.4: An infinite recursive lattice can include references to itself.

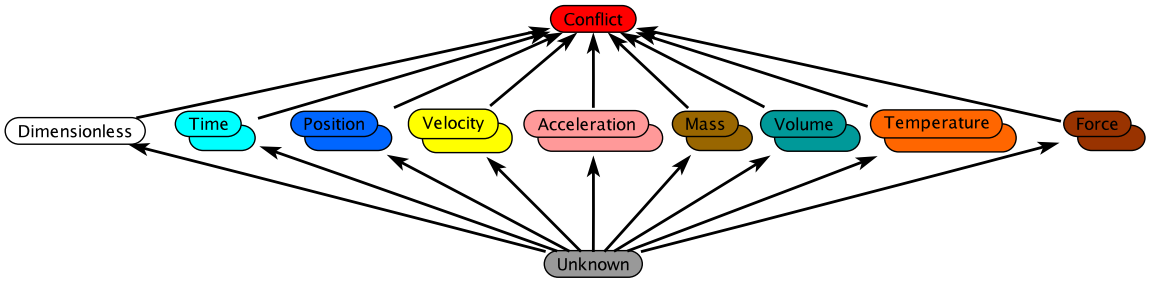


Figure 9.5: A generic lattice for unit analysis.

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, etc.

In these cases, the lattice that represents all of the possible types becomes not only infinite, but also infinite in height. This means that we lose some of the algorithmic bounds that we had with finite-height lattices, but gain the richness of patterns that can be expressed as structured data types. In addition, there are heuristics that allow us to deal with many cases decidably. In Section 9.3 we discuss specifically the design of infinite recursive lattices for supporting records of concepts, and these issues are discussed in more depth there.

9.2 Unit Systems

One of the drawbacks of the dimensional analysis presented in Chapter 7 was that it could not check for inconsistencies arising from different units of the same dimension, such as having

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 9.6: Attributes of the *Time* base dimension.

one component expecting 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 9.5 shows a lattice that contains dimensions that cover several base SI units for dimensions 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 [18] we explicitly enumerate all of the dimensions that will be considered for a particular model. This means that any single unit ontology cannot hope to be comprehensive, but it also means that we are able to distinguish semantically between dimensions that are composed of the same elementary units. For example, 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. There are 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

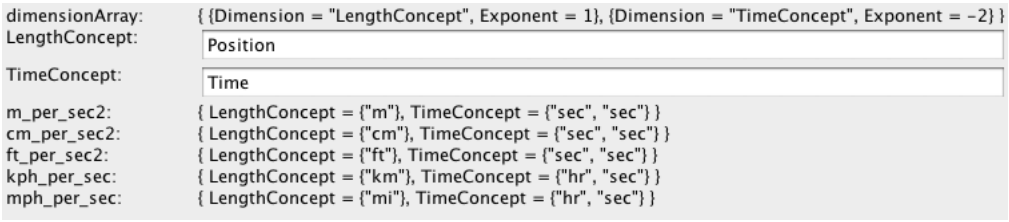


Figure 9.7: Attributes of the *Acceleration* derived dimension.

expressed as products or quotients of other dimensions.

Base dimensions are the building blocks of our unit systems. Within a given base dimension, as shown in Figure 9.6, 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 choose cm (centimeters) 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 base 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 9.7. 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.

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 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 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 need to express its units it terms of kelvins. This intuitively makes sense, since the result of multiplying or dividing units with non-zero offsets is not well defined.

In cases where there are unit mismatches, users may want to automatically translate between units. We have created actors that leverage the information in the unit ontology in order to aid in this process. These conversion actors are described in more detail in Section 9.2.

Note that other work with similar aims of adding unit information and static checking to programming systems includes packages for SystemC [30], Modelica [4], SCADE [44], and Ada [16]. Work has been done in functional languages to extend the algorithms of type

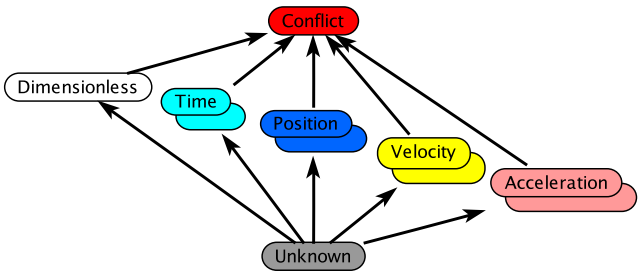


Figure 9.8: A lattice for unit analysis of the two-car system.

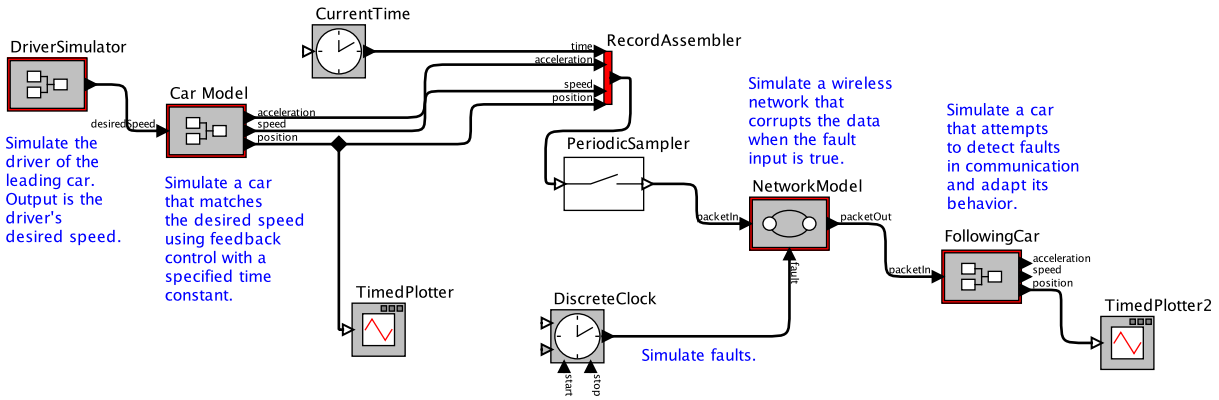
inference to work for unit inference [19], decreasing the number of annotations required. A proposal for unit types in the hybrid-system modeling language CHARON is presented in [3], and also stresses unit inference to decrease annotation requirement. Modules are each assigned a single unit system, and conversion between units may occur at module boundaries. 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 unit systems as one type of analysis. This means that our tool allows model builders to create unit systems that are domain specific, make semantic distinctions between units that would not be distinguishable in a general unit system, or combine units with other semantic concepts.

Example Model: Adaptive Cruise Control

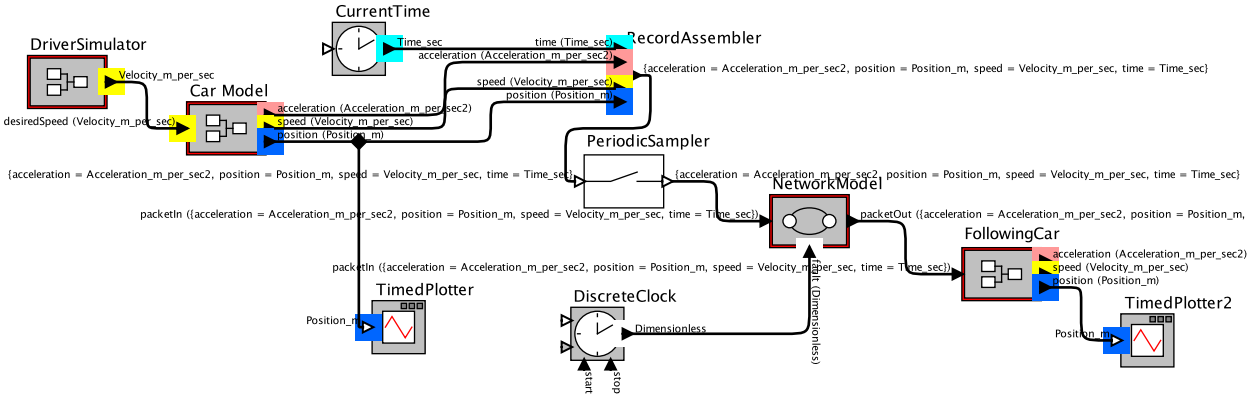
Here we present an example of a model used in a cyber-physical system, and then examine what types of analyses we may run on this model and how they can aid us in finding errors and better understanding our model. We use an example model that allows simulation of a system of two vehicles connected by 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 [27], is shown in Figure 9.9a 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 potentially faulty wireless link to the following car. The following car then uses the information received over to follow the car as safely as possible, in a system of collaborative cruise control.

We take as a starting point the dimension analysis presented in [27], but take issue with some of the impractical restrictions that they place on their ontologies. 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 many common errors that are the result of incorrect units within a dimension.



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



(b) Completed unit resolution.

Figure 9.9: Unit resolution of the adaptive cruise control example.

To address these shortcomings, we present our infinite unit lattice for this adaptive cruise control model in Figure 9.8. This has the same dimensions as the lattice presented in [27], 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 produce outputs in the same units that they accept inputs, so it is simplest to only specify the behavior of actors which differ from this behavior. For defaults, we allow the 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 9.2. In reality, multiplying or dividing by a unit with a non-zero offset will result in a conflict, since the semantics of

Multiplication

$$\otimes(x,y) = \begin{cases} Unknown & \text{if } x = Unknown \text{ or } y = Unknown \\ Position(scale_x \times scale_y) & \text{if } x = Time(scale_x) \text{ and } y = Vel(scale_y) \\ & \text{ or } x = Vel(scale_x) \text{ and } y = Time(scale_y) \\ Vel(scale_x \times scale_y) & \text{if } x = Time(scale_x) \text{ and } y = Accel(scale_y) \\ & \text{ or } x = Accel(scale_x) \text{ and } y = Time(scale_y) \\ y & \text{if } x = Dimensionless \\ x & \text{if } y = Dimensionless \\ Conflict & \text{otherwise.} \end{cases}$$

Division

$$\oslash(x,y) = \begin{cases} Unknown & \text{if } x = Unknown \text{ or } y = Unknown \\ Accel(scale_x/scale_y) & \text{if } x = Vel(scale_x) \text{ and } y = Time(scale_y) \\ Vel(scale_x/scale_y) & \text{if } x = Position(scale_x) \text{ and } y = Time(scale_y) \\ Time(scale_x/scale_y) & \text{if } x = Position(scale_x) \text{ and } y = Vel(scale_y) \\ & \text{ or } x = Vel(scale_x) \text{ and } y = Accel(scale_y) \\ Dimensionless & \text{if } D_x = D_y \text{ and } scale_x = scale_y \\ x & \text{if } y = Dimensionless \\ Conflict & \text{otherwise.} \end{cases}$$

Table 9.2: Manual constraints for adaptive cruise control unit system example.

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 multiplication and division constraints for a general unit system are given below.

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:

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

Here *divideDim* performs analogously 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.

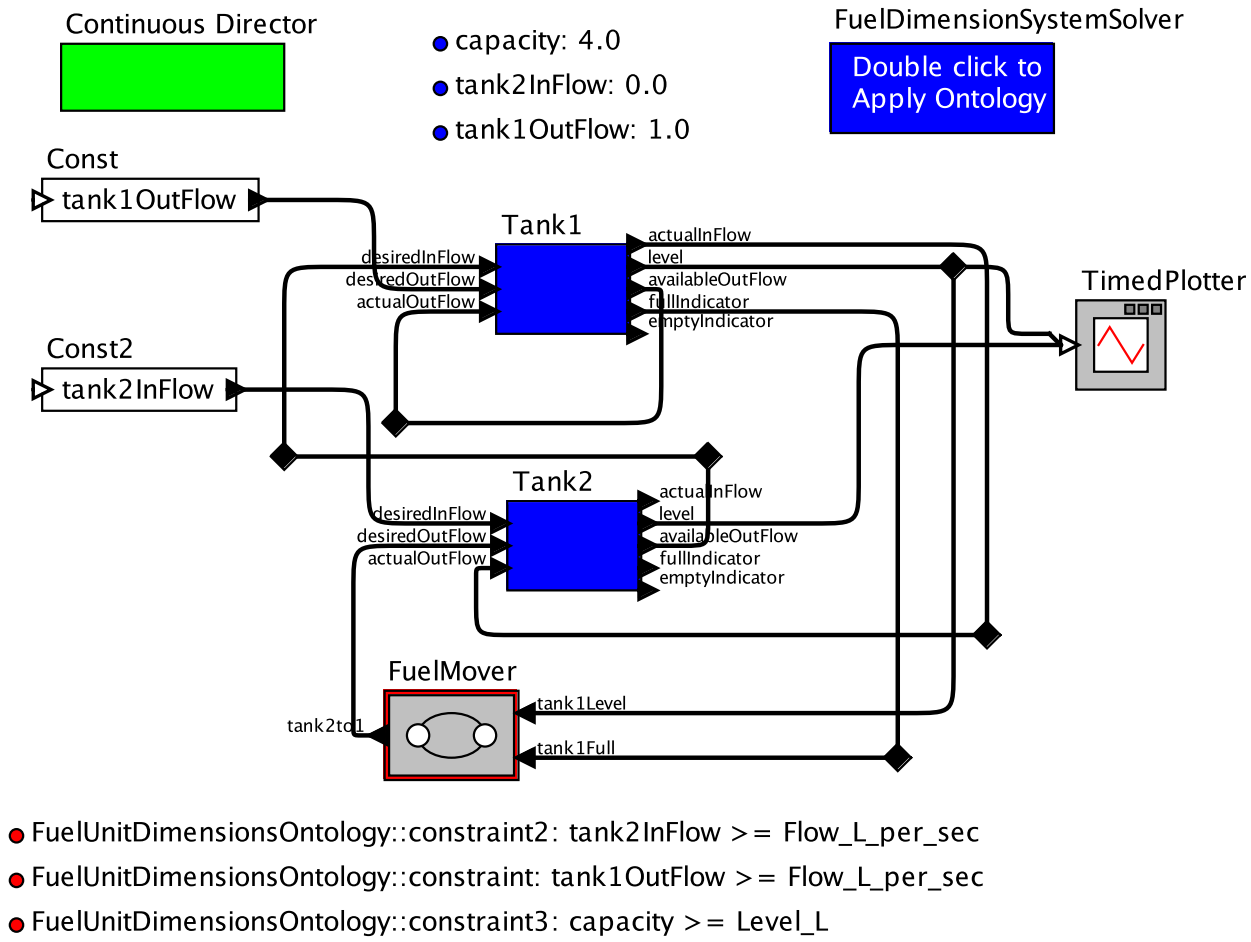


Figure 9.10: Model of a two-tank aircraft fuel system.

Example Model: Fuel System

By no means are unit systems only useful for the standard dimensions presented here. In [10], Derler et al. present an example of a fuel system in an 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 9.10. 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 allows 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

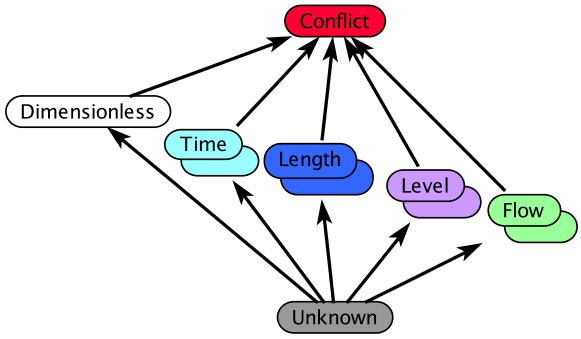


Figure 9.11: A lattice for unit analysis of a fuel system.

the basic units of length and time gives the complete ontology shown in Figure 9.11.

Like in the adaptive cruise control example, we 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 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 9.12, with the coloring and naming of the inferred concepts drawn from the ontology in Figure 9.11.

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 UnitsConverter actor to the model, as shown in Figure 9.13. This allows conversion from one unit to another within the same dimension, and the UnitsConverter 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 is a unique property of this actor. We think that the

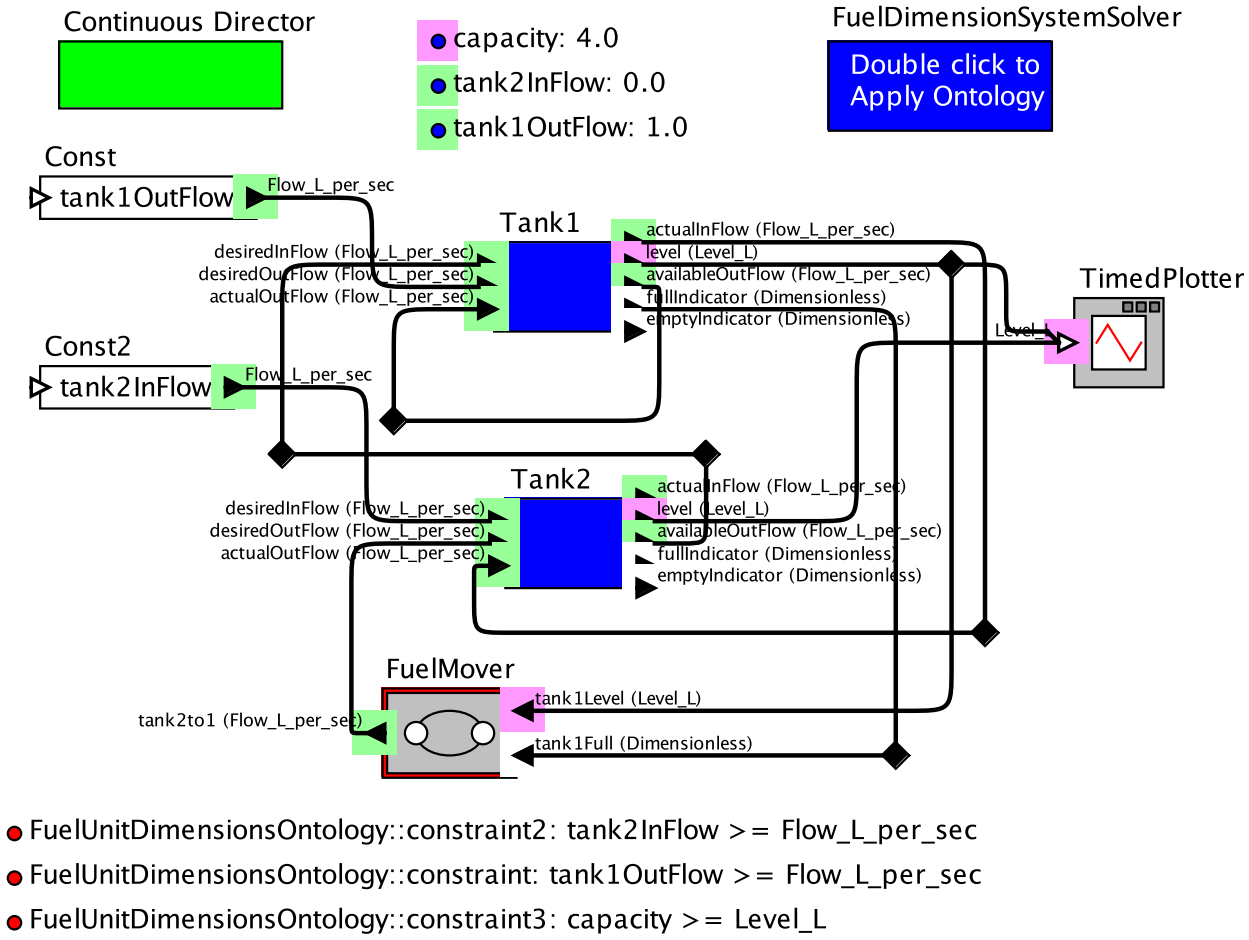


Figure 9.12: The result of inferring units over the fuel system model.

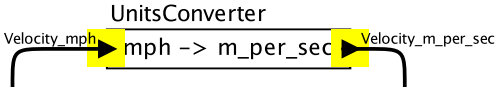


Figure 9.13: Using a UnitsConverter to convert from mph to m/s.

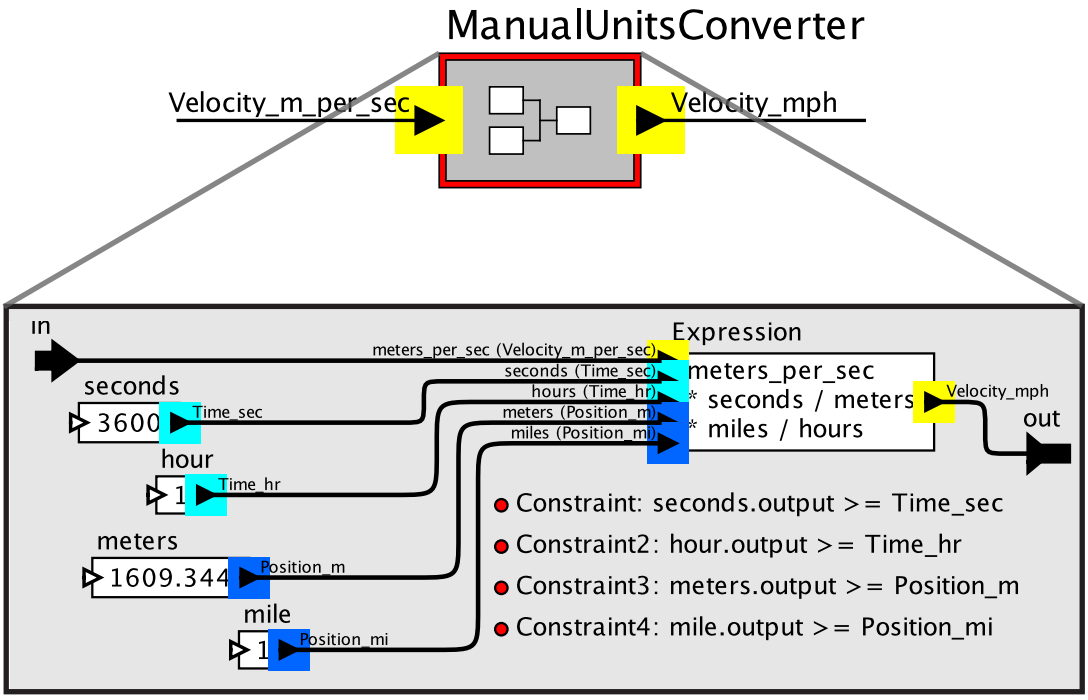


Figure 9.14: Using manual unit conversion to convert from m/s to mph.

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.

Replacing a UnitsConverter actor with a checked manual conversion

While the `UnitsConverter` actor is very useful for automatically inferring the conversion from one unit to another, there are some situations in which model builders would likely want to steer away from its use: Models with UnitsConverter must have a unit ontology analysis present in order to be run. Thus, models with all unit conversions done explicitly are more portable.

Fortunately, it is not difficult to create a manual conversion that continues to make the same unit checks as a UnitsConverter actor, but without the drawbacks. Figure 9.14 shows one such example, which manually does the conversion from meters per second to miles per hour by multiplying and dividing by the conversion factors. This includes the information that 1 hour is equivalent to 3600 seconds, and that 1 mile is 1609.344 meters. Because each of the conversion factors includes its units, the entire conversion computation can be checked by the ontology analysis. By dropping this actor in as a drop in replacement for a $m_per_sec- > mph$ UnitsConverter actor, one can also see that the run-time behavior is the

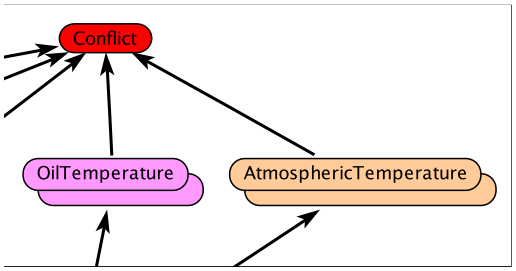


Figure 9.15: One way to model two semantically distinct temperatures separately.

same. Unlike a UnitsConverter actor, however, this type of manual conversion continues to work correctly in the absence of the unit system ontology analysis.

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 distinctions between arbitrary concepts. Thus, users can model distinct concepts from their domain even if they traditionally have the same units.

Imagine, for example, that the car in our adaptive cruise 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 9.15 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.

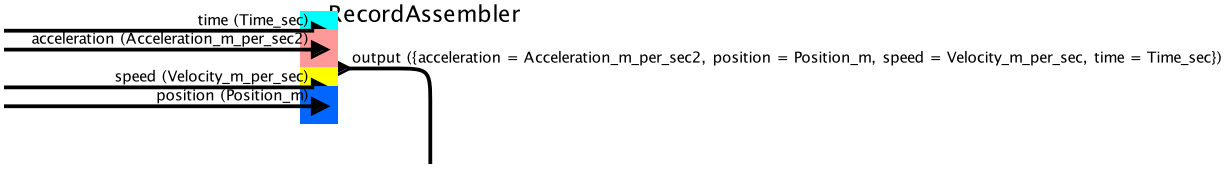


Figure 9.16: Unit resolution over the RecordAssembler actor inferring a record of concepts.

9.3 Concepts with Structured Data Types

Another shortcoming of the basic ontology analysis presented in Chapter 6 is that it does not gracefully handle structured data types of Ptolemy, because signals that carry complex structured data types cannot be simply classified as a single dimension like *Acceleration* or *Time* like other signals. This means that models that can be easily analyzed with the basic ontology analysis are unable to leverage useful abstraction mechanisms like Ptolemy II record types.

A record type is one example of a structured data type that can be used to simplify models. A record type is 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 RecordDisassembler 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 easily in Ptolemy, but doing so exposes a shortcoming in the 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 9.1.

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 9.16 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.

When using record concepts, it becomes possible to make models more abstract and simplify connections. Figure 9.17 shows a simplified version of a network model from the cruise control example. Here, rather than deal with all of the individual signals for times, positions, velocities, and accelerations that travel over the network, it can deal with an abstraction of network packets. If we chose to model the network differently, with a more

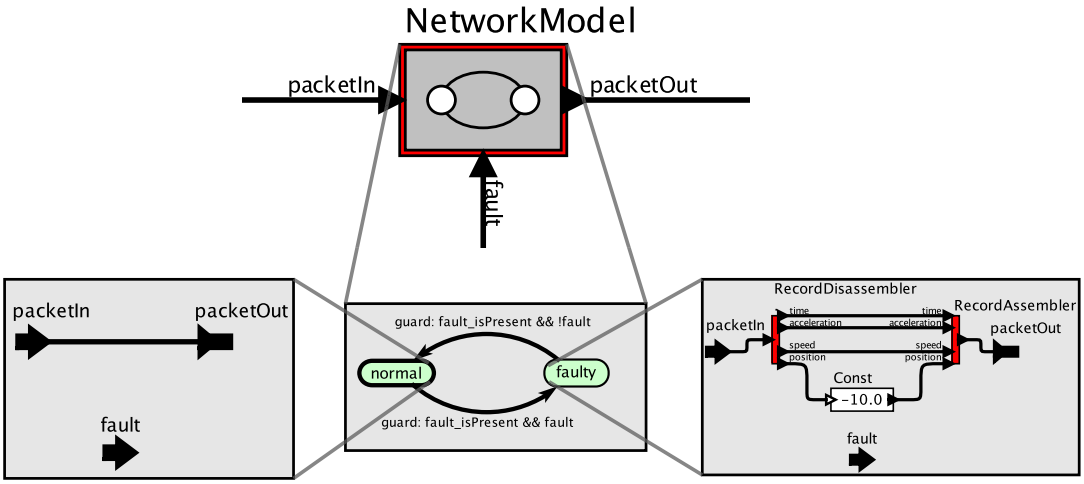


Figure 9.17: The interface of a network model becomes much simpler with records.

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 it carried. This makes models more abstract and reusable, and is an important workflow that we aim to support.

One danger 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 [55]. They deal with this problem by placing limits in 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.

9.4 Combining Ontologies

Sometimes, a model builder may want to run two separate analyses on the same model. Nothing prevents this in the current infrastructure. Since ontology analyses are orthogonal to the model under inspection, a model builder can create many separate analyses for a single model. In some cases, however, a model builder may want to combine information in different analyses, using information from one analysis to aid another. It is theoretically possible to create a more broad new analysis that includes concepts from the original analyses, but this

requires much work to be duplicated from the original analyses. It would be more useful to be able to reuse existing complete analyses through a mechanism of ontology analysis composition.

Product Lattices

One way to combine lattice-based ontologies is by taking the Cartesian product. Each element in the Cartesian product of two concept lattices refers simultaneously to concepts from both of the original concept lattices. In addition, the product order is a natural way to order the elements of this composition. Given two concept lattices A and B, the elements in the Cartesian product $A \times B$ ordered by the product order also form a lattice. Additionally, this order preserves the ordering of the elements of A given a fixed B, and vice versa. These properties make the Cartesian product with the product order a natural way to form a combined concept lattice. Here, we refer to a lattice formed in this way as a **product lattice**.

Product Constraints

Just as it is possible to define the partial order of a product lattice from the component lattices, it is also possible to define the constraints of a product ontology from the component constraints.

If we used this approach for all of the new constraints, however, the analysis would be identical to the result from running the two component analyses separately. The composed ontology analysis derives its additional power from the fact that it may specify constraints that simultaneously consider concepts from multiple different original ontologies. Thus, it is also possible for users to specify new constraints that override the original constraints of the component ontologies.

Abstract Interpretation Example

9.5 Conclusion

In this chapter, we have presented a system for supporting useful patterns of infinite ontologies, and ways to compose ontologies together into new ontology analyses.

One important class of analyses enabled by infinite ontologies are unit systems. Our framework allows user-specified unit systems that include notions of base dimensions and derived dimensions. It specifies reasonable default constraints that model 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.

In addition to the infrastructure supporting unit systems, the framework contains mechanisms for supporting arbitrary user extensions for new types of infinite ontologies. 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.

Chapter 10

Self-analysis: Checking Monotonicity

The algorithm that we use to perform our ontology analysis makes certain assumptions in order to guarantee the existence of a unique result. The ontology itself must be a complete lattice structure, the individual constraints must have the structure presented in Chapter 6, and concept functions must be monotonic. The first requirement is easy to check from the concept lattice editor, and non-lattice structures can be explicitly rejected. For monotonicity, however, end users were saddled with the responsibility of making sure that any concept functions they wrote for constraints were monotonic functions. Preferable to that would be to have a method for automatically determining the monotonicity of user-provided Ptolemy expressions. Since the Ptolemy expression language [24] is Turing complete, it is impossible to completely check for any non-trivial program property, including monotonicity. That does not prevent us, however, from writing a sound conservative check for monotonicity that flags certain expressions as potentially non-monotonic. Since most concept functions used in real analyses use only a subset of the expression language, there is potential to make a monotonicity analysis that works well for the subset of the language that is used for writing real concept function expressions.

10.1 Motivation

In order to be useful for users writing ontology analysis, the framework must be correct, deterministic, and understandable. Given the assumptions that the ontology is a lattice, and that all constraints are monotonic, we have already shown that there exists a single unique least fixed point to any system of constraints. When users provide the constraints, however, it is impossible to guarantee that the provided constraints are monotonic. Let us consider an example of what can go wrong if we then try to run an analysis composed of non-monotonic constraints.

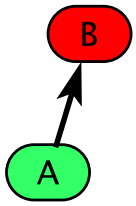


Figure 10.1: A very simple concept lattice.

Example 1

Assume that we have two model elements, x and y. Assume that the concepts for these model elements are given by p_x and p_y, and their domain is the very simple concept lattice as shown in Figure 10.1. Now, assume we were given the following two constraints:

$$f(p_y) \leq p_x$$
$$f(p_x) \leq p_y$$

where f is defined as follows:

$$f(x) = \begin{cases} B, \text{if } x \leq A \\ A, \text{otherwise} \end{cases}$$

If we treat this two element, two constraint system as a complete ontology analysis, we can try to infer the resolved concepts. In this case, however, there exists no least solution. There are in fact two incomparable solutions to the constraints: $(p_x, p_y) = (A, B)$, and $(p_x, p_y) = (B, A)$. Neither solution is less than the other in the concept lattice, and there exists no other solution less than these solutions.

The problem here is that the function f is not monotonic. Given a set of monotonic function constraints, however, there is a unique least solution given by the Rehof and Mogensen algorithm.

10.2 Related Work

There are relatively few static analyses that focus on analyzing the monotonicity of functions. Our main starting point was [35], which defines a similar analysis with a similar motivation. Unfortunately, there are problems with that analysis being both too conservative and unsound. We devote Section 10.5 to discussing these issues more thoroughly.

Since we are interested in finite lattices, the notion of monotonic functions and continuous functions coincide. There exist more analyses that deal with function continuity such as [6] by Chaudhuri, Gulwani, and Lublinerman. Unfortunately, these analyze functions according to the definition of continuity with regard to real-valued functions, that "small changes in the inputs cause small changes to the outputs," rather than the Scott continuity that we are interested in.

10.3 Running Example

As one example, let us consider what the constraint of the integrator actor would look like when doing our dimension analysis over the dimension ontology from Figure 7.1. If we do not know the dimension of the input to the integrator, we cannot know its output. The integral of acceleration is velocity, of velocity is position, and of a dimensionless quantity is time. Any other uses of the integrator give results that cannot be analyzed by our given ontology.

Represented as a Ptolemy II expression, this gives us the following:

Expression 10.1: Ptolemy II Expression for Integrator Constraint

```
(x <= Unknown) ? Unknown :
(x <= Acceleration) ? Velocity :
(x <= Velocity) ? Position :
(x <= Dimensionless) ? Time :
Conflict
```

We will use this expression as a running example to compare different monotonicity analyses.

10.4 Definitions

We say that one element of a lattice covers another if they are ordered and immediate neighbors in the lattice order. More formally, we say that x **covers** y, written $y <\cdot x$, if $y < x$ and $y < z \leq x \implies z = x$. We also say use the term **cover set** of y to refer to all of the elements that cover y:

$$\text{cover}(c) = \{x \mid c <\cdot x\}$$

We also refer to the set of all elements in a lattice below a certain value c as the **down set** of c:

$$\text{down}(c) = \{x \mid x \leq c\}$$

Expressions (over lattices)

For the sake of simplicity, we will assume that the expressions that we are dealing with each contain only a single free variable, which we will often refer to as x. In a slight abuse of notation, we use function application to mean substitution of the free variable. Thus, $e(\bot)$ represents the expression e with all instances of the free variable x replaced with $\bot$.

In addition to the concept lattice over which our functions and expressions are defined, there is another concept lattice of importance: the monotonicity lattice. This is the concept lattice over which our monotonicity analysis is defined, and is shown in Figure 10.2. After performing the monotonicity analysis, an expression can be categorized as any of the concepts from this lattice.

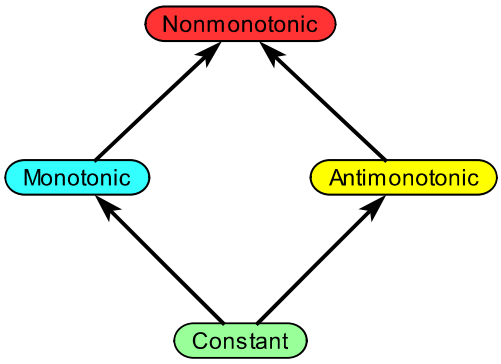


Figure 10.2: Concept lattice used for monotonicity analysis.

When an expression is resolved to be **Constant**, that means that its value does not depend on the free variable. Thus,

$$\forall x, y : \ e(x) = e(y)$$

An expression that is resolved to be **Monotonic** means that it is order-preserving:

$$\forall x, y : \ x \leq y \implies e(x) \leq e(y)$$

whereas an expression that is **Antimonotonic** means that it is order-reversing:

$$\forall x, y : \ x \leq y \implies e(x) \geq e(y)$$

Note that constant expressions are both monotonic and antimonotonic. Expressions that are resolved to **Nonmonotonic** mean that the analysis was not able to prove anything about them. Since we aim to have a safe analysis, an expression that is not monotonic or antimonotonic must resolve to Nonmonotonic. Since our analysis may be conservative, however, an expression resolving to Nonmonotonic can not guarantee anything about the expression's monotonicity.

10.5 Existing Work (2002 Murawski Yi paper)

We begin our analysis with an overview of the state of the art in monotonicity analysis. The most relevant paper is [35], which presents a monotonicity analyzer for a simple lambda calculus designed to be part of their Zoo program analysis framework. Like us, they require functions to be monotonic in order to guarantee properties of their analysis like termination and solution uniqueness.

One of the most interesting areas of their work for us is their analysis of conditional expressions. They present two types of analysis for conditionals: the first is a general but more conservative analysis, called simply **if** analysis, and the other is a more specialized

but less conservative analysis called **ifc**. (This name is given since the specialization requires some subexpressions to be fixed constants). The general analysis proves useful for some cases, but too conservative for others. The specialized **ifc** analysis proves to be not conservative enough, in that it is unsound. That is, it can judge non-constant expressions as constant and non-monotonic expressions as monotonic.

General Conditional Analysis

In the existing work, the language of the expressions being analyzed was restricted to have only a single type of Boolean expression: $e_1 \leq e_2$. This guarantees that all conditional expressions take the following form:

$$(e_1 \leq e_2)?e_3 : e_4$$

This does not restrict the expressiveness of the analysis severely, because $\geq$, $<$, and $>$ can be easily constructed from this arrangement by swapping e_1 and e_2 or e_3 and e_4. Under this assumption, the first and most general analysis of the monotonicity of conditional expressions is presented:

For brevity, we will use 0 as shorthand for Constant, + as shorthand for Monotonic, and − as shorthand for Antimonotonic.

me_1	me_2	me_3	me_4	Φ	Overall
+	−	+	+	$e_3(\top) \leq e_4(\bot)$	+
+	−	−	−	$e_3(\top) \geq e_4(\bot)$	−
−	+	+	+	$e_3(\bot) \geq e_4(\top)$	+
−	+	−	−	$e_3(\bot) \leq e_4(\top)$	−
0	0	α	α	none	α

We read this table as follows: if the less than side of the predicate, greater than side of the predicate, then branch of the conditional, and else branch of the conditional can all be analyzed to have monotonicity concepts in the first four columns, and additionally the condition Φ is true, then the overall conditional expression can be inferred to have the monotonicity concept given in the final column. The monotonicity lattice follows the "is a" relationship, so proving a property lower in the lattice implies the properties higher in the lattice. In particular, constant functions are both monotonic and antimonotonic. The final row of the table may have α replaced with any monotonicity concept, as long as it is the same in all three places. We could have replaced this row with three distinct rows for each of 0, +, and −. In all cases where none of these rules apply, the overall expression must be conservatively concluded to be **Nonmonotonic**. This means that we could not prove any assertion about the expression, and that it could potentially be non-monotonic.

To get the intuition for how this approach works, let us consider the first row in the table (as the next 3 rows are simply symmetries of the first row). In order to prove that an expression is monotonic, one needs to prove that the then branch of the expression is

monotonic, the else branch is monotonic, and that any transition from one branch to the other is monotonic. That transition occurs whenever the truth value of $e_1 \leq e_2$ changes. If e_1 is monotonic and e_2 is antimonotonic, then as the input increases, the truth value of $e_1 \leq e_2$ can only change from being true to being false. If the conditional predicate always moves from being true to being false, and any value that can be produced by e_3 is less than or equal to any value that can be produced by e_4, then any transition will be monotonic. Since e_3 and e_4 are both monotonic, checking that $e_3(\top) \leq e_4(\bot)$ is sufficient to conclude that any value produced by e_3 will be less than or equal to any value produced by e_4. Rows 2-4 are symmetries of this case.

In regards to our running dimensionality example, the following (monotonic) constraint can be successfully resolved to **Monotonic** using this general if analysis:

```
(x <= Unknown) ? Unknown :
(x <= Velocity) ? Position :
Conflict
```

Constant Conditional Analysis (ifc Analysis)

There are many cases where an if statement does not satisfy the strict condition Φ given in the general if analysis, but is monotonic nonetheless. In order to address these cases, Murawski and Yi introduce a more exact analysis for conditional statements a specific form. They introduce the notion the **ifc** statement, which is a conditional in which the predicate is of the form $x <= c$, where x is a variable and c is a constant. Knowing that the conditional is of this form allows an analysis that makes the condition Φ more exact, referring to the constant c in the checks.

The analysis of **ifc** statements as given in [35] is reproduced below:

me_3	me_4	Φ	Overall
$+$	$+$	$\forall d \in \text{cover}(c):\ e_3(c) \leq e_4(d)$	$+$
$-$	$-$	$\forall d \in \text{cover}(c):\ e_3(c) \geq e_4(d)$	$-$
0	0	irrelevant	0

They aim to make a sound extension to the generalized rule, but their extension is unsound, in that it allows nonmonotonic expressions to be classified as monotonic.

There are two problems with this analysis. The first is that it allows non-constant expressions to be classified as constant by not checking that the two branches are equal. As a result, any conditional with constant branches can be inferred to be constant. For example, functions such as the following, which are clearly not constant, can be analyzed as constant:

```
(x <= Dimensionless) ? Time : Conflict
```

In fact, using this analysis, even our original integrator constraint is inferred to be constant. This is clearly an oversight, and the problematic row can be fixed by adding a check that both sides of the conditional are equal or by removing the final row altogether.

There is also a second, more subtle problem with this analysis, however. Even if we remove the problematic row, the remaining analysis is not sound. This is because of the way that it checks the transition of the conditional. In the monotonic case, for example, given a predicate of the form $x <= c$, it checks the transition between c and elements above c. This assumes that the only way that the conditional could transition from being true to being false would be for x to transition from a value $x \leq c$ to a value $x > c$, which is a valid assumption for a totally ordered set. Since we are dealing with partial orders, however, $x > c$ is not the only way that $x <= c$ can be false. x and c can also be incomparable. Thus, the original analysis ignores the cases where $x <= c$ becomes false by x becoming incomparable to c.

For example, let us consider the following function:

```
(x <= Acceleration) ? Velocity :
(x <= Velocity) ? Position :
Conflict
```

Even though it does meet all the requirements for being labeled monotonic by the **ifc** rule from [35], it is not monotonic. To demonstrate this fact, consider the two inputs $x = Unkown$ and $y = Velocity$. Since this expression evaluates to $f(x) = Velocity$ and $f(y) = Position$, we have a case where $x \leq y$ but $f(x) \not\leq f(y)$. This directly contradicts the definition of monotonicity, and proves that the analysis is unsound.

10.6 Revised Analysis

A Sound ifc Rule

The first step to revising the analysis is to make the **ifc** rule sound. This can be achieved by expanding the border cases that are checked as part of Φ before declaring the expression to be monotonic. The revised analysis must include any ordered pairs of points that could be on the border between the predicate being true and false.

While it is true that any transition from x being less than or equal to c to x being greater than c will take place at c and the cover set of c, general lattices are partial orders, not total orders. This means that $x \leq c$ being false does not necessarily mean that $x > c$ is true. There could also be concepts that are incomparable to c, meaning that $x \leq c$ and $x > c$ are both false. An ordered pair of concepts where one is less than c and another is incomparable to c will also be part of the border between cases where the conditional predicate is true and the cases where it is false.

In order to check that all of these transitions maintain monotonicity, we need explicitly check for them by including them in our rule Φ:

me_3	me_4	Φ	Overall
$+$	$+$	$\forall b \in \text{down}(c), \forall d \in \text{cover}(b) \setminus \text{down}(c) : \ e_3(b) \leq e_4(d)$	$+$
$-$	$-$	$\forall b \in \text{down}(c), \forall d \in \text{cover}(b) \setminus \text{down}(c) : \ e_3(b) \geq e_4(d)$	$-$

This corrected analysis is technically able to infer the monotonicity of additional cases for which the general monotonicity analysis is too coarse. For example, the following example can be correctly inferred to be monotonic:

(x <= Unknown) ? x : Time

When evaluated on our suite of monotonic constraints, however, this corrected analysis does not by itself allow any additional constraints to be evaluated as monotonic.

More Expressive Conditional Guards

In order to support a wider variety of expressions, we have extended the monotonicity analysis to support more relations and Boolean operators within the conditional predicates. In order to classify these expressions in a useful way, however we need to define an order on $true$ and $false$. We have arbitrarily chosen $true < false$. This means that we can now talk of the entire conditional predicate being monotonic or antimonotonic, rather than just the expressions on either side of the $\leq$ sign. As a side effect of this choice, we can express the general conditional analysis in a more general way as:

mp	me_3	me_4	Φ	Overall
$+$	$+$	$+$	$e_3(\top) \leq e_4(\bot)$	$+$
$+$	$-$	$-$	$e_3(\top) \geq e_4(\bot)$	$-$
$-$	$+$	$+$	$e_3(\bot) \geq e_4(\top)$	$+$
$-$	$-$	$-$	$e_3(\bot) \leq e_4(\top)$	$-$
0	α	α	none	α

Here, mp is the monotonicity of the entire conditional predicate, over the lattice $true < false$.

This gives us the following table for Boolean relations, of the form e_1 relation e_2:

Relation	me_1	me_2	Overall
$\leq$ or $<$	$+$	$-$	$+$
$\leq$ or $<$	$-$	$+$	$-$
$\geq$ or $>$	$+$	$-$	$-$
$\geq$ or $>$	$-$	$+$	$+$
Any	0	0	0

We can also support Boolean operators including conjunctions ($\wedge$), disjunctions ($\vee$), and negations ($\neg$):

e_1 op e_2 table ($\wedge$ or $\vee$)

me_1	me_2	Overall
$+$	$+$	$+$
$-$	$-$	$-$
0	0	0

$\neg e_1$ table

me_1	Overall
+	−
−	+
0	0

Incorporating these extensions allows our monotonicity analysis to work on a richer set of expressions, but it does not make the analysis powerful enough to check the monotonic constraint for the integrator in Expression 10.1. In order to do that, we will need a fundamentally changed analysis.

Non-compositional ifc Monotonicity Analysis

One of the key reasons that the previous analyses are not exact enough for our needs is that most of the constraints that we are interested in build a monotonic expression out of non-monotonic subexpressions. This is a problem for all of the previously presented approaches, because they all rely on the assumption that e_3 and e_4 are monotonic in order to prove the monotonicity of $p?e_3 : e_4$.

Yet there are many cases in which the overall expression can be monotonic without e_3 and e_4 being monotonic. An expression e is judged to be non-monotonic if there exists any pair of inputs x and y, with $x \leq y$ but $e(x) \not\leq e(y)$. Even if the expression is monotonic "everywhere else," it is still a non-monotonic expression. If we combine just the "monotonic parts" of non-monotonic expressions, we can create a resulting composition that is overall monotonic. Let us formalize this notion.

For a given non-monotonic expression, let us consider the set of all of the counterexamples to monotonicity (and antimonotonicity). This is the set of all ordered pairs of values x and y, $x \leq y$, such that the expression evaluated at x and y are not ordered in the correct order required for monotonicity (or antimonotonicity).

$$\mathrm{MC}(e) = \{(x, y) \mid x \leq y \wedge e(x) \not\leq e(y)\}$$
$$\mathrm{AC}(e) = \{(x, y) \mid x \leq y \wedge e(x) \not\geq e(y)\}$$

For a given expression, if $\mathrm{MC}(e) = \emptyset$, then we can conclude it is monotonic, and if $\mathrm{AC}(e) = \emptyset$, then we can conclude that it is antimonotonic. Since the expressions that we deal with are deterministic, clearly any element $(a, b) \in \mathrm{MC}(e)$ must have $a < b$. For our purposes, we can use an alternate form that only considers counterexamples where b covers a:

$$\mathrm{MC}'(e) = \{(x, y) \mid x <\cdot\, y \wedge e(x) \not\leq e(y)\}$$
$$\mathrm{AC}'(e) = \{(x, y) \mid x <\cdot\, y \wedge e(x) \not\geq e(y)\}$$

The use of $<\cdot$ rather than $\leq$ or $<$ allows us to make the set as small as possible without losing information. This is permissible because if there exists a counterexample to monotonicity between elements that are not immediate neighbors in the lattice order, there must also

exist counterexamples to monotonicity that are immediate neighbors in the lattice order. Formally, if $a \leq b \leq c$ and $f(a) \not\leq f(c)$, then it must also be the case that either $f(a) \not\leq f(b)$ or $f(b) \not\leq f(c)$.

If there were a way to efficiently calculate these sets, then one could use that information to build a check of the monotonicity of expressions that are made up of even of non-monotonic subexpressions. In order to do so, when checking an expression for monotonicity that is not made up of monotonic subexpressions, we should make sure to check for monotonicity on the counterexamples to monotonicity of the subexpressions in addition to any other places that we would normally need to check.

Given an expression in the same form of the **ifc** rule, namely $(x <= c)?e_3 : e_4$, then we can formulate our non-compositional monotonicity analysis as follows:

me_3	me_4	Φ	Overall
$\top$	$\top$	$\forall (b,d) \in \mathrm{MC}'(e_3) \cup \mathrm{MC}'(e_4) \cup S_{border},\ e(b) \leq e(d)$	$+$
$\top$	$\top$	$\forall (b,d) \in \mathrm{AC}'(e_3) \cup \mathrm{AC}'(e_4) \cup S_{border},\ e(b) \geq e(d)$	$-$

where S_{border} is defined as follows:

$$\{(b,d) \mid b \in \mathrm{down}(c) \wedge d \in \mathrm{cover}(b) \setminus \mathrm{down}(c)\}$$

Note that S_{border} defines exactly the same set over which the revised **ifc** rule is defined. This is the border between $x \leq c$ evaluating to true and false.

Calculating Counterexamples

While this new rule is sound, it does not help us with finding the sets of counterexamples. If one simply searched naively for counterexamples in every non-monotonic expression, the new problem created would be just as much work as our original monotonicity analysis problem.

To avoid this, we introduce new concepts that allow us to keep track of counterexamples of non-monotonic expressions. We will use value parametrized concepts (see Section 9.1) called **NonMonotonic** and **NonAntimonotonic**, represented as $\widetilde{+}(S)$ and $\widetilde{-}(S)$, respectively, as shown in the revised concept lattice in Figure 10.3. These represent expressions that are not monotonic (or not antimonotonic), but have a finite set of counterexamples to monotonicity (or antimonotonicity). The parametrized value S is a set that contains those counterexamples. A counterexample to monotonicity is a pair of elements (a, b) such that $a \leq b$, but $e(a) \not\leq e(b)$. Similarly, a counterexample to antimonotonicity is a pair (a, b) such that $a \leq b$, but $e(a) \not\geq e(b)$.

In addition, we require that the counterexample set be complete. This means that expressions resolved to $\widetilde{+}(S)$ must be monotonic everywhere other than S. Formally, e resolving to $\widetilde{+}(S)$ means that $\forall a, b$ with $a <\cdot\, b$:

$$(a,b) \notin S \implies e(a) \leq e(b).$$

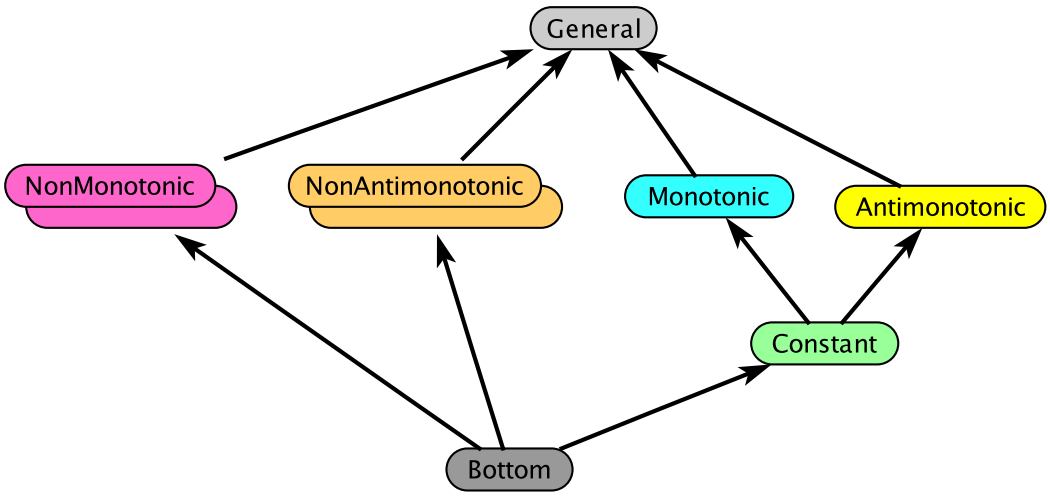


Figure 10.3: The monotonicity lattice can be extended to keep track concepts that are not monotonic.

This restriction means that if the expression e can be proved to have monotonicity of $\widetilde{+}(S)$, then $\mathrm{MC}'(e) = S$, if e can be proved to have monotonicity of $\widetilde{-}(S)$, then $\mathrm{AC}'(e) = S$.

Note the use of $<\cdot$ rather than $<$. This allows us to make the set S as small as possible without losing information. This is because if there exists a counterexample to monotonicity between elements that are not immediate neighbors in the lattice order, there must also exist counterexamples to monotonicity that are immediate neighbors in the lattice order. i.e. If $a \leq b \leq c$ and $f(a) \not\leq f(c)$, then it must also be the case that either $f(a) \not\leq f(b)$ or $f(b) \not\leq f(c)$.

In order to reduce the size of these sets, we also include the restriction that in each pair (a, b) b must cover a. This is because any non-monotonic function must have a covering pair that is a counterexample to monotonicity, so this restriction does not affect which non-monotonic functions we are able to detect.

These non-monotonic concepts have the property that they are monotonic (or antimonotonic) at all covering pairs not in the counterexample set. This means that as before, we only need to check the counterexamples from each of the subexpressions, in addition to the pairs from the border between the conditional being true and being false.

me_3	me_4	Φ	Overall
$\widetilde{+}(S_1)$	$\widetilde{+}(S_2)$	$\forall (b,d) \in S_1 \cup S_2 \cup S_{border}, e(b) \leq e(d)$	$+$
$\widetilde{-}(S_1)$	$\widetilde{-}(S_2)$	$\forall (b,d) \in S_1 \cup S_2 \cup S_{border}, e(b) \geq e(d)$	$-$

where S_{border} is defined as follows:

$$\{(b,d) \mid b \in \mathrm{down}(x) \wedge d \in \mathrm{cover}(b)\}$$

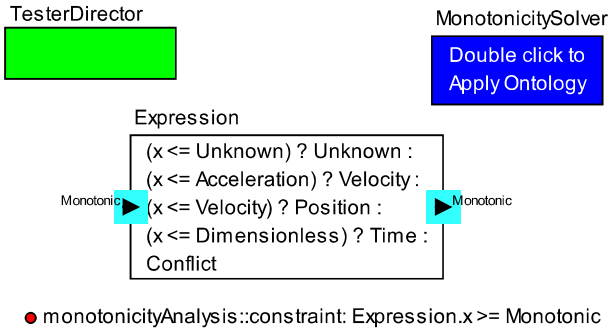


Figure 10.4: Final algorithm correctly inferring the monotonicity of the integrator constraint expression

Since these non-monotonic cases tend to arise in conditionals, this is the most logical place to begin to keep track of these non-monotonic concepts in the first place. Since we already must explicitly check the monotonicity of a fixed set of points in the course of checking the Φ condition, we can get the set of counterexamples without doing any extra work. We can simply keep track of all of the pairs that do not meet the Φ condition. When creating the non-monotonic value-parametrized concept for the expression, this set of pairs that did not meet the Φ condition gives exactly the set of counterexamples.

me_3	me_4	$\neg\Phi$	Overall
$\widetilde{+}(S_1)$	$\widetilde{+}(S_2)$	$\forall(b,d) \in S_{fail}, e(b) \not\leq e(d)$	$\widetilde{+}(S_{fail})$
$\widetilde{-}(S_1)$	$\widetilde{-}(S_2)$	$\forall(b,d) \in S_{fail}, e(b) \not\geq e(d)$	$\widetilde{-}(S_{fail})$

Here, $S_{fail} \subseteq S_1 \cup S_2 \cup S_{border}$ is the maximal such subset that fails to meet the Φ condition, and S_{border} is defined as before. Finding this maximal set is straightforward if it is constructed in the process of checking Φ from the previous table.

10.7 Results

Integrator Example

This is finally powerful enough to analyze our running example correctly:

```
(x <= Unknown) ? Unknown :
(x <= Acceleration) ? Velocity :
(x <= Velocity) ? Position :
(x <= Dimensionless) ? Time :
Conflict
```

This expression can now be correctly evaluated as monotonic, even though the second subexpression is not monotonic. The second expression is:

```
(x <= Acceleration) ? Velocity :
(x <= Velocity) ? Position :
(x <= Dimensionless) ? Time :
Conflict
```

We can prove this expression to be nonmonotonic by producing a counterexample to monotonicity. This particular expression has two counterexamples: $(Unknown, Velocity)$ and $(Unknown, Dimensionless)$.

By keeping track of these counterexamples for each of the subexpressions, our analysis is able to keep track of the counterexamples that need to be checked at the next level of the hierarchy.

Verification of Monotonicity of Expressions

In order to test the effectiveness of our monotonicity analysis, we will run our analysis on the constraint expressions published in our first ontology analysis paper [27], which contains a dimension system analysis similar to that presented in Chapter 7.

In order to be able to analyze the constraints from that work, we first translate them into the form used for our analysis. This requires all conditional statements to have the form $variable \leq constant$ in their conditions. An overview of these translations is given in Table 10.1 with the constraints that are too simple to be interesting (like least upper bounds, single variables, etc.) omitted.

The first thing that one notices about these translations is that equality tests have been changed to (equivalent) inequality tests, as is required by the analysis. In addition, the equations that were over multiple variables have been rewritten as being over a single variable whose domain is a product lattice. Other translations are also relatively straightforward, and most are mechanical enough that they could be automated. Theses include things like replacing

$$(b_1||b_2)?e_1 : e_2$$

with

$$(b_1)?e_1 : (b_2)?e_1 : e_2$$

since the analysis algorithm requires all of the conditions of conditionals to be pure relational nodes.

The most demanding constraint fragment to rewrite was of the form

$$(x == y)?e_1 : e_2 \ .$$

This constraint is not difficult because of syntactic quirks of our analysis. Rather, it is difficult for us to analyze the monotonicity of because it deals with many different parts of the lattice at one time. In fact, none of the lattice values for which the conditional statement is true are even contiguous. That is, none of the values of x and y for which the condition is true cover each other in the product lattice. For this reason, we have elected to simply

Constraint	Domain	Definition
Integrator (forward)	D	$\begin{cases} \bot, & \text{if } x \leq \bot \\ Velocity, & \text{if } x \leq Position \\ Acceleration, & \text{if } x \leq Velocity \\ Dimensionless, & \text{if } x \leq Time \\ Conflict, & \text{otherwise} \end{cases}$
Division	$D \times D$	$\begin{cases} \bot, & \text{if } x \leq \bot \times Conflict \\ \bot, & \text{if } x \leq Conflict \times \bot \\ Acceleration, & \text{if } x \leq Velocity \times Time \\ Velocity, & \text{if } x \leq Position \times Time \\ Time, & \text{if } x \leq Position \times Velocity \\ Time, & \text{if } x \leq Velocity \times Acceleration \\ Dimensionless, & \text{if } x \leq Acceleration \times Acceleration \\ Dimensionless, & \text{if } x \leq Velocity \times Velocity \\ Dimensionless, & \text{if } x \leq Position \times Position \\ Dimensionless, & \text{if } x \leq Time \times Time \\ Dimensionless, & \text{if } x \leq Dimensionless \times Dimensionless \\ x.projectLeft, & \text{if } x \leq Conflict \times Dimensionless \\ Conflict, & \text{if } x < Conflict \times Conflict \\ Dimensionless, & \text{otherwise} \end{cases}$
Multiplication	$D \times D$	$\begin{cases} \bot, & \text{if } x \leq \bot \times Conflict \\ \bot, & \text{if } x \leq Conflict \times \bot \\ Velocity, & \text{if } x \leq Acceleration \times Time \\ Velocity, & \text{if } x \leq Velocity \times Dimensionless \\ Position, & \text{if } x \leq Velocity \times Time \\ Position, & \text{if } x \leq Position \times Dimensionless \\ x.projectLeft, & \text{if } x \leq Conflict \times Dimensionless \\ x.projectRight, & \text{if } x \leq Dimensionless \times Conflict \\ Conflict, & \text{otherwise} \end{cases}$

Table 10.1: Constraints from [27] for some representative actors

break this conditional up into each of its constituent values over the lattice in question and deal with them individually.

The results from analyzing the multiplication and division constraints can be seen in Figure 10.5, which contains constraints for both the multiplication and division of dimension concepts. Since multiplication and division are both binary operations, these concept functions both take two inputs. To represent this, the expressions are defined over a product lattice of the original dimension lattice. The definition of these expressions also makes use of left and right projections, both of which are monotonic operations. The monotonicity analysis is able to prove the monotonicity of the multiplication constraint, but not the division constraint. This is due not to the conservatism of the analysis, but due to the fact that the division constraint given in [27] is actually not monotonic. The problematic portion is given at the end, where it deals with dimensionless quantities. One line of the original constraint is:

$$f_d(x, y) = Dimensionless, \qquad \text{if } x = y \ .$$

The problem with this constraint arises when x and y are both $Conflict$. This is the highest possible input in the product order, but f_d returns $Dimensionless$. Because other inputs to f_d do resolve to $Conflict$, and $Dimensionless < Conflict$, this means that the constraint is not monotonic.

10.8 Conclusion

This chapter has presented a static analysis algorithm for determining the monotonicity of expressions in our Ptolemy II modeling tool. The analysis is inspired by the static monotonicity analysis of [35], but it corrects unsound rules found in that work. In addition, it extends the analysis to make it non-compositional the first case we have seen to use this type of approach. This allows us to infer the monotonicity of expressions that are made up of non-monotonic subexpressions. We then use this new monotonicity analysis to prove the monotonicity of constraints from [27], as well as to find an erroneous non-monotonic constraint from that work.

The monotonicity analysis presented here has a symbiotic relationship with the rest of the ontology analysis framework. On the one hand, knowledge that the constraints are monotonic is required to guarantee that the algorithm our framework's solver uses will terminate. On the other hand, the monotonicity check itself is implemented as an ontology analysis.

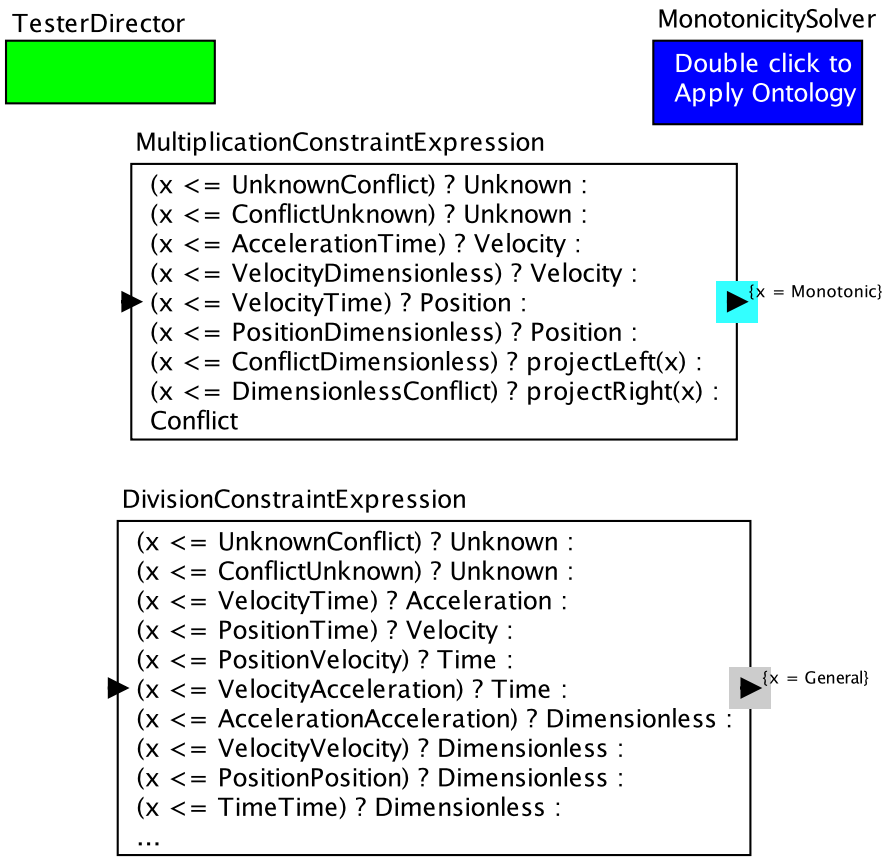


Figure 10.5: Running the monotonicity analysis on the constraints from Table 10.1

Part IV

Conclusions

Chapter 11

Conclusion

This thesis demonstrates a general technique for adding generic analyses to programs that can be applied to their abstract syntax, such as control flow graphs, etc. It comprises of user-defined domain of information represented as a lattice, as well as constraints defined over that structure. This allows us to leverage efficient fixed-point algorithms that allow us to efficiently infer the least solutions to these constraints throughout the static structure of the program or model being analyzed. Using these types of analyses allow users to verify that their programs are free of certain design errors, as well as provide documentation about their programs in a domain specific language.

We have also presented a host of other techniques to aid in this process, including an error minimization algorithm adapted from [52] that aids in quickly finding the source of errors in models, a method of using these same methods and algorithms to check for the monotonicity of expressions, and a special infrastructure for dealing with ontologies that are infinite in size in general, including those that are infinite in one of two patterns: infinite flat lattice ontologies, and infinite recursive ontologies. We have also developed syntax for concisely representing ontologies that deal with units, a large and useful domain for ontology analysis.

We have developed an implementation of these techniques as a part of Ptolemy II, and all of the code and infrastructure is included under a BSD license in the upcoming Ptolemy 9.0 release. An abridged version of this document concerning only the use of ontologies as an end user of Ptolemy is included as a chapter in the Ptolemy Book, available from `http://ptolemy.berkeley.edu/ptolemyII/designdoc.htm`.

Bibliography

[1] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. "Kepler: an extensible system for design and execution of scientific workflows". In: *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*. June 2004, pp. 423 –424. DOI: 10.1109/SSDM.2004.1311241.

[2] Scott W. Amber. *UML 2 Class Diagrams*. [Accessed Jun. 25, 2012]. 2003–2010. URL: http://www.agilemodeling.com/artifacts/classDiagram.htm.

[3] Madhukar Anand, Insup Lee, George Pappas, and Oleg Sokolsky. "Unit & Dynamic Typing in Hybrid Systems Modeling with CHARON". In: *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE*. Oct. 2006, pp. 56–61. DOI: 10.1109/CACSD-CCA-ISIC.2006.4776624.

[4] Peter Aronsson and David Broman. "Extendable Physical Unit Checking with Understandable Error Reporting". In: *Proceedings of the 7th International Modelica Conference*. Como, Italy, 2009, pp. 890–897.

[5] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. "Metropolis: an integrated electronic system design environment". In: *Computer* 36.4 (Apr. 2003), pp. 45 –52. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1193228.

[6] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. "Continuity analysis of programs". In: *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '10. Madrid, Spain: ACM, 2010, pp. 57–70. ISBN: 978-1-60558-479-9. DOI: http://doi.acm.org/10.1145/1706299.1706308. URL: http://doi.acm.org/10.1145/1706299.1706308.

[7] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula. "Dependent Types for Low-Level Programming". In: *Programming Languages and Systems*. Ed. by Rocco De Nicola. Vol. 4421. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007, pp. 520–535. ISBN: 978-3-540-71314-2. URL: http://dx.doi.org/10.1007/978-3-540-71316-6_35.

[8] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL '77. Los Angeles, California: ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973. URL: http://dx.doi.org/10.1145/512950.512973.

[9] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL '77. Los Angeles, California: ACM, 1977, pp. 238–252. DOI: http://doi.acm.org/10.1145/512950.512973. URL: http://doi.acm.org/10.1145/512950.512973.

[10] Patricia Derler, Edward A. Lee, and Alberto L. Sangiovanni-Vincentelli. *Addressing Modeling Challenges in Cyber-Physical Systems*. Tech. rep. UCB/EECS-2011-17. EECS Department, University of California, Berkeley, Mar. 2011. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-17.html.

[11] 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". In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 127 –144. ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.805829.

[12] Huáscar Espinoza, Hubert Dubois, Sébastien Gérard, Julio Medina, Dorina Petriu, and Murray Woodside. "Annotating UML Models with Non-functional Properties for Quantitative Analysis". In: *Satellite Events at the MoDELS 2005 Conference*. Ed. by Jean-Michel Bruel. Vol. 3844. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 79–90. URL: http://dx.doi.org/10.1007/11663430_9.

[13] W3C Owl Working Group. "OWL 2 Web Ontology Language Document Overview". In: *W3C Recommendation* (Oct. 2009), pp. 1–12. URL: http://www.w3.org/TR/2009/REC-owl2-overview-20091027/.

[14] Thomas R. Gruber. "A Translation Approach to Portable Ontology Specifications". In: *Knowledge Acquisition* 5.2 (Apr. 1993), pp. 199–220.

[15] Cécile Hardebolle and Frédéric Boulanger. "ModHel'X: A Component-Oriented Approach to Multi-Formalism Modeling". In: *Models in Software Engineering*. Ed. by Holger Giese. Vol. 5002. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 247–258. URL: http://dx.doi.org/10.1007/978-3-540-69073-3_26.

[16] Paul N. Hilfinger. "An Ada package for dimensional analysis". In: *ACM Trans. Program. Lang. Syst.* 10 (2 Apr. 1988), pp. 189–203. ISSN: 0164-0925. DOI: http://doi.acm.org/10.1145/42190.42346. URL: http://doi.acm.org/10.1145/42190.42346.

[17] Stephen C. Johnson. "Lint, a C program checker". In: *Computer Science Technical Report 65*. Bell Laboratories, July 1978.

[18] Michael Karr and David B. Loveman III. "Incorporation of units into programming languages". In: *Commun. ACM* 21 (5 May 1978), pp. 385–391. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/359488.359501. URL: http://doi.acm.org/10.1145/359488.359501.

[19] Andrew J. Kennedy. "Relational parametricity and units of measure". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '97. Paris, France: ACM, 1997, pp. 442–455. ISBN: 0-89791-853-3. DOI: 10.1145/263699.263761. URL: http://doi.acm.org/10.1145/263699.263761.

[20] Gary A. Kildall. "A unified approach to global program optimization". In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL '73. Boston, Massachusetts: ACM, 1973, pp. 194–206. DOI: 10.1145/512927.512945. URL: http://doi.acm.org/10.1145/512927.512945.

[21] Holger Knublauch, Ray Fergerson, Natalya Noy, and Mark Musen. "The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications". In: *The Semantic Web ISWC 2004*. Ed. by Sheila McIlraith, Dimitris Plexousakis, and Frank van Harmelen. Vol. 3298. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, pp. 229–243. ISBN: 978-3-540-23798-3. URL: http://dx.doi.org/10.1007/978-3-540-30475-3_17.

[22] Don Lawson. "Engineering disasters : lessons to be learned". In: 1 Birdcage Walk, London, UK: Professional Engineering Publishing Limited, 2005, pp. 221–229. ISBN: 0-7918-0230-2.

[23] Gérard Le Lann. "An analysis of the Ariane 5 flight 501 failure - a system engineering perspective". In: *Proceedings of the 1997 international conference on Engineering of computer-based systems*. ECBS'97. Monterey, California: IEEE Computer Society, 1997, pp. 339–346. ISBN: 0-8186-7889-5. URL: http://dl.acm.org/citation.cfm?id=1880177.1880238.

[24] Edward A. Lee, Thomas Huining Feng, Xiaojun Liu, Steve Neuendorffer, Neil Smyth, and Yuhong Xiong. "Expressions". In: Heterogeneous Concurrent Modeling and Design in Java. http://ptolemy.berkeley.edu/ptolemyII/ptIIlatest/ptII/doc/expressions.pdf. EECS Department, University of California, Berkeley, June 2010. Chap. 3.

[25] Edward A. Lee and Yuhong Xiong. "A Behavioral Type System and Its Application in Ptolemy II". In: *Formal Aspects of Computing Journal* 16.3 (2004), pp. 210 –237.

[26] 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*. (recipient of the MODELS 2009 Distinguished Paper Award). ACM/IEEE. Oct. 2009, pp. 393–407. URL: http://chess.eecs.berkeley.edu/pubs/611.html.

[27] 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*. ACM/IEEE. Oct. 2009, pp. 393–407. URL: http://chess.eecs.berkeley.edu/pubs/611.html.

[28] Ben Lickly, Charles Shelton, Elizabeth Latronico, and Edward A. Lee. "A Practical Ontology Framework for Static Model Analysis". In: *EMSOFT '11: Proceedings of the Ninth International Conference on Embedded Software*. ACM. Oct. 2011, pp. 23–32. URL: http://chess.eecs.berkeley.edu/pubs/862.html.

[29] 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*. 2006, pp. 13–17.

[30] T. Maehne and A. Vachoux. "Supporting dimensional analysis in SystemC-AMS". In: *Behavioral Modeling and Simulation Workshop, 2009. BMAS 2009. IEEE*. Sept. 2009, pp. 108 –113. DOI: 10.1109/BMAS.2009.5338878.

[31] Frank Manola, Eric Miller, and Brian McBride. "RDF Primer". In: *W3C Recommendation* (Feb. 2004). URL: http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[32] Ghassan Misherghi and Zhendong Su. "HDD: hierarchical delta debugging". In: *ICSE '06: Proceedings of the 28th international conference on Software engineering*. Shanghai, China: ACM, 2006, pp. 142–151. ISBN: 1-59593-375-1. DOI: http://doi.acm.org/10.1145/1134285.1134307.

[33] Boris Motik, Bernardo Cuenca Grau, and Ulrike Sattler. "Structured Objects in OWL: Representation and Reasoning". In: *Proc. of the 17th Int. World Wide Web Conference (WWW 2008)*. Ed. by Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang. Beijing, China: ACM Press, Apr. 2008, pp. 555–564.

[34] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997. ISBN: 9781558603202.

[35] Andrzej Murawski and Kwangkeun Yi. "Static Monotonicity Analysis for Lambda-definable Functions over Lattices". In: *Third International Workshop on Verification, Model Checking and Abstract Interpretation*. Lecture Notes on Computer Science. Venice, Italy, Jan. 2002.

[36] James Oberg. "Why the Mars probe went off course". In: *IEEE Spectr.* 36 (12 Dec. 1999), pp. 34–39. ISSN: 0018-9235. DOI: 10.1109/6.809121. URL: http://portal.acm.org/citation.cfm?id=337684.337688.

[37] Wolfgang Pree and Josef Templ. "Modeling with the Timing Definition Language (TDL)". In: *Model-Driven Development of Reliable Automotive Services*. Ed. by Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. Vol. 4922. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 133–144. URL: http://dx.doi.org/10.1007/978-3-540-70930-5_9.

[38] Kepler Project. *Getting Started with Kepler Tagging 2.3*. Aug. 2011. URL: https://code.kepler-project.org/code/kepler/trunk/modules/tagging/docs/tagging.pdf.

[39] Jakob Rehof and Torben Æ. Mogensen. "Tractable Constraints in Finite Semilattices". In: *SAS '96: Proceedings of the Third International Symposium on Static Analysis*. London, UK: Springer-Verlag, 1996, pp. 285–300. ISBN: 3-540-61739-6.

[40] Jakob Rehof and Torben Æ. Mogensen. "Tractable Constraints in Finite Semilattices". In: *SAS '96: Proceedings of the Third International Symposium on Static Analysis*. London, UK: Springer-Verlag, 1996, pp. 285–300. ISBN: 3-540-61739-6.

[41] Andreas Griesmayer Roderick, Roderick Bloem, and Byron Cook. "Repair of Boolean Programs with an Application to C". In: *In 18th Conference on Computer Aided Verification (CAV'06)*. Springer, 2006.

[42] Diana Sánchez, José Cavero, and Esperanza Martínez. "The Road Toward Ontologies". In: *Ontologies: A Handbook of Principles, Concepts and Applications in Information Systems*. Ed. by Raj Sharman, Rajiv Kishore, and Ram Ramesh. Vol. 14. Integrated Series in Information Systems. Springer US, 2007, pp. 3–20. ISBN: 978-0-387-37022-4. DOI: 10.1007/978-0-387-37022-4_1. URL: http://dx.doi.org/10.1007/978-0-387-37022-4_1.

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

[44] Rupert Schlick, Wolfgang Herzner, and Thierry Le Sergent. "Checking SCADE Models for Correct Usage of Physical Units". In: *Computer Safety, Reliability, and Security*. Ed. by Janusz Górski. Vol. 4166. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 358–371.

[45] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. "Pellet: A practical OWL-DL reasoner". In: *Web Semant.* 5.2 (June 2007), pp. 51–53. ISSN: 1570-8268. DOI: 10.1016/j.websem.2007.03.004. URL: http://dx.doi.org/10.1016/j.websem.2007.03.004.

[46] Frank Tip. "A Survey of Program Slicing Techniques". In: *Journal of Programming Languages* 3.CS-R9438 (1994), pp. 1–58.

[47] 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". In: *SIGBED Rev.* 2 (3 July 2005), pp. 36–39. ISSN: 1551-3688. DOI: http://doi.acm.org/10.1145/1121802.1121810. URL: http://doi.acm.org/10.1145/1121802.1121810.

[48] Dmitry Tsarkov and Ian Horrocks. "FaCT++ description logic reasoner: system description". In: *Proceedings of the Third international joint conference on Automated Reasoning*. IJCAR'06. Seattle, WA: Springer-Verlag, 2006, pp. 292–297. ISBN: 3-540-37187-7, 978-3-540-37187-8. DOI: 10.1007/11814771_26. URL: http://dx.doi.org/10.1007/11814771_26.

[49] Tim Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 0123742749, 9780123742742.

[50] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. "Automatically finding patches using genetic programming". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 364–374. DOI: http://dx.doi.org/10.1109/ICSE.2009.5070536.

[51] Hongwei Xi. "Dependent Types for Program Termination Verification". In: *Higher-Order and Symbolic Computation* 15.1 (Mar. 2002), pp. 91–131. ISSN: 1388-3690. DOI: 10.1023/A:1019916231463. URL: http://dx.doi.org/10.1023/A:1019916231463.

[52] Andreas Zeller, Ieee Computer Society, and Ralf Hildebrandt. "Simplifying and Isolating Failure-Inducing Input". In: *IEEE Transactions on Software Engineering* 28 (2002), p. 2002.

[53] Yonggang Zhang. "An Ontology-based Program Comprehension Model". PhD thesis. Montreal, Quebec, Canada: Concordia University, Sept. 2007.

[54] Yonggang Zhang, Juergen Rilling, and Volker Haarslev. "An Ontology-Based Approach to Software Comprehension - Reasoning about Security Concerns". In: *Proceedings of the 30th Annual International Computer Software and Applications Conference*. Vol. 01. COMPSAC '06. Chicago, Illinois, USA: IEEE Computer Society, 2006, pp. 333–342. ISBN: 0-7695-2655-1. DOI: 10.1109/COMPSAC.2006.27. URL: http://dx.doi.org/10.1109/COMPSAC.2006.27.

[55] Yang Zhao, Yuhong Xiong, Edward A. Lee, Xiaojun Liu, and Lizhi C. Zhong. "The design and application of structured types in Ptolemy II". In: *Int. J. Intell. Syst.* 25 (2 Feb. 2010), pp. 118–136. ISSN: 0884-8173. DOI: http://dx.doi.org/10.1002/int.v25:2. URL: http://dx.doi.org/10.1002/int.v25:2.

[56] Hong Zhou, Feng Chen, and Hongji Yang. "Developing Application Specific Ontology for Program Comprehension by Combining Domain Ontology with Code Ontology". In: *Proceedings of the 2008 The Eighth International Conference on Quality Software*. QSIC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 225–234. ISBN: 978-0-7695-3312-4. DOI: 10.1109/QSIC.2008.31. URL: http://dx.doi.org/10.1109/QSIC.2008.31.