© 2008 TANYA CRENSHAW

ARCHITECTURE REFERENCE MODELS FOR EARLY SAFETY ANALYSIS

BY

TANYA CRENSHAW

B.S., University of Portland, 1998

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Doctoral Committee:

Professor Lui Sha, Chair
Research Associate Professor Elsa Gunter
Professor P.R. Kumar
Research Associate Professor Ralph Johnson

Abstract

In 2001, Lui Sha published a paper entitled "Using Simplicity to Control Complexity." It describes an architecture that switches between a high-assurance-control subsystem and a high-performance-control subsystem. But his solution is much bigger and can be more widely applied; the Simplex architecture is a solution-creating technique for combining two algorithms such that a system retains the safety of the first while gaining the features of the second.

Using this architecture has been difficult because it has not been clear what kinds of problems the Simplex architecture solves; neither has it been clear in what ways developers can describe Simplex to conduct an early analysis of their own Simplex-based designs.

Simply put, my work is as much about Simplex as it is about describing Simplex architectures. This dissertation provides a collection of precise, logical descriptions of the Simplex architecture in four different modeling paradigms. I also describe my implementation of a Simplex architecture in a distributed control environment.

For Keesha, a good dog.

Acknowledgements

How shall I go in peace and without sorrow? Nay, not without a wound in the spirit shall I leave this city.

Long were the days of pain I have spent within its walls, and long were the nights of aloneness; and who can depart from his pain and his aloneness without regret?

Kahlil Gibran, *The Prophet*. 1923.

In *The Prophet*, Almustafa spends twelve years in the city of Orphalese waiting for his ship to return him to the isle of his birth. One day, he climbs the hill near the city and sees his ship returning. He is exuberant; he gets to go home. But just as quickly as his heart fills with joy, it is diluted with equal parts sorrow. While his years in the city have been unkind to him and have caused him great pain, the people have not. As he reflects upon this, the people of the city, who have also seen his ship approach, gather at the city gates to say their farewell.

At the time I deposit this dissertation, I will have been in Illinois for five years, ten months and 21 days. In that time, the Midwest has offered me an army of friends, colleagues and mentors that have helped me through this greatest life challenge:

- My best friend: Joe Hendrix.
- My advisor who challenged me with research and taught me how to brag about myself: Lui Sha.
- The funding agencies and industry leaders who gave their financial support: National Science Foundation, Rockwell Collins, and Lockheed Martin. This includes NSF award CSR-EHS:0720482.
- My small circle of good friends who kept me out of my tree with their pep talks, bike rides, ice cream and skim lattes: Erin Wolf Chambers, Amy Young, Shamsi Iqbal, and Iulia Dragan-Chirila.
- The Bevande Baristas who made so many of those lattes: Jane and Bethany.

- The woman who helped me get to multiple foreign countries, register for conferences, write and send important recommendation letters, get a job, and cope: Molly Flesner.
- Those who were lost: Andy Pack, the “Old Timer” Kent McConkey and Dr. Jennifer Hou.
- Those who were born: Grace Chambers, Thomas Belcher, Cole Stewart, Sean Thompson, Sarah and Chet Gardner, Miles and Mason Williams.
- The woman who made sure I got paid on time every semester: Shirley Finke.
- The university staff who helped me with funding, scholarship applications, outreach, and bureacratic administrivia: Angie Bingaman, Barb Cicone, Mary Beth Kelley, Kay Tomlin, Holly Bagwell, Sonya Harris, and Colin Robertson.
- The Mckinley physician who told me stories about anacondas, told me my finger would grow back, and helped me discover the three-medication combo that would enable me to breathe in and out in the over-pollenated midwest: Dr. Walter Maguire.
- My university faculty mentors: Susan Larson, Klara Nahrstedt, Lenny Pitt, Elsa Gunter, Craig Zilles, Ralph Johnson, and P. R. Kumar.
- My MentorNet mentor who textually kicked my butt; when I whined about staying at graduate school for a seventh year, she wrote to me, in all caps, DO NOT LINGER: Beth Simon.
- The goofy gentleman I met at SIGCSE in Houston who turned out to be invaluable in my job search, in planning a security course, and eating good food at later SIGCSE conferences: Paul Myers.
- The handful of strangers who agreed to talk to me about their jobs as teaching university professors which helped me to find my own job as a university professor: Roshanak Roshandel, Adair Dingle, Tammy VanDeGrift and Laurie Murphy.
- My “older brothers” in the Real-Time and Convergence Lab research groups who all graduated before me and taught me important lessons including the very practical lesson about keeping diagrams organized neatly throughout graduate school so that they do not have to be redrawn for the defense and dissertation: Deepu Chandy, Spencer Hoke, Satish Gopalakrishnan, Xue Liu, Ajay Tirumala, Sumant Kowshik, Craig Robinson, and Qixin Wang.
- My “younger brothers” who taught me more important lessons and gave me more respect than I probably deserved: Mu Sun, and Min Young Nam.

- All the friends and colleagues who ever read one of my mediocre paper drafts, unsatisfying job application packages, or watched one of my lame practice presentations, including my qual practice presentation where I melted into tears and Iulia had to pull me out of the bathroom and Joe had to tell Pedro to "*let her answer the question*" which is the moment when I really, *really* liked him: Damon Cook, Jeffrey Overby, Matt Belcher, Nitish Korula, Margaret Fleck, Danny Dig, Chris Garver, Michael Treaster, Jay Patel, Tony Chang, Jodie Boyer, Brittney Smith, Nicholas Chen, Adam Lee, Maurice Rabb, Roger Whitney, Ramona Thompson, and Baris Aktemur.

- The people who made me laugh: Heather Metcalf, Michael "The Artist" Katelman, Santiago Escobar, Clownhair Zack, Nat Thompson and Roy Campbell.

- My friends who kept reality nearby with radio shows, stories about women and mothers and daughters and husbands and boyfriends, sign language, and drawings of Thomas the Train: Ben "Hige-Chan" Walt, Kit Moore, Christina McQuirk, Chetan Palajani, Gretchen Klein, Zoe Chao, Grace Chambers, Ruchi Bhanot, Kim Belcher, and Lars Olendorf.

- The MergeSort dancers.

Finally, my thanks to the west coast satellite team: my mother and brother and father and friends: Erin, Chet^2, Sarah, Mrs. Clock, Cousin Lisa and Michael and the twins, Cheryl, Ryan, Maritza, Megan, and Rob, all of whom I will see very soon.

Table of Contents

1 Summary **1**

2 The easy button: A vision for software development **5**
2.1 Contributions 8

3 My other office is the convergence laboratory **9**
3.1 The five layers of the testbed 10
3.2 The application layer 10
3.3 The Etherware layer 11
3.4 The real world layer 12
3.4.1 The testbed's frontier: Nine cars, a camera, a C++ component, and a Java BufferedReader 12
3.5 Motivating the need for proven, domain-specific solutions 13
3.5.1 A case study of design patterns in the convergence laboratory 14
3.5.2 Metrics 14
3.6 Results 15
3.6.1 Memento usage improves system availability 15
3.6.2 Façade usage may incur dependency inversions 16
3.6.3 Proxy usage results in system-wide failure 17
3.6.4 Filter usage improves robustness by mitigating message delay and loss 18
3.7 Lessons learned 18

4 Defining the Simplex problem domain **19**
4.1 Problem frames tutorial 19
4.2 Problem frame catalog 20
4.2.1 Domain types 20
4.2.2 Required behavior problem frame 21
4.2.3 Connection domain problem frame 22
4.2.4 Model builder problem frame 23
4.3 The Simplex problem frame 25
4.3.1 Decomposing the environment domain 25
4.3.2 Observable safety properties 26
4.3.3 The problem frame 27

5 Simplex: Using simplicity to control complexity **30**
5.1 Examples: The Simplex prototypes 32
5.1.1 The inverted pendulum 33
5.1.2 The diving controller 34
5.2 Uncovering the necessary vocabulary for describing the Simplex architecture 36

6 A dot and a big table **41**
6.1 The simple dot example 41
6.2 Comparing three tools: Maude, AADL, A2M 42
6.2.1 Maude 43
6.2.2 AADL 43
6.2.3 A2M 44

7 Modeling Simplex in Maude **45**
7.1 A very brief introduction to Maude 46
7.2 A Simplex model in Maude 46
7.3 Model-checking the Simplex specification 49
7.4 Evaluation of modeling Simplex in Maude 49
7.4.1 Testable precision 49
7.4.2 Boundary delineation 50
7.4.3 Safety definition 51
7.5 Conclusion 51

8 Modeling Simplex in AADL **52**
8.1 A very brief introduction to AADL 53
8.1.1 History and motivation 53
8.1.2 AADL core standard 53
8.1.3 AADL behavior annex 56
8.2 A Simplex model in AADL 57
8.3 Evaluation of AADL for describing Simplex 60
8.3.1 Testable precision 60
8.3.2 Boundary delineation 62
8.3.3 Safety definition 66
8.4 Conclusion 68

9 The AADL2Maude toolchain **69**
9.1 The big links in the long chain 70
9.2 What is ready now: Peter's interpreter and my examples 71
9.3 A Simplex model for the A2M interpreter 74
9.3.1 The reactive plant 76
9.3.2 The two-phase, periodic controllers 76
9.3.3 Making a choice: The five-state decision machine 78
9.4 Executing and model-checking the Simplex specification 79
9.5 Analyzing dependency inversions in A2M 80
9.6 Evaluation of the A2M interpreter for Simplex 81
9.6.1 Testable precision 81
9.6.2 Boundary delineation 82
9.6.3 Safety definition 84
9.7 Conclusion 87

10 Simplex in higher order logic **88**
10.1 Decomposition of the logical reference model 89
10.2 Logical framework 92
10.3 Conculsion 94

11 Simplex improves safety in the convergence laboratory . . . 95
11.1 Applications in the convergence laboratory 97
11.1.1 The vision sensor subsystem 98
11.1.2 Flaws in the original vision subsystem 100
11.2 Be as accurate as possible 104
11.3 Never keep secrets 106
11.4 The resulting Simplex architecture 111
11.5 Conclusion 114

12 Related work 115
12.0.1 Patterns 115
12.0.2 Model-checking real-time systems 117
12.0.3 Formalizing patterns 118

A Vision configuration files 119

B Introductory sensor example in AADL 120

C Generic packages to aid with instantiating Simplex architectures in AADL 122

D Generic Simplex model in AADL 125

E Simple dot example in AADL 129

F Single-thread example in AADL 137

G Single-thread example in A2M 139

H Model-checking the single-thread example in A2M 142

I The simple dot example in the A2M 143

J Model-checking the simple dot example in A2M 157

References 158

Author's Biography 164

1 Summary

As a computer science researcher, I focus on the field of software engineering for real-time systems; I investigate how to conduct early safety analyses of safety-critical, real-time systems. Many such systems are Cyber-Physical Systems, or CPS, which comprise networked devices monitoring and controlling the physical world. Their domains include avionics, automobiles and health-management networks. For these domains, safety is paramount. Yet, current software engineering practices lack the foundations necessary to reason about the safety of these real-time systems. My work blends software engineering principles with a detailed knowledge of the CPS domain to provide developers with models of proven solutions; developers can use these models to instantiate their own solutions and conduct early evaluations of their safety-critical architectures.

To make an impact on software development for the CPS domain, I ground my research in the challenging systems development problems faced in the avionics industry. Systems in this industry are developed in multiple successive stages. Partial designs are captured in the early stages in databases, diagrams, and spreadsheets. Safety is designed into the system based on technical lore and the past experiences of senior designers. Domain expertise is important, but safety properties cannot be evaluated until the later stages when implementation artifacts are available. Severe design flaws discovered at these late stages adds detrimental cost to a project.

I have investigated architecture-level reference models to provide early design analysis without implementation artifacts. I leveraged numerous case studies, expert domain knowledge, and industry standards to innovate models of proven, domain-specific solutions for next-generation model-driven development processes.

Background

In 2001, Lui Sha published a paper entitled "Using Simplicity to Control Complexity." It describes an architecture that switches between a high-assurance-control subsystem and a high-performance-control subsystem. But his solution is much broader and more applicable; the Simplex architecture is a solution-creating technique for combining two algorithms such that a system retains the safety of the first while gaining the features of the second.

Using this architecture has been difficult because it has not been clear what kinds of problems the Simplex architecture solves; neither has it been clear in

what ways developers can describe Simplex to conduct an early analysis of their own Simplex-based designs.

Simply put, my work is as much about Simplex as it is about describing Simplex architectures. My work provides a collection of precise, logical descriptions of the Simplex architecture in four different modeling paradigms. As a result, developers now have a formulaic design approach to utilizing and analyzing the Simplex architecture for their own applications.

Summary of results

A case study of design patterns to motivate the need for domain-specific solutions for the CPS domain (Chapter 3). A natural origin for a model-driven development process is software patterns [30]. Patterns are proven solutions for a variety of problems faced by software developers. Aside from a small handful [19, 81], not many patterns are available for the CPS domain, especially those that pay attention to the hazards that can occur in CPS. My case study demonstrates how four classic design patterns are not sufficient for CPS developers, and motivates the need for domain-specific solutions, like the Simplex architecture

Etherware is a middleware for networked control [7] which drives much of my research. Though not an avionics system, Etherware is a tangible, very applicable, non-proprietary example of CPS. It is a component-based, networked system that observes and controls the physical world. Etherware's original developers used design patterns for its architecture. Based on a case study of Etherware's pattern usage, I illustrate the positive and negative impact that four classic design patterns have on key safety qualities [15]. The lesson gained from my survey is that CPS have characteristics that stand far apart from other domains. CPS relate to the physical world, a place that is adverse to safety. These systems are subject to unreliable interactions: sporadic or incorrect sensor input, and control commands not followed with precision. As a result, developers for these system need proven, *domain-specific* solutions.

A problem architecture in the CPS domain (Chapter 4). Etherware's navigation system is subject to unreliable sensor input and thus can calculate a flawed estimate of car locations. Yet, the navigation system must also deliver location information in such a way that collisions between cars are avoided. The same kind of interaction between navigation and control in Etherware is seen in flight control systems on commercial aircraft. The aircraft navigation system supplies location information to the autopilot, one of the most safety-critical subsystems in flight control. What kinds of models do developers need to resolve potential safety hazards due to interactions between unreliable components and safety critical ones?

To develop a collection of proven solutions for this problem first requires

a clear understanding of the problem itself. These systems must control the behavior of a plant situated in an environment introducing disturbances to the plant. The software that controls the plant is not directly connected to it; it may only make observations via a sensor and issue commands via an actuator. I have developed a detailed *problem architecture* which highlights hazards of this disconnect. Between estimating the plant state and articulating safety requirements, the Simplex problem frame presents a challenging problem to solve. However, understanding the problem puts a developer significantly closer to solving the problem successfully. Imagine trying to climb a mountain without knowing the terrain ahead; is it icy, rocky, or thousands of feet tall?

Modeling proven solutions during early design stages (Chapters 7, 8, and 10). I defined the problem architecture, but I also precisely defined a collection of Simplex solutions. These descriptions are provided as: i) A parameterized model in Maude, an executable specification language; ii) a parameterized model in AADL, an architecture description language; iii) a model in Higher Order Logic. As a result, developers now have multiple approaches for a formulaic design approach to utilizing and analyzing the Simplex architecture for their own applications.

Many CPS subsystem features can be described at two levels: minimal required features and desirable features. This divides a subsystem into two control algorithms. One is clearly safe. It is simple enough to verify. The second has more desirable features, but is too complicated to verify. Architectures for these subsystems can be created using any of my Simplex reference models which describe this technique for combining two such algorithms so that a system retains the safety of the first while gaining the features of the second.

A case study of a semi-automated analysis of specific Simplex architecture instances (Chapter 9). Given that I have grounded my work in the avionics domains, I have worked towards making my architecture solutions accessible to industry engineers. Formal logic offers the complete verification of safety properties, but demands a certain mathematical knowledge not found in most CPS developers who often come from the electrical and mechanical engineering industries. Based on industry surveys, I developed a set of six metrics to evaluate an automated Simplex analysis tool. Based on these metrics, I have conducted a case study of the existing AADL interpreter made available by José Meseguer and his formal methods group at the University of Illinois at Urbana-Champaign. Based on the case study, I provide a set of concrete recommendations to the formal methods group to improve their toolchain for an industrial setting.

A new prototype of the Simplex architecture in a distributed control environment (Chapter 11). My Simplex research conducted in the Ether-

ware testbed was driven by a demand for safety in real-time, mission-critical, embedded systems. When I was introduced to the testbed in 2004, cars were behaving strangely; at times, they were colliding with each other, a definite safety violation. My Simplex prototype provides the testbed with notable improvements in the way that cars are located and identified by the vision subsystem. In experimental trials, cars are now correctly reported at least 99% of the time. Moreover, the vision subsystem now expresses car location confidence so that other components can take fail-safe measures; cars are stopped if their locations are not well-known.

Intellectual Merit

Cyber-physical systems face uncertainty. Unreliable sensors, flawed plant estimates, inaccurate or worn-down devices, unreliable third-party components, and missing environmental assumptions all contribute to this uncertainty. My research has uncovered a proven system architecture which allows for the use of unreliable components to interact with safety-critical components without compromising safety. I have precisely identified and implemented solutions which can provide safety despite this kind of uncertainty.

Broader Impact

Model-driven development is a new development process paradigm that emphasizes higher levels of abstraction than current languages such as C++ and Java offer. At each stage of development, the system is represented as a more and more refined model, until the final system deployment is generated automatically. This kind of development process is currently getting a lot of attention, and promises to solve many of the problems faced in developing large, component-based systems. The computer science community is far from this kind of development process, but my models of proven solutions and my case study of a semi-automated analysis take the next steps toward this goal. Although set in the avionics domain, I expect that my research into modeling proven solutions for this domain can be extended to other domains of CPS.

2 The easy button: A vision for software development

Staples is an office supply company that advertises "the easy button." Need pens? Push the easy button. Need hot high tech gifts? Push the easy button. Want to save 10% on printer ink? Easy. Button. In reality, the button is a link that users can place on their Windows desktop to make ordering office supplies easier. In television commercials, the button is glossy and cartoon-red; it sits on the desk of the savvyest office employee, while the other employees fret in disaster, covered in yellow stickies and glue.

The commercials are silly, but the concept is visionary: a simple interface that is easy to use and empowers users with exactly what they need. There are other examples: Apple Computer's one-button mouse and Amazon.com's 1-Click patent. Science fiction offers more advanced machines. "Star Trek: The Next Generation" characters have their replicator. Walk up to the machine and say, "Make me a roast beef sandwich," and it does.

What if this easy-button concept could be abstracted, applied not just to machines and their interfaces, but also to the construction of these machines? It would be the machine to build and program all other machines. Like the replicator, one could approach it and say, "Make me a machine to mow my lawn," or, "Write me a program to control an airplane." This kind of machine for software might put me out of a job; I would happily spend more time practicing yoga and perhaps finally learn how to surf. Yet, as David Parnas warns, this kind of machine could not possibly exist for building software.

Depending on Parnas' television habits, he may know nothing of the easy button or the Star Trek replicator. Instead, he speaks about a rational design process, and that it may only be faked [56]. There are too many forces against a logical approach to developing software: Customers who don't know what they want, budget restrictions that force developers to reuse old or incompatible parts, political pressures that change the direction of the project, the C.E.O who insists on hiring his incompetent nephew. Ultimately, "human errors can only be avoided if one can avoid the use of humans" but no rational machine may be built either.

Walker Royce, Vice President of IBM Rational Worldwide Brand Services, describes similar observations of the development cycle,

> *The typical sequence for the conventional engineering project management style is: (1) early success via paper designs and thorough (often*

too thorough) artifacts; (2) commitment to executable code late in the life cycle; (3) integration nightmares due to unforeseen implementation issues and interface ambiguities; (4) heavy budget and schedule pressure to get the system working; (5) late shoe-horning of suboptimal fixes, with no time for redesign; and (6) a very fragile, unmaintainable product, delivered late [60, pg 46].

To address these issues, Parnas describes a series of thorough design, documentation, and implementation steps for a rational design process. Faking it is worthwhile, he argues, because it guides programmers on how to proceed to the next step, design better, backtrack less, and measure the progress for a particular project.

Parnas' faked rational process begins with the beautifully crafted requirements document that succinctly describes the behavior of the system; it is without duplication and inconsistencies, and it contains "everything you need to know to write software that is acceptable to the customer, and no more." It is written by the software's users, it is backed by a simple mathematical model, and it is organized using a separation of concerns. The requirements document is just step one of seven, and each step must be brilliantly documented. Parnas' admits the difficulty in this. Programmers regard documentation as evil, and if pressed, may not be able to identify a sentence out of a police line-up.

* * *

My domain of interest is Cyber-Physical Systems, or CPS, which comprise networked devices that monitor and control the physical world. These systems have three qualities that make them particularly interesting for a case study in a rational design process. First, they include critical infrastructures such as automobiles, manufacturing plants, avionics and health-management networks. For these domains, safety is paramount. Second, they relate to the physical world and are subject to unreliable interactions. Sensor input is sporadic or incorrect, and control commands are not always followed with precision. Third, CPS architects take advantage of the Commercial-Off-The-Shelf (COTS) component model. As a result, they gain the valuable expertise of third-party companies at a lower cost, but face the complexity of assembling large numbers of components they may know little about or may not be able to explicitly verify.

The task of faking a rational design process for CPS seems even more impossible: Improve the development of safety-critical systems built from a collection of black-boxes which relate to a multitude of unknowns in the physical world. In these systems, a single component fault must not invalidate system safety requirements, violate deadlines, or cause further mayhem in other components. Their massive size and small budgets do not allow formal verification of every component, so the architecture must be able to cheaply handle faults. Moreover, third-party components may be too complex to be formally verifiable, or

they may simply be untrusted. Why don't I just invent anti-gravity; it seems like an easier task.

Whether or not it seems possible is perhaps a premature question; one ought to ask first, "Is the ideal, even faked, really worthwhile?" One might reject ideals published by academics; most of these people have never had to deliver robust software to demanding clients. However, Parnas' argument is backed by a community of industrial software developers who demand better specification and evaluation tools for CPS. For example, the Core Method, based on survey results from member companies of the Software Productivity Consortium, echoes the rational design process; it describes the features that customers want for specifying requirements for large real-time systems. Paraphrased from [23], features should include:

- **Testable precision**. Support the development of precise and testable specifications for real time, mission-critical, embedded systems.
- **Boundary delineation**. Support the delineation of system boundaries, the precise specification of system interfaces, and the description of the system's environment. Users may indicate where the specification is internally incomplete, although the tool must allow users to isolate fuzzy or incomplete requirements and proceed with work on requirements that are well understood.

These features are just part of the dream, but what is the reality? Today's CPS developers, especially those in the avionics domain, are challenged by the huge systems they must develop and the lack of a rational design process with which to develop them. Instead, they rely heavily on heuristics and recommended practices [43]. Early designs are partially captured in ad-hoc representations using database, diagram, and spreadsheet artifacts. Safety must be designed into the system based heavily on technical lore and the past experiences of senior designers. As a result, safety properties cannot be evaluated until late development stages when implementation artifacts are available.

To close the gap between today's development cycle and the seemingly impossible rational design process, I have conducted research into a collection of precise reference models of the Simplex architecture. Introduced by Lui Sha over ten years ago, the Simplex architecture has been demonstrated on a number of case studies, including an inverted pendulum [3, 42], a diving controller [71], and an F-16 controller [65]. Thus far, it has been difficult for people to tell whether Simplex is applicable to their problem and how they might architect solutions based on Simplex principles. My Simplex reference model serves as a precise model of a proven solution in the CPS domain for model-driven developers to reuse in their own solution architectures.

2.1 Contributions

My research contribution is a proven system architecture in the CPS domain which allows for the use of an unreliable component to interact with a safety-critical component without compromising safety. This proven system architecture is described at the logical design stage in such a way that it is repeatable; it is described precisely enough so that developers may conduct an early analysis of their own domain-specific architectures without implementation artifacts.

My work is as much about Simplex as it is about describing Simplex architectures. In particular, I make these contributions:

1. A case study of design patterns to motivate the need for domain-specific solutions for the CPS domain (Chapter 3).
2. A collection of precise, logical descriptions of the Simplex architecture, a proven solution in the CPS domain. These descriptions are provided as:
    - A problem frame architecture (Chapter 4),
    - A parameterized model in Maude, an executable specification language (Chapter 7),
    - A parameterized model in AADL, an architecture description language (Chapter 8),
    - A model in Higher Order Logic (Chapter 10).
3. A case study of a semi-automated analysis allowing the assembly of specific Simplex architecture instances and evaluation of their safety properties (Chapter 9)
4. A new prototype of the Simplex architecture in a distributed control environment. I find this contribution especially exciting; it is the first time that the Simplex architecture has been prototyped for such a large platform in a research setting (Chapter 11).

3 My other office is the convergence laboratory

One of the greatest driving forces in my research has been the convergence laboratory testbed at the University of Illinois at Urbana-Champaign [1] [31]. This testbed uses a networked, component-based middleware for control called Etherware [6].

I selected the testbed because it demonstrates the three qualities of interest for the CPS domain introduced in Section 2. First, the testbed models a critical infrastructure: a small fleet of cars whose key safety property is collision avoidance. Second, the testbed is designed for control; it relates to the physical world. Such a control system must model real-world devices in the software. States of external devices must be represented in the code using software proxies. Device states and their proxies must be consistent. Third, it is a component-based testbed whose multiple components execute concurrently across multiple nodes. The testbed controls the plant using multiple heterogeneous nodes concurrently executing tasks such as sensing and actuation.

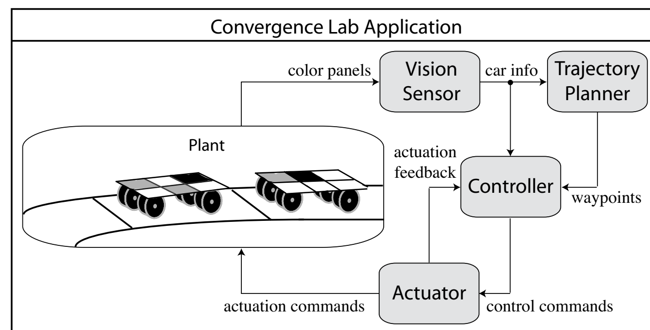


Figure 3.1: The testbed application components control a fleet of remote-controlled cars.

Since my research requires me to work as an application developer for the testbed, I introduce it here. This description provides sufficient background for the motivating case study of design patterns described later in this chapter and the implementation described in Chapter 11. For those more interested in learning about Etherware and the convergence laboratory, consult Girish Baliga's doctoral dissertation [7].

3.1 The five layers of the testbed

The convergence laboratory testbed employs a five-layer architecture. They are, from top down:

- **Application**. Application-layer components, shown in Fig. 3.1, execute on top of the testbed's Etherware middleware to control a fleet of remote-controlled cars. The application is based on a control loop consisting of a Vision Sensor, Controller, and Actuator. The Trajectory Planner generates waypoints for each car to be followed by each car's Controller. The cars have two directives. First, they must follow the set of waypoints determined by the system Trajectory Planner to reach a set of goal-bins in the network of roads. Second, they must do so without colliding with any other cars.
- **Etherware**. Implemented in Java, it is a domain-specific middleware for control over wireless or wired networks. The lossy channels and communication delays in wireless networks force Etherware to provide tools and services which accommodate these drawbacks without sacrificing control system stability. Etherware's goals are distributed operation, location independence and asynchronous communication, while being both scalable and robust.
- **Operating System**. Consists of support for threads and interprocess communication.
- **Network**. Consists of the physical layer of the typical network protocol stack.
- **The Real World**. A layer also observed in the *Pedestal* pattern [62], as discussed in Chapter 5. The real world layer consists of a plant and environment. The plant consists of the physical devices under the machine's control; in this case, these are the camera and the motors that control the steering angle and velocity of the cars. The environment consists of disturbances; one example is the friction of the track.

3.2 The application layer

The application model in the testbed is a collection of components. Each component is wrapped in a *Shell* (or *Façade* [29]) which provides a uniform interface between Etherware and the component. Components can record their state in a *Memento* [29]. The *Memento* offers support for failed component restart, migration and upgrade. Etherware service components also use these same design patterns.

Components at the Application layer communicate with each other via communication channels called Message Streams. Components may Tap a Message

Stream to intercept messages communicated between components. In this way, components may *Filter* [32] messages communicated between components along a Message Stream.

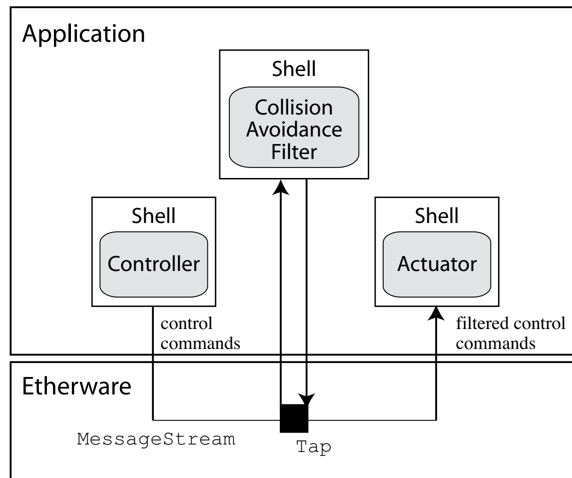


Figure 3.2: An example of Etherware's Message Stream and Tap. All application components are wrapped in a *Shell.* The Controller sends controller commands to the Actuator which receives these commands as filtered by the Collision Avoidance Filter.

The application of interest to this dissertation is the collision avoidance application; I mention it here to exemplify the Tap. The collision avoidance application uses the Tap mechanism to do its job. As shown in Figure 3.2, the Controller sends controller commands to the Actuator. Another application component, the Collision Avoidance Filter, intercepts these controller commands from the Message Stream using a Tap. If a car's trajectory is to intersect another, the Collision Avoidance Filter filters dangerous commands from the message stream, altering their contents to be the "stop" command. Once the two cars' trajectories diverge, the commands are no longer altered.

3.3 The Etherware layer

Etherware itself is made up of two parts, a kernel and a set of three service components. The kernel schedules tasks and delivers messages to local components in the Etherware and Application layers. The service components provide additional services. The first service is a ProfileRegistry which manages the available components in the system. A LocalProfileRegistry is located on each node and maintains a list of components on the particular node. A single node is equipped with a GlobalProfileRegistry which maintains a list of the other components throughout the system, including their name and port information. Meanwhile a second service, the IPNetworkMessenger, provides the functionality necessary for message delivery to other nodes in the network.

Messages communicated by Etherware are well-formed XML files. Message delivery functionality is restricted to the IPNetworkMessenger, making it simple to port Etherware to different network types [7]. The third service provided by Etherware is a NetworkTimeService translates time stamps of messages from one machine to another so that messages may be properly ordered for stable control. The service constructs a table of clock offsets and skews for the set of machines in the distributed system, and translates time stamps based on the appropriate offset, skew and communication delay.

3.4 The real world layer

The real world layer consists of a plant and environment. The plant consists of the physical devices under the machine's control. Given that the plant must interface to the Etherware and Application layers, I describe it here.

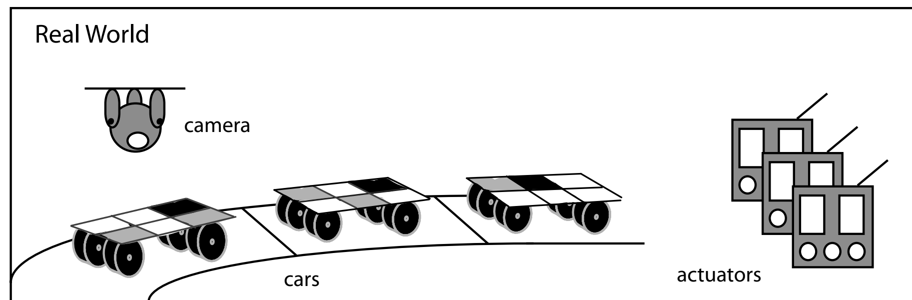


Figure 3.3: The real world layer comprises cars, actuators, and two cameras mounted to the ceiling of the testbed laboratory.

As shown in Fig. 3.3, the plant comprises:

- **Cars**. A small fleet of remote-controlled, battery-powered cars.
- **Actuators**. A set of FM-transmitters, one for each car. Each actuates the two motors which control steering angle and velocity on a given car.
- **Cameras**. Two cameras are mounted to the left and right halves of the testbed's ceiling to capture the two halves of the driving platform.

Ideally, I would just plug an actuator into Etherware via a Java interface, but reality is not that simple. This is actually one of the reasons I was originally drawn to embedded systems research: how does one make a computer talk to a *thing* like a car or a camera?

3.4.1 The testbed's frontier: Nine cars, a camera, a C++ component, and a Java BufferedReader

Every car in the testbed is equipped with a little paper quilt of six color patches that uniquely identifies it. Observations of car locations and orientations are

made in the testbed with two cameras mounted to the ceiling. The raw camera feed is processed by a C++ Matrox Imaging Library (MIL) that extracts a set of color blobs.

So far so good. But how does one go from this C++ library to the Java Vision Sensor component which delivers car information to the rest of the Etherware application? The collection of blobs produced by the MIL are interpreted by a C++ module, CarTracker. This C++ modules is invoked by the Etherware VisionSensor component as a Java `Runtime` object. The location information is relayed to Etherware via a Java `BufferedReader` named CarTrackerReader. Once received, the VisionSensor can deliver the car information to the rest of the interested components.

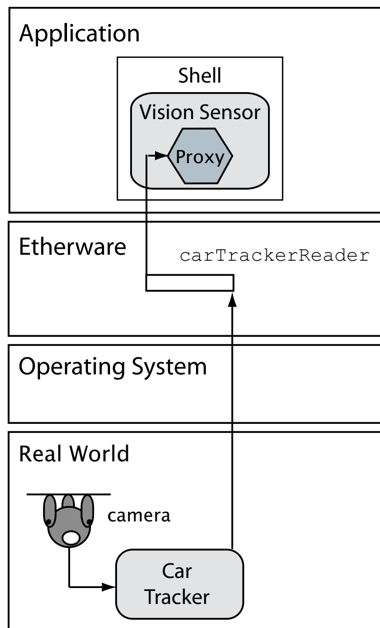


Figure 3.4: The CarTracker is a C++ component executed by Etherware as a Java `Runtime` object which uses a third-party imaging library to extract car location information. The location information is relayed to Etherware via a `BufferedReader` named CarTrackerReader. The CarTracker is invoked by the VisionSensor component and housed within the component's *Shell*.

3.5 Motivating the need for proven, domain-specific solutions

Armed with this introduction to the testbed, I return to the greater vision of this dissertation: an offering of Simplex described precisely enough so that developers may conduct an early analysis of their own domain-specific architectures without implementation artifacts. This description must support model-driven development, featuring testable precision and boundary delineation.

A natural origin for a model-driven development process is software patterns [30]. Patterns are proven solutions for typical problems faced by software developers. However, aside from a small handful of resources [19, 62, 81], not many patterns are available for the CPS domain, especially those that pay attention to the hazards that can occur in CPS. As a result, many CPS developers are faced with building solutions based on generic design patterns.

My early work with patterns investigates the impact of developing domain-specific solutions based on generic patterns. My case study evaluates Etherware, a domain-specific middleware for networked control whose original developers used design patterns for its architecture. Though not an avionics system, Etherware is a very applicable, non-proprietary example of a CPS. It is a component-based, networked system that observes and controls the physical world while maintaining strict safety properties.

3.5.1 A case study of design patterns in the convergence laboratory

My case study examines four patterns [30, 32]. They are:

- *Memento*. Records the internal state of an object. This record can be used to later restore the state of the object.
- *Façade*. Provides a simplified, high-level interface for a set of objects. Clients invoking these objects' services don't have to be concerned with the differences between their varying interfaces, and can just invoke services using the *Façade*.
- *Proxy*. Instantiates an object in place of another. In many cases, a *Proxy* creates a local placeholder for a remote object.
- *Filter*. Allows dynamic compositions of objects to perform transformations on streams of data.

I evaluate Etherware's pattern usage based on each pattern's impact on availability, robustness, and reliability, three qualities which impact a system's safety [44, 9]. The evaluation also includes a dependency inversion analysis [18] to analyze the relationships between component failures. The case study does not intend to evaluate the patterns themselves, but rather advance the community's understanding of patterns' utility in the CPS domain.

3.5.2 Metrics

A system is safety critical when its incorrect behavior can directly or indirectly lead to a state hazardous to human life [44]. Decisions which shape the architecture for CPS are driven in part by three qualities; availability, reliability, and robustness [44, 9].

- **Availability** can be quantified by the probability that a system is available when needed [9]. To increase availability of software components, developers can use fault-detection or fault-recovery tactics such as *heartbeat*, a periodic signal emitted by one component and monitored by another.

- **Reliability** can be quantified by the probability that a component will perform its intended function satisfactorily for a prescribed time and under stipulated environmental conditions [44]. Reliability is important to safety-critical, real-time systems, for it insures that components correctly execute to completion and meet their deadlines. For example, a *Watchdog* pattern can be used to monitor the internal, time-dependent computational progress of a subsystem [19].

- **Robustness** is the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions [54]. One approach to robustness is to ensure that components enter a fail-safe state under such conditions. The *Safety Executive* [19] is one pattern that describes how to coordinate a component's entry into its fail-safe state.

3.6 Results

3.6.1 Memento usage improves system availability

Etherware components may store state information in a *Memento* at termination. Using the *Memento*, Etherware can perform quick reinitialization of failed components, or migrate components to other nodes during system execution. The *Memento* helps Etherware to achieve high availability, since quick restarts and component migration increase the probability that a component is available when needed.

While Etherware's *Memento* improves availability, it can be so expensive that it affects reliability. The time and space demanded by storing and transferring the *Memento* can decrease the probability that a component will perform its intended function, since there is more opportunity for error. However, foregoing it also means foregoing the advantage of quick restarts and component migration. For Etherware in particular, the potential expense has not been a concern; it has not caused problems with the message size and transmission time demanded by the *Memento*.

The *Memento* also offers a means to improve robustness. In many cases, component failures are contained within a component's *Shell* and components can recover gracefully by restarting with a *Memento*.

Overall, the *Memento* is important for safety-critical systems. In terms of all three software qualities, Etherware's *Memento* has a predominately positive

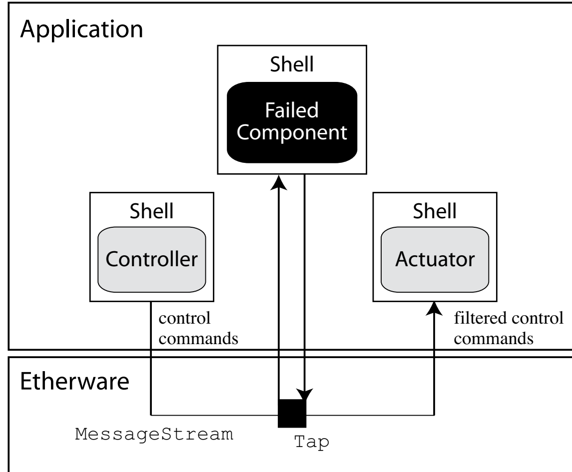


Figure 3.5: A failed component remains in the Message Stream. Messages from the application-level Controller to the Actuator are sent via the *Shell* of the failed component, incurring unnecessary delay and possible corruption.

impact on safety. In particular, there has been significant benefit arising from the capability for quick component restarts.

3.6.2 Façade usage may incur dependency inversions

Etherware employs a *Façade* or *Shell* to mask the failures of application-level components from the rest of the system. While Etherware's *Shell* has little impact on availability, it improves reliability. Thanks to the quick restart provided in part by the *Memento*, a component can fail and restart within a *Shell* without other component's knowledge, and still perform its function by the deadline.

In the same way, the *Shell* achieves greater system robustness, since component failures are contained within a component's *Shell* and components can recover gracefully by restarting with a *Memento*. Unfortunately, some component failures in Etherware are not recoverable during system execution, despite the *Memento*.

When such failures occur, failed components contained in their *Shells* remain in the path of the Message Stream as seen in Figure 3.5. This forces messages to incur unnecessary and possibly harmful delay. Moreover, messages may also be corrupted as they make their way from their source to their destination via failed components that are tapping or filtering a given Message Stream.

Though somewhat helpful in improving robustness, it is important to consider the criticality of a *Shell* with respect to its component. The dependency inversion analysis [18] conducted in [15] provided insights to the interactions between critical and non-critical components in the system. "Dependency inversion" is the dependency of a safety-critical component on less critical components [18]. This is an undesirable alternative to a use relationship in which one component uses the services of another, but can continue to perform in spite of

any failure of the other.

The analysis revealed a dependency inversion between a *Shell* and its component in Etherware. As a result, one must consider the alternatives to resolve this relationship. The first alternative is to utilize fault-tolerance techniques that eliminate possible fault propagation from the less critical component to the more critical one, thereby removing the dependency inversion. The *depend* relationship becomes a *use* relationship. If such techniques are not viable, the second alternative is to recategorize the less critical component at least as safety-critical level as the critical component. This implies that more time and budget must be dedicated to validating the newly categorized component.

3.6.3 Proxy usage results in system-wide failure

Etherware is a middleware for controlling devices in the physical world, or the **Real World** layer [62]. These devices must be represented in software at the **Application** layer. This is done using a *Proxy*. Figure 3.4 summarizes the details of the single *Proxy* used in the collision avoidance application. This *Proxy*, invoked by a separate component and housed within that component's *Shell*, mirrors the state of all the cars on the track. Two cameras capture the locations of all the cars on the track. A VisionTracker processes the camera image to uniquely identify each car, obtain its x-position, y-position, and orientation, and stream them all to Etherware via a special buffer, CarTrackerReader. The *Proxy* then receives location information from this reader.

In the original design of Etherware, the *Proxy* was not considered as a possible source of system failure. Its functionality was assumed to be too simple to cause system failure. However, the wear and tear of the physical devices led to unexpected failure. The color panels used on the cars for unique identification faded over time, causing the VisionTracker to treat one car as two different cars in the exact same location. A division-by-zero occurred in the C++ component, leading to a segmentation fault in the VisionTracker. The C++ component was invoked as an external program using Java's `Runtime exec()` method. The segmentation fault caused the Java Virtual Machine on the particular node to crash. Without the node providing the VisionTracker, car locations could not be obtained, and the remaining components suffered. The bug led to a system-wide failure.

The bigger question is, "Is it really *Proxy's* fault?" No. In this case, the proxy was doing exactly what it was supposed to do. The real problem was that the original designers of Etherware did not design the proxy carefully enough such that it was robust against inaccurate observations of the physical world.

3.6.4 Filter usage improves robustness by mitigating message delay and loss

Etherware's support for components to *Filter* the Message Stream provides a mechanism by which communication delays and errors can be mitigated. In particular, the Kalman Filter improves robustness, since failure to convey a particular sensor message does not cause failure in the Controller. This particular use of a Kalman Filter is rather unlike what was originally intended by the pattern authors. Instead of composing operations on streams of data, as done by the Collision Avoidance Filter, the Kalman Filter estimates the current state of the cars using previous data and control commands.

This observation about the use of the *Filter* motivated the development of a new software pattern. The Kalman Filter performs state estimation based on current and previous data rather than simply applying an operation to a stream of data. Investigations of other safety-critical middleware technologies, notably the Gain Scheduler Middleware described in [76, 77], demonstrated similar augmentations. As a result, what came from the Etherware case study was not simply the lessons learned from pattern usage, but also my development of a particular software pattern. This pattern, the *Adaptive Control Filter* is described in its entirety in [14].

3.7 Lessons learned

The case study illustrates the positive and negative impact the usage of four classic design patterns in Etherware has on key safety qualities. The results of the case study found positive and negative impacts. The *Memento* design pattern improves system safety while the *Filter* improves robustness. However, implementing a *Proxy* for real world devices requires consideration such that robustness is not negatively impacted. Similarly, *Façade* can introduce dependency inversions.

The lesson gained from the *Proxy* usage is most relevant. Because CPS relate to the physical world, they have characteristics that stand far apart from other domains. The physical world is a place that is adverse to safety. These systems are subject to unreliable interactions: sporadic or incorrect sensor input, and control commands that are not always followed with precision. As a result, developers for these system need proven, *domain-specific* solutions that account for their special safety requirements.

4 Defining the Simplex problem domain

Having motivated the demand for domain-specific CPS solutions, the next step is to identify one family of CPS problems that require solutions. Problem frames [38] is an approach developed by Michael Jackson to describe and categorize problems. They are used to describe a software development project's problem architecture, and decompose a problem into multiple subproblems. In this way, smaller subproblems can be resolved using well-understood solutions.

4.1 Problem frames tutorial

Problem frames are used to describe a software development project's problem architecture, and decompose a problem into multiple subproblems. In this way, smaller subproblems can be resolved using well-understood solutions.

A problem frame is decomposed into individual domain descriptions: a single machine domain, a collection of domain descriptions, and a set of requirements on the domains. Suppose the problem is to write software to drive a remote-controlled car around a circular track. The car is equipped with a left and a right sensor; each detects the car's distance from either edge of the track. Using this example, and Fig. 4.1, I quickly tour of the essential parts of a problem frame:

- **Machine domain.** Indicated by a double-stripe, the machine domain is the computer whose software must be designed and developed to solve a problem. For the car and track problem, the machine domain contains the software developers must write to control the car.

- **Given problem domain.** Shown as a plain box, a given problem domain is one whose description is given, such that the software developer is not allowed to design the domain. For the car and track problem, the given problem domain is comprised of the remote-controlled car and track

- **Designed problem domain.** Indicated by a single-stripe, it is a problem domain that the software developer gets to design, such as a database or display.

- **Shared phenomena.** The solid lines represent the interfaces between domains, or what Jackson calls shared phenomena. For the car and track problem, an example of shared phenomena is the machine command to

the car. This is data that is generated and sent by the machine domain and is received and used by the car's actuators.

- **Requirement.** Indicated by a dashed oval and arrow, the requirement stipulates some required behavior in a particular domain that the machine domain must guarantee. The requirement is a description of *what* a domain must *do*, not a description of how the machine achieves it. For the car and track problem, the car is required to drive along the track, but no statement is made as to how this ought to happen.

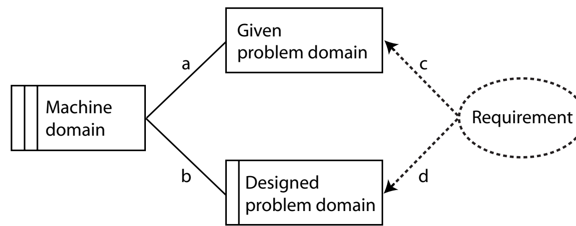


Figure 4.1: A basic problem frame. Boxes represent the various domains. Labeled interfaces between the domains represent shared phenomena.

4.2 Problem frame catalog

Jackson introduces a catalog of different problem frames that can be applied to a variety of individual subproblems. He organizes this catalog using two different dimensions:

- **Domain Types**. Based on the phenomena shared by a domain, a domain may be causal or lexical. Section 4.2.1 describes these two types of domains.

- **Problem Frame Classes**. A number of problem frame classes exist; I introduce three that apply to the CPS problem domain. Based on the type of problem being solved, an individual problem frame can be a *required*, *connection domain*, or *model building* problem frame. Sections 4.2.2-4.2.4 describe each.

4.2.1 Domain types

Shared phenomena may be causal or symbolic. Causal phenomena are directly controlled by some domain, such as a car's velocity; meanwhile, symbolic phenomena simply express values which symbolize other phenomena, such as a database entry storing a car's velocity. These types of phenomena define the types of domains in a problem frame.

Jackson describes two types of domains relevant to this dissertation: lexical and causal. A lexical domain is a physical representation of symbolic phenomena; these domains comprise data stored in main memory or on a hard disk. Lexical domains are typical in computer science, and include databases, file systems, or any other structure which manipulates symbolic phenomena.

In a causal domain, there are explicit rules about the relationships among phenomena. Consider again the example of controlling a remote-controlled car. If the machine domain commands the car to increase its speed, the car's motor turns faster, and its speed increases.

4.2.2 Required behavior problem frame

The required behavior problem is to build a machine which controls some real-world behavior so that it satisfies certain conditions. Shown in Fig. 4.2, the problem frame consists of two domains: the control machine, indicated by a double stripe, and the controlled domain. The "C" labeling the controlled domain indicates it is causal. The requirement, in the dashed oval, stipulates some desired relationship involving phenomena `b` in the controlled domain.

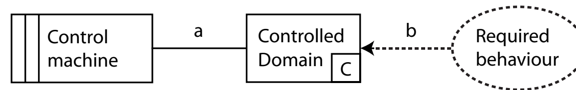


Figure 4.2: Jackson's required behavior problem frame. The "C" indicates that the domain or phenomena are causal.

Fig. 4.3 shows the example problem frame with a car controller machine and a causal car and track domain. The car controller and the car share two phenomena; I use Jackson's notation to label these. The car controller generates steering angle and speed commands to the car; I indicate this with `CC!{angle, speed}` to make it explicit that the *car controller* domain generates this shared phenomena. The car provides track sensor feedback indicating its distance from either edge of the track. Similarly, this phenomena is labeled `CT!trackSensor`.

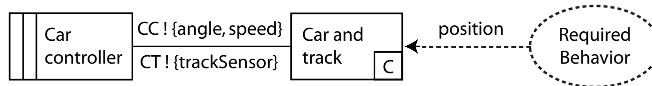


Figure 4.3: My example of the required behavior problem frame: build a machine which directs a car to drive along a track.

The car and track domain description must contain the information necessary to build the car controller machine. This includes properties about the car and track, including: i) the dimensions, mass, and maximum speed of the car; ii) the shape or road conditions of the track. Moreover, the requirement constrains the car and track domain such that the car's `position` stays within the track.

Similarly, the car controller machine specification must "describe a machine whose behavior will ensure that the requirement is satisfied" [38]. For example, the car controller machine must use the car and track domain properties to calculate and issue appropriate commands based on sensor readings.

4.2.3 Connection domain problem frame

Fig. 4.4 shows a connection domain problem frame. The problem is to build a machine that controls some behavior in an environment. The machine is connected to the environment indirectly through a sensor and actuator domain. With respect to the environment, there are two kinds of shared phenomena. The first are the monitored variables controlled by the environment. The second are the controlled variables controlled by the actuator. The requirements describe the relationship between the monitored variables and the controlled variables in the environment.

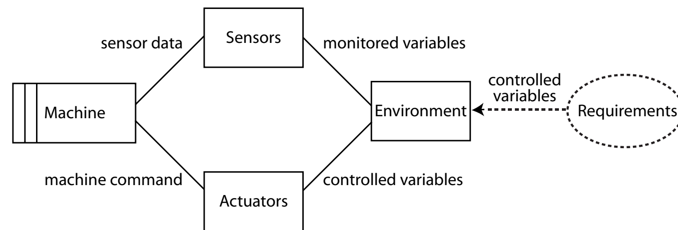


Figure 4.4: The connection domain problem frame. The machine interacts with the environment via two connection domains.

In the previous subsection, I introduced the simple problem of a car following a track. I used the required behavior class to describe the problem. It has only two domains: the controlled and the controller. But even simple systems like this have more that need to be considered. First, sensors must be used to detect the car's distance from either edge of the track. Second, a remote control device, or actuator, must be used to control the motors on the car. These sensor and actuator domains, which relay information from the controlled domain to the control machine, are two examples of what Jackson calls a connection domain.

Revisiting the example in Fig. 4.5, this problem domain makes the sensor and actuator domains explicit. In doing so, it distinguishes the monitored variables from the controlled ones. The track sensors monitor the voltage levels at the left and right. These voltage values are translated into left and right track positions, and reported to the car controller. The car controller uses these values to calculate the next steering angle and speed commands, which are sent to the actuator domain. The actuators translate these commands into voltages and are given to the car's stepper motor which controls the steering, and the DC motor which controls the speed.

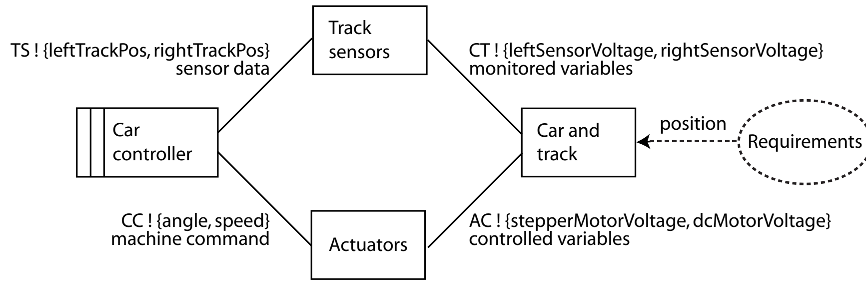


Figure 4.5: My example of a connection domain problem frame. The car controller machine is connected indirectly to the car environment via a sensor and actuator domain.

4.2.4 Model builder problem frame

In the previous problem frame, recall that the track sensor domain must translate the voltage values obtained from the environment into values which can be reported to the car controller. For simple scenarios, this translation merits a few variables to keep an internal, software view of the real world. In the simple car and track example, it only takes a couple operations to translate the track sensor readings into the car's actual distance from either side of the track. These operations require the granularity of the sensor readings, the units of measurement, and other calibration information.

However, in more complex scenarios, this translation from the sensor readings to the software's internal view of the real world suggests that the sensor domain must maintain a full model of the environment. Suppose that instead of simple sensors, an aerial camera is used to monitor the location and direction of the car. In this case, an arsenal of vision libraries would be necessary to translate raw camera data into the software's internal view of the real world. Such a scenario demands a model.

The model building problem frame makes modeling the real world explicit. Any problem using this domain is split into two subproblems; the machine that builds the model and the machine that uses it. Using the example shown in Fig. 4.6, the problem is: i) to build a machine that models the real world, and; ii) build a machine that displays the modeled real-world phenomena.

The model building problem frame is special because it is explicitly translating causal phenomena in the real world into lexical phenomena that another machine can manipulate. As a result, the real world domain is labeled with a "C" indicating that it is a causal domain; the "X" indicates that the model is a lexical domain.

The requirements in this problem frame are also somewhat unique, compared to my previous problem frames. In the top of Fig 4.6, notice that the dashed line connecting the real world and the requirements has no arrow. This means that the requirements only refer to phenomena in the real world, but do not constrain them. This is because the ultimate goal of the model building problem frame

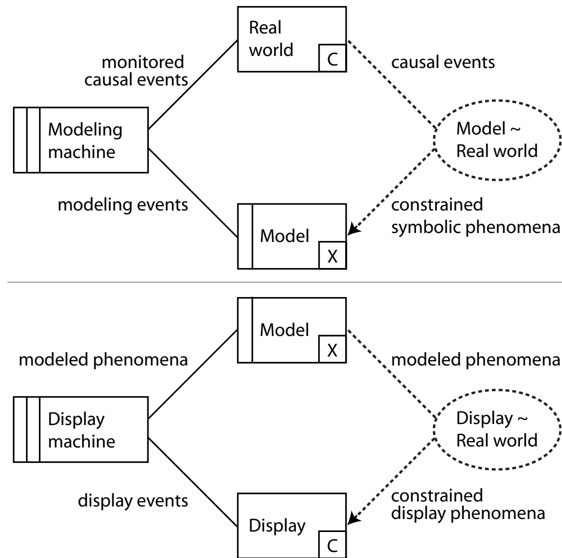


Figure 4.6: The model building problem domain (top). The problem is to build a model which approximates the real world. A second problem frame (bottom) uses the modeled phenomena to display the causal events.

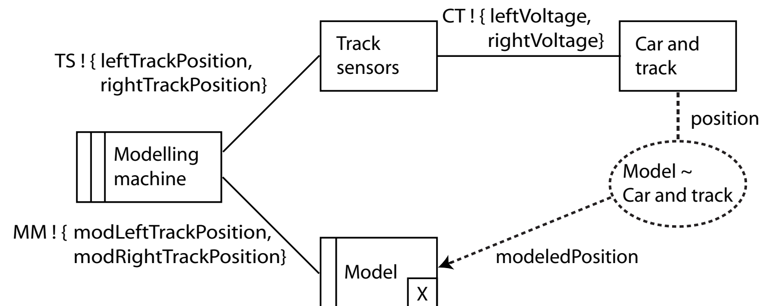


Figure 4.7: The example problem must model the car and track using information obtained from the track sensors, a connection domain.

is to create a machine such that the causal events correspond to the symbolic representation of that phenomena.

What does this mean for my example? As shown in Fig. 4.7, I must consider the problem of modeling the car and track, using information obtained via the track sensors. As before, the track sensors monitor the real world via voltage values of the left and right sensors. The track sensors provide track positions to the modeling machine. These positions are then represented as modeled positions, which ultimately lead to a modeled car position.

There is a real challenge in solving the model builder. For even in a causal domain, the set of modeled physical laws may not always be followed. Motors wear out, sensors break, and control commands are lost over unreliable communication lines. Moreover, the model may be incomplete; an unexpected failure, such as a fire, could occur. In the implementation highlighted in Chapter 11,

the challenge was to combine an unreliable camera with a set of physical laws to result in a 99% correct identification rate.

4.3 The Simplex problem frame

Armed with a quick tutorial on problem frames, and an introduction to the connection problem frame and model builder problem frame, I can discuss one family of problems seen in the CPS domain. I call it the "Simplex Problem Frame" because this family of problems can be solved by the Simplex architecture that will be introduced in the next chapter.

The problem is to build a machine which:

1. Controls some part of the physical world according to some minimum feature set such that safety is guaranteed. If safety requirements are met, then the machine may control the physical world according to some additional desired features;
2. Controls this part of the physical world via two connection domains: a sensor domain, and an actuator domain;
3. Guarantees safety despite the connection domain's unreliability.

Only problems whose functionality can be divided into a "minimal feature set" and "desired feature set" can be formulated into a Simplex problem frame. For problems without this dichotomy, the Simplex architecture is useless.

To describe the problems solved by the Simplex architecture, I contribute two additional problem frame concepts not provided by Jackson's work.

- **Decompose the Environment Domain**. Given that the Simplex architecture is predominately targeted for control domains, I use the ubiquitous language [22] of the domain and separate the environment into an environment and a plant. This distinction is described in Section 4.3.1.
- **Identify the Observable Safety Properties** The Simplex architecture requires that the safety properties maintained in the physical plant are properties that can be observed. Moreover, these properties must further be broken into high-confidence safety and core safety. These kinds of safety properties are articulated in Section 4.3.2

4.3.1 Decomposing the environment domain

Given that the Simplex architecture is predominately targeted for control domains, it is important to make a distinction between the plant and the environment. In control systems, the plant consists of the physical devices under the machine's control: the speed or turning angle of the car. The environment

consists of disturbances: phenomena which can disturb the plant, but the machine cannot control. Disturbances can include phenomena such as the road conditions or the current wind speed.

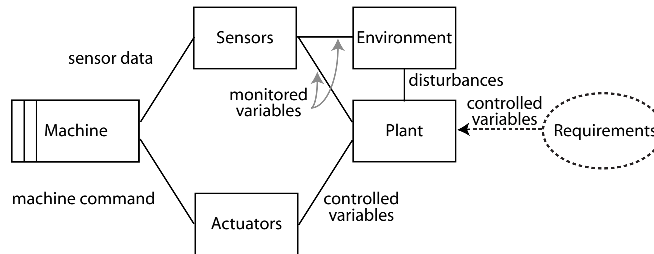


Figure 4.8: For control systems, separate the environment domain from the plant domain.

To make this notion of a plant explicit, revisit the connection domain problem frame. Fig. 4.8 separates the plant domain from the environment domain. The sensors share monitored variables with both the plant and the environment. In addition, the plant and the environment share a set of disturbance variables introduced by the environment. In many control scenarios, many of the phenomena in the environment are considered a disturbance to the plant. This is because the disturbance phenomena are assumed, are not directly detectable, or are too costly to detect. Most systems can safely assume the gravitational constant on earth; the modeling machine uses this assumed value when modeling the plant. In automotive systems, sensors cannot directly detect icy road conditions, but they can detect the slip rates of a vehicle's wheels or the side-to-side motion of a vehicle.

Thus, the union of the monitored variables and disturbance variables from the environment represent all possible information about the environment. However, the control machine must calculate and issue its commands based on the model built from: i) the monitored variables, and; ii) the assumptions made on the disturbance variables in the domain descriptions.

4.3.2 Observable safety properties

To motivate the concept of observable safety properties, return to the simple example with the car and track. Suppose again that the remote-controlled car must follow a circular track. The car is equipped with a left and right sensor such that each detects the car's distance from either edge of the track. Moreover, suppose that the track has a wall on either side which the car must not hit. Thus, it becomes a matter of safety that the car follow the track precisely.

First, to monitor safety properties in a plant, one must recognize that the software cannot detect directly whether or not the car has hit the wall. The software may only *observe* the values returned by the sensors. If the system never observes that the sensors return a value of "0" indicating the car's distance

from the wall, then the system can only conclude that the car has not yet hit the wall. Still, it may the case that a sensor has failed, and the system concludes that the system is safe based on erroneous information. Certainly, for any safety-critical system, redundancy is a useful approach to avoiding some of these problems [33].

Second, it is also necessary to differentiate between high-confidence safety and core safety. Consider the scenario in which the car is driving directly toward a wall and is currently 3 meters from the wall. In the time instant in which the 3 meter distance is observed, the car is safe; it is not colliding with the wall. However, if the car is also moving at a speed of 3 meters-per-second, then in the next second, the car will not be safe. Thus to achieve high-confidence safety:

- The plant must be safe at the current time instant;
- The modeling machine can predict, based on the current control command, that the plant will be safe in a future time window.

To state it another way, when a system has achieved high-confidence safety, it means that choosing the complex algorithm command now will not prevent the simple algorithm command from ensuring safety later.

There are a number of reasons why monitoring these kinds of safety properties is not trivial. First, for plants that cannot be modeled using a linear Gaussian model, it is a challenge to build a modeling machine which can predict future plant states. Second, in cases of monitoring distributed systems, knowledge at a given node may be incomplete [39].

Simplex, like any architecture concerned with safety, cannot guarantee absolute safety. First, its safety is limited by the safety properties that can be monitored by the system. The initial Simplex prototype could not have succeeded without the ability to detect if the pendulum were within its stability envelope. Second, it is limited by developer paranoia. A system can only be robust against the faults and failures that a developer anticipates. For example, in the early inverted pendulum prototype, Simplex could not have prevented pendulum instability that resulted from a faulty or corrupt sensor.

4.3.3 The problem frame

To formulate this family of problems seen in the CPS domain, I have decomposed the problem into three problem frames. The first is a model building problem frame. The second two are variations on the required behavior problem frame. Each problem frame decomposition interfaces the machine to the physical world via a connection domain.

Fig. 4.9 summarizes the collection of three problem frames. The top panel is the model builder; the problem is is to build a machine that constructs a precise model, or a safe approximation of the environment. To say the model is safe means that the modeled states are within limited safety tolerances of

the actual values in the real world. If these tolerances are not met, the model must indicate how, so that the software entities using the model can make appropriate decisions. Thus, the model requirements constrain the modeled variables to approximate the monitored and controlled variables in the plant and environment. Moreover, the safety requirements constrain the modeled variables such that they are within a safe range of the monitored variables.

The bottom two panels of Fig. 4.9 show the two connection domain problem frames that use the model. The problem is to build a machine that controls some part of the physical world, according to a set of safety requirements. For the center panel of Fig. 4.9, this machine implements a set of minimal features while guaranteeing the core safety requirements. For the bottom panel, the machine implements the set of desired features only if the high-confidence safety guarantees are met.

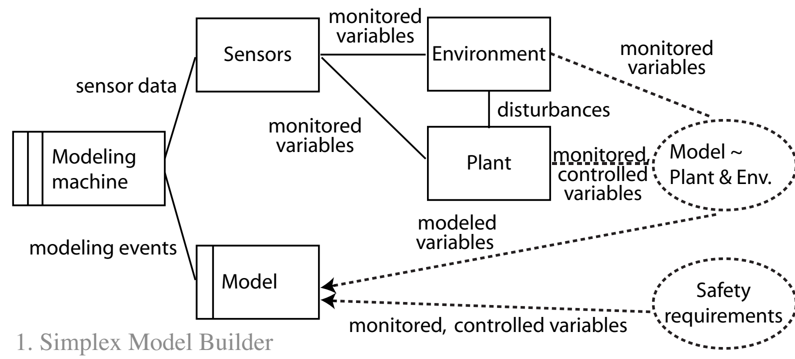
As a result, the Simplex problem frame demands three sets of safety requirements be defined. I list them here, each with a concrete example from the car and track example in Fig. 4.5

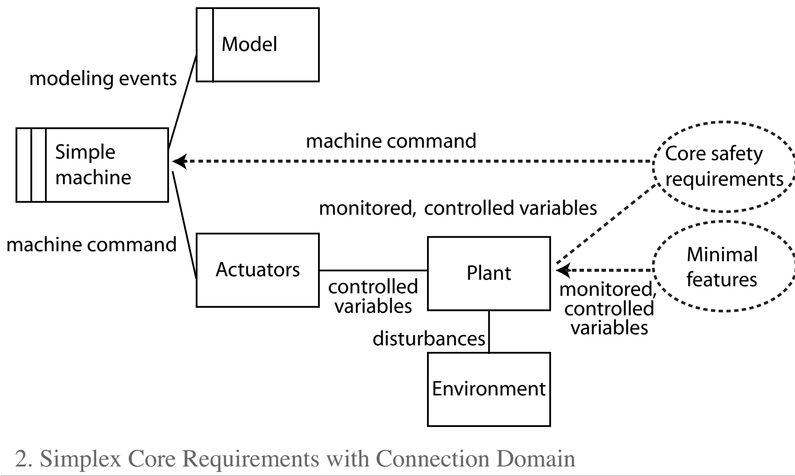
- **Model safety requirements**. The model is required to provide an estimation of the real world within a specified tolerance. For example, the model should approximate the location of a car within 0.25 meters.

- **Core safety requirements**. The simple machine is required to maintain plant safety. For example, the car must keep a distance of 0.5 meters from the track wall.

- **High-confidence safety requirements**. The complex machine is required to maintain high-confidence plant safety. For example, the car must keep a distance of 0.75 meters from the track wall.

Between building a good model and articulating the safety requirements, the Simplex problem frame presents a challenging problem to solve. However, understanding the problem puts a developer significantly closer to solving the problem successfully. Imagine trying to climb a mountain without knowing the terrain ahead; is it icy, rocky, or thousands of feet tall?





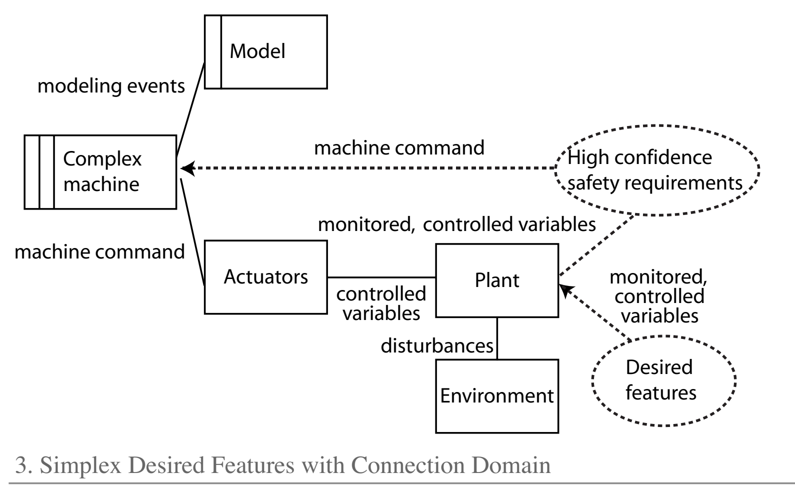


Figure 4.9: The trio of problem frames that comprise the family of problems. 1. The model builder describes the problem of building a model that approximates the disturbances as well as the monitored and controlled variables in both the environment and the plant. 2. The connection domain problem frame describes only the minimal required behavior. 3. Another connection domain problem frame describes the desired features along with the high-confidence safety requirements.

5 Simplex: Using simplicity to control complexity

In 2001, Lui Sha published a paper entitled "Using Simplicity to Control Complexity." It describes an architecture that switches between a high-assurance-control subsystem and a high-performance-control subsystem. But his solution is much bigger and can be more widely applied; the Simplex architecture is a solution-creating technique for combining two algorithms such that a system retains the safety of the first while gaining the features of the second.

One Simplex prototype is an inverted pendulum control application [3, 42]. The safety requirement is that the pendulum must not fall. The simple algorithm is used as a baseline control algorithm; it has been proven to work and is invoked only when the pendulum falls outside its stability envelope. The complex algorithm uses additional, possibly unverified, control laws and is used as long as the pendulum is stable. The emphasis of the prototype is dynamic upgrades: developers can upload untested control algorithms to the system on-the-fly without worry of a falling pendulum.

The Simplex architecture has been demonstrated on a number of case studies, including an inverted pendulum [3, 42], a diving controller [71], and an F-16 controller [65]. For each prototype, the system's functionality is separated into a "minimal feature set" enforced by a simple controller and "desired feature set" implemented by a complex controller.

Fig. 5.1 pictorially summarizes the complete Simplex architecture. Each box is a domain and each directed line indicates the data flow between domains. The figure shows a solution architecture; unlike the domains of Michael Jackson's problem frames, these domains are simply abstract components that reside within the architecture. Starting from the bottom of the figure, the environment introduces disturbances to the plant. Sensors can partially monitor the environment and the plant. The sensors deliver sensor data to the modeling machine that builds the model−−or the software's internal view of the external world.

The model delivers modeling events to both the simple machine and the complex machine. Each machine uses the event information to calculate an appropriate control command. Each machine forwards its control command to the decision machine, which chooses the appropriate control command to maintain the safety requirements.

For each pair of control commands, the decision machine forwards the selected control command to the model. The selected command is recorded in the

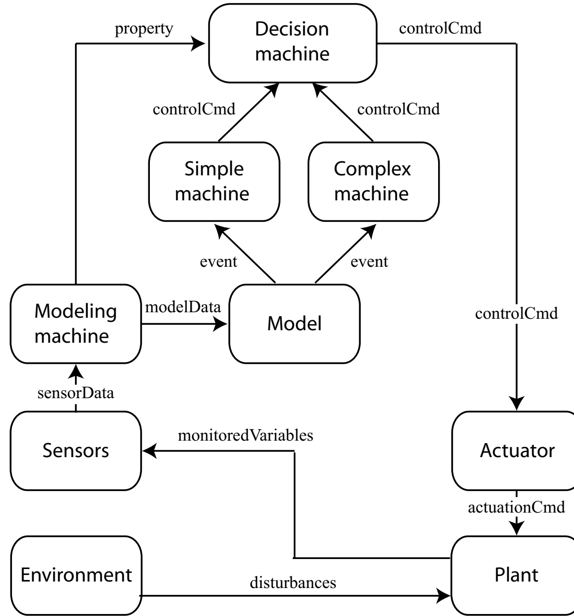


Figure 5.1: The Simplex Architecture

model for estimation and prediction purposes by the modeling machine as well as sent to the actuators. Finally, the actuation command reaches the plant, and the control loop is complete.

The decision machine is the brain of Simplex. To make the choice between the two algorithms, the decision machine requires all of the requirements specifications from the trio of Simplex problem frames:

- **Model safety requirements**. The model is required to provide an estimation of the real world within a specified tolerance.
- **Core safety requirements**. The simple machine is required to maintain plant safety.
- **High-confidence safety requirements**. The decision machine may only choose the complex complex controller if high-confidence plant safety is present.

Thus, the decision machine has a big job to do. Based on the properties delivered by the modeling machine, it has to determine if the current plant state satisfies the high-confidence safety requirements. If so, it is free to choose the control command from the complex machine. Otherwise, if the plant state satisfies only the safety requirements, it must choose the control command from the simple machine.

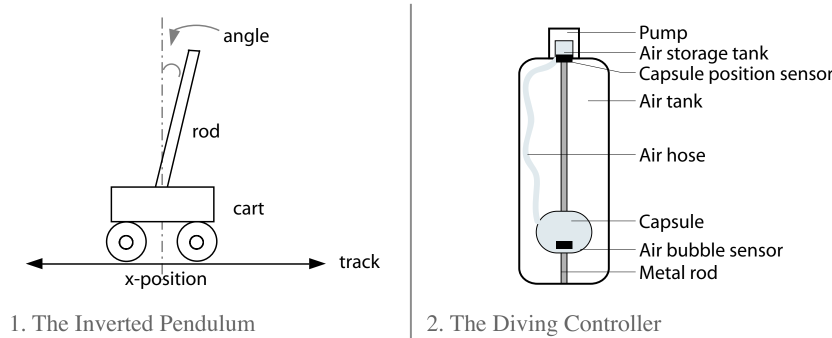


Figure 5.2: Two Simplex prototypes. 1. The inverted pendulum. A small cart moves back and forth along a track to keep the rod balanced. 2. The diving controller. A small capsule navigates up and down a metal rod to mimic the surface and submerge control for a submarine.

5.1 Examples: The Simplex prototypes

To provide more tangible examples of the problem-domain based description of the Simplex architecture, I discuss two existing Simplex prototypes developed at the University of Illinois at Urbana-Champaign and the Software Engineering Institute. These are the inverted pendulum and the diving controller prototypes, depicted in Fig. 5.2.

It is interesting to note the application-specific ways in which the complex machine differs from the simple machine. As shown in the upcoming prototypes, this difference varies according to the system's goals. For the inverted pendulum, the goal of the system was to provide for dynamic upgrades. Two controllers are available to balance the pendulum. The complex controller is allowed to be uploaded dynamically during system execution. This complex controller is used as long as the pendulum stays within a given safety envelope. Otherwise, the simple controller is used.

For the diving controller prototype, the goal was robustness. Developed for the United States Navy to demonstrate robust control in a distributed, component-based system, the diving controller uses quadruple-redundant Simplex architecture. The simple controller is robust but has lousy performance. The complex controller has better performance but at times produces faulty control commands.

Developers interested in using the Simplex architecture may likely be interested in its cost. In particular, what is the tradeoff between the benefits of the complex controller and the cost of the additional resources of the modeling machine, decision machine, and complex controller. This is difficult to predict, except empirically. Some analyses of cost have been performed [28], but only by experiments on individual prototypes.

5.1.1 The inverted pendulum

To demonstrate the ability of Simplex to support on-line upgrades, the inverted pendulum prototype was developed [3, 42]. Shown in Fig. 5.2, an inverted pendulum is composed of a rod (the pendulum), balanced on a cart that can move along a track. The cart's x-position must be constantly repositioned such that the pendulum remains balanced.

Problem frame

The inverted pendulum's problem frame is characterized by the plant, environment, sensors and actuators. The plant is the pendulum and the cart and the monitored variables are the angle of the pendulum and the x-position of the cart. The environment is the cart's track and the pendulum's surroundings. Sensors capture the x-position and the angle information of the pendulum. Two motors actuate the new x-position and angle according to incoming control commands.

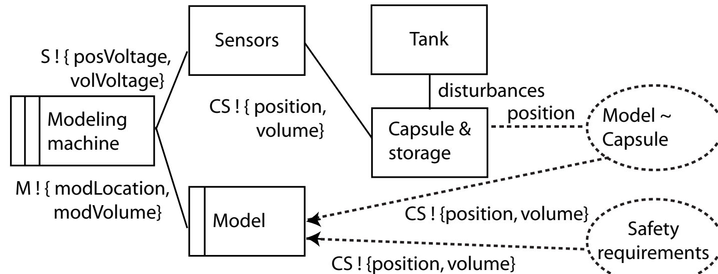
The problem is to build a machine that keeps the pendulum balanced. The requirements are:

- **Model safety requirements**. Provide a precise estimation of the pendulum angle and x-position. Based on the properties of the cart and track, determine if the pendulum is within one of two stability envelopes. The first stability envelope describes the maximum angle and x-positions that can occur before the pendulum falls. The second, the restricted stability envelope, places a tighter bound on these angle and x-position requirements. The model must use the estimation of the current state to identify if the pendulum is in either stability envelope. For the inverted pendulum, such stability envelopes are defined by "stability in the sense of Lyapunov" [10].
- **Core safety requirements**. The pendulum must stay within the stability envelope.
- **High-confidence safety requirements**. The pendulum must stay in the restricted stability envelope.

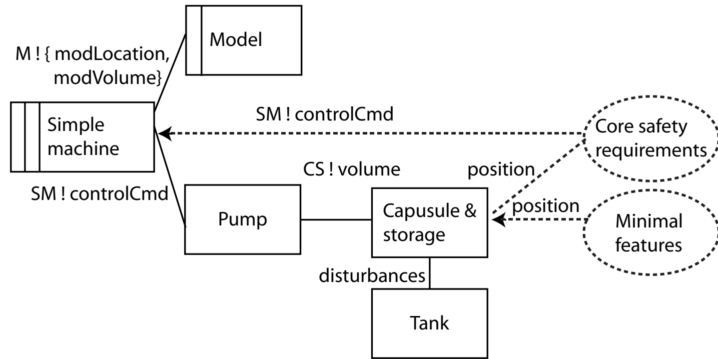
Architecture

The Simplex architecture for the inverted pendulum prototype matches exactly the architecture summarized in Fig. 5.1. The only matter to discuss is the modeling machine and the decision machine.

The implementation of the modeling machine for the inverted pendulum is grounded in control theory. It uses the domain description of both the plant and the environment − properties such as the mass of the cart and friction coefficients of the track − to identify the stability envelopes using the Lyapunov [50] calculation. Given that the modeling machine identifies these envelopes, the



1. Diving Controller Model Builder.



2. Diving Controller Core Requirements with Connected Domain.

Figure 5.3: The problem frames for the diving controller. 1. The diving controller model builder must keep track of the capsule position and volume of air in the storage tank. 2. The core requirements with connection domain problem frame associates the control command with the safe position of the capsule.

decision machine is very simple: if the pendulum is in the restricted stability envelope, use the complex control command, otherwise, use the simple control command.

5.1.2 The diving controller

To demonstrate the ability of Simplex to perform in a component-based, distributed control system, a diving controller prototype [71] was developed. The diving controller, whose physical prototype is summarized in Fig. 5.2, mimics the control necessary for a submarine to submerge and surface.

Problem frame

The diving controller's problem frame is characterized by the environment, plant, sensors and actuators. The environment consists of a 5-foot tall, cylindrical water tank. Vertically installed in the tank is a metal rod; the plant is a capsule, or "submarine," which slides up and down the rod. Whether or not the capsule submerges is determined by the amount of air pumped into it. Excess air, unnecessary when the capsule is fully submerged, is kept in a storage tank mounted to the top of the tank; air is transferred between the capsule and

storage tank via an air hose. Two sensors detect the position of the capsule and the volume of air in the storage tank. An air pump is used for actuation, either by pumping air out of the storage tank and into the capsule, or vice versa.

The problem is to build a machine that raises and lowers the capsule according to user input via a graphical user interface. The requirements are:

- **Model safety requirements**. The model must provide a precise estimate of the capsule position and volume of air in the storage tank.
- **Core safety requirements**. The capsule must maintain a position according to the last-received well-formed command from the user. The capsule must maintain this position despite flawed user commands, as well as software, hardware, and network faults.
- **High-confidence safety requirements**. The capsule must maintain a position according to the last-received well-formed command from the user.

Fig. 5.3 summarizes two of the three Simplex problem frames for the diving controller. The top of the figure shows the model building problem frame; the problem is to build a machine that can approximate the position of the capsule in the tank. This problem begins with the water tank which introduces disturbances into the plant. The plant comprises the capsule and air storage tank. The position of the capsule and the volume of air in the storage tank are the monitored variables. From the sensors, the modeling machine reads the voltage for the position and air volume which are then translated into the symbolic phenomena: the modeled position of the capsule and volume of the tank.

The bottom of Fig. 5.3 shows the first connection domain problem frame describing the requirements for the behavior of the capsule. The core safety requirements constrain the control commands of the simple machine against the position of the capsule. In other words, the control commands must always keep the position of the capsule in a safe state. For this prototype, safety means to follow the well-formed commands presented to the system by the user interface.

The second problem frame describing the desired features has been omitted from the figure. It differs only by the high-confidence requirements and the complex machine. Given that the high-confidence safety requirements make no statement about faults, any ill-formed commands from the user, or faults in the the communication between components, will result in the use of the simple machine.

Architecture

Fig. 5.4 summarizes the architecture of the diving controller prototype. Four redundant control systems are used, each one equipped with a simple and complex

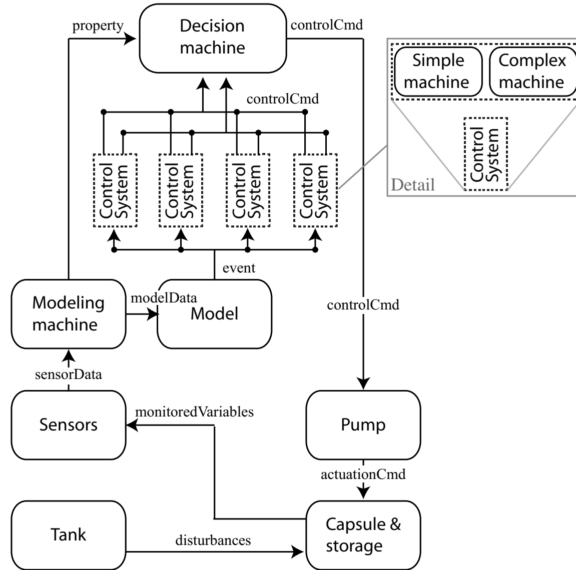


Figure 5.4: The Simplex architecture for the diving controller prototype. Four redundant control systems are used. Detail: Each control system is equipped with a simple and complex machine.

machine. The simple control algorithm used is robust, but has very low performance. The complex control algorithm has better performance, but sometimes produces faulty commands. Moreover, the decision machine is equipped with a third safety controller in case communication is severed between the control machines and the decision machine. This safety controller, again, performs low performance but robust control for the capsule.

5.2 Uncovering the necessary vocabulary for describing the Simplex architecture

Armed with this introduction to the Simplex architecture and two of its example prototypes, I return to the greater vision of this dissertation: an offering of Simplex described precisely enough so that developers may conduct an early analysis of their own domain-specific architectures. This description must support model-driven development, featuring testable precision and boundary delineation.

To provide such a Simplex model requires a description language with a vocabulary appropriate to Simplex; a good place to start is to understand what must be said about software architectures in general. "Software architecture" is one of those nebulous terms that are difficult to precisely articulate; moreover, the software engineering community does not exactly agree on a definition. One just has to pick a favorite. In their book [9], Bass, Clements, and Kazman define software architecture as follows,

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*

To expand this definition more concretely, a software architecture is a set of components about which the following is known:

- **Component Interfaces**: A well-defined means by which each component communicates or interacts with other components.
- **Connections**: A description of how individual components in the system are connected together.
- **Component Behavior**: The observable functionality of a component as far as it affects or influences other components.

Given this definition of software architecture, a language that describes the Simplex architecture must be able to express at least these three things: interfaces, connections, and behavior. Is that enough?

Recall the Simplex problem architecture defined in Chapter 4. The problem is to build a machine that controls some part of the physical world according to some minimum feature set such that safety is guaranteed. If safety requirements are met, then the machine may control the physical world according to some additional desired features.

When describing the solution architecture to this problem, developers must describe three sets of requirements. First, they must describe the model's safety requirements–or the accuracy and precision of the software's estimate of the physical world. Second, they must describe the core safety requirements by which the simple machine must maintain plant safety. Third are the high-confidence safety requirements for the complex machine.

Clearly, interfaces, connections and behavior are not enough. To describe or model the Simplex architecture, developers need a way of describing all of these safety requirements. The language for describing these requirements depends on the specific application. Simple applications are fine with English, others do not need anything more expressive than Linear Temporal Logic[1].

Interfaces, connections, behavior, safety. Is *that* enough?

Recall in Fig. 5.5 the Simplex architecture diagram. Notice the line that divides the software components from the physical world. Unlike enterprise systems or web applications, much of real-time software's behavior is dedicated to modeling or controlling the physical world. This line is the same "axis of symmetry" in Barry Rubel's *Pedastle* pattern for control systems shown in Fig. 5.6; it provides a boundary interface between the real world and the software world [62].

[1]Linear Temporal Logic, or LTL, will not always suffice. For a more comprehensive framework addressing a formal description of safety for Simplex, refer to Chapter 10.

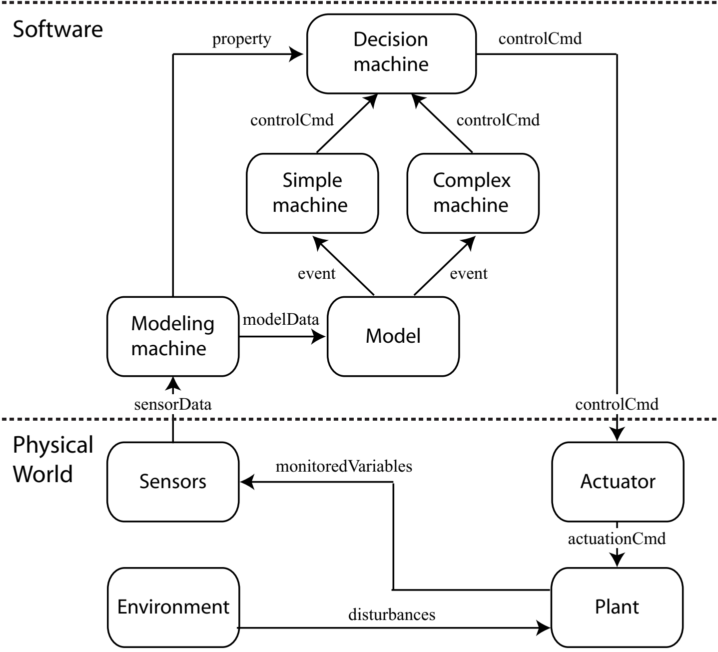


Figure 5.5: The Simplex Architecture

Because of the physical world's prominence in the Simplex architecture, a description language ought to differentiate the physical world from the software. By the same token, it is important to delineate the interface between the software and the physical world.

There is one last thing that's needed to effectively describe the Simplex architecture if it is going to be a reusable architecture: parameterization. One must be able to separate the generic parts of Simplex from the application-dependent ones. This way, there is no dangerous editing of information that doesn't need it.

Think about it: the keystone to Simplex is choosing between two alter-

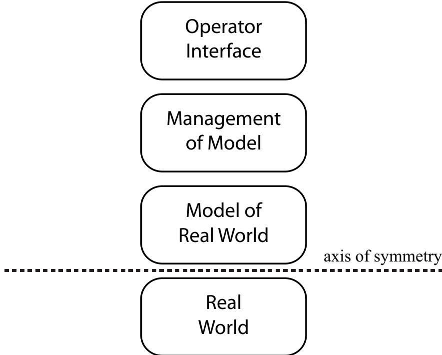


Figure 5.6: Barry Rubel's four layer architecture, or *Pedastle* design pattern, which highlights an axis of symmetry between the "Real World" and the software which models and controls that world.

natives. No matter the application −− diving controller, inverted pendulum, or F-16 fighter −− the architecture's behavior boils done to a simple if-then construct: *If the system has high confidence safety, then choose the complex command; otherwise, choose the simple one*. What it means to be a "system" or to have "high confidence safety" is very application-dependent; a description language should empower a developer to only fill in the necessary details without mucking around in the generic behaviors.

That's it. One needs six language constructs to describe a Simplex architecture:

- **Component interfaces**. Expose the abstracted functionality of the component as it may be accessed by other components.
- **Component connections**. Describe how components are connected via their interfaces.
- **Component behavior**. Express the functionality of the component as it may affect the functionality of other components.
- **Safety properties**. Express the properties that must always be true for a component or subsystem.
- **Hardware-software delineation**. Allow types which identify components in the physical world and the software one.
- **Parameterization**. Separate generic functionality from application specific functionality; identify what is required from other components.

How do they all fit within the three features set out at the beginning of this dissertation? Recall from [23] that real-time developers want:

- **Testable precision**. Support the development of precise and testable specifications for real time, mission-critical, embedded systems.
- **Consistency definition**. Define what makes a set of requirements consistent, including principles, guidelines, and techniques for determining if requirements are internally consistent and for keeping them internally consistent when they are changed.
- **Boundary delineation**. Support the delineation of system boundaries, the precise specification of system interfaces, and the description of the system's environment. Users may indicate where the specification is internally incomplete, although the tool must allow users to isolate fuzzy or incomplete requirements and proceed with work on requirements that are well understood.

Fig. 5.7 breaks the six language constructs into three categories. The first two categories are not new: testable precision and boundary delineation. The

Testable Precision	**Support precise and testable specifications.**
T.1. Component behavior	Express the functionality of the component as it may affect the functionality of other components.
Boundary Delineation	**Support precise specification of system interfaces; support description of the system's environment.**
B.1. Hardware-software delineation	Allow types which identify components in the physical world and the software one.
B.2. Component interface	Expose the abstracted functionality of the component, as it may be accessed by other components.
B.3. Connections	Describe how components are connected via their interfaces.
B.4. Parameterization	Separate generic functionality from application-specific functionality ; identify what is required from other components.
Safety Definition	**Define what makes an architecture safe for the complex controller and the simple controller.**
C.1. Safety properties	Express the properties that must always be true for a component or subsystem.

Figure 5.7: The six language constructs necessary to describe Simplex architectures

third is necessary to include in any definition of safety in an architecture description of Simplex. Of course, the perfect language does not exist. In an ideal world, a developer writes the perfect description language for modeling her domain. But research is never ideal; subsequent chapters evaluate three approaches to modeling Simplex and how they fare against these six language constructs.

6 A dot and a big table

What lays ahead are three chapters describing three different approaches to describing the Simplex architecture in a machine checkable format. In order to make a fair comparison of the three, I have created an example which is simple enough that it can be modeled by even the simplest of the three, but it is interesting enough to make a useful comparison.

This section details this simple example and summarizes the comparison of the three tools. Interested readers can go on to the next three chapters to get the details, but the punchline of my six years of graduate school is shown in Fig. 6.2.

6.1 The simple dot example

Consider an object, a dot, starting at $x = 0$ meters and moving in one dimension towards a wall located at 50 meters. The dot has one safety requirement; do not hit the wall. The dot has one functional requirement; get as close to the wall as quickly as possible.

Given this is Simplex, I separate the control of the dot into an unreliable complex controller and a trustworthy simple controller:

- **Complex controller**. Command the object to move towards the obstacle at its maximum velocity.
- **Simple controller**. Command the object to stop completely.

For the purposes of the modeling machine, I summarize what is known about the dot in its environment. The dot can move at a maximum velocity of 1 meter-per-second. For the purposes of this simple example, control commands are issued as a target velocity, v. The object executes these control commands with exact precision. The object has instant acceleration, but when traveling at its maximum velocity, its maximum breaking force yields a minimum deceleration rate of $\frac{1}{7}$ meter-per-second^2. Thus, it takes 3.5 meters to stop when moving at full speed. Sensor data provides perfect observations about the dot's location, expressed as a distance from the dot's origin.

The example may seem unrealistic, and it is. Instant acceleration? Perfect observations? Keep in mind though that my work is not about control laws or sensor precision. My work is about description and automated analysis of

Simplex architectures. I created the dot and the wall scenario because I wanted a concrete example that I could carve into the Simplex component-based architecture

At a high level, how should Simplex work for this example? The object starts at 0 meters, at a safe distance from the obstacle. Hence, the Simplex decision machine chooses the complex controller's output to move full speed towards the obstacle. As the object approaches the obstacle, and nears an unsafe distance, it enters a state that does not satisfy high confidence safety; Simplex must choose the simple controller output. Upon receiving each input, the CHKR must determine the following in order to choose the UN or TW output:

- What is the current position?
- Suppose the complex controller's command is issued, what is the estimate of the dot's position at the next time-step?
- Is the estimate of the dot's position a high-confidence safety state? That is, could a simple control command be issued at the next timestep that would ensure the dot does not hit the wall?

That is, the decision machine must examine the current state and "possible next" states of the dot to determine if the output from complex controller is recoverable. If so, then the decision machine chooses the complex controller's output. Otherwise, it chooses the simple command.

Fig. 6.1 shows the transition from choosing the complex controller to the simple one. Indicated by (1), the dot reaches position $x = 45$ meters at 45 seconds. Given the complex controller output, an estimate of the next state is made; the single possible next state is $x = 46$ meters. Since the dot has high-confidence safety, the complex control is selected. Indicated by (2), the object reaches position $x = 46$ meters at 46 seconds. This time, the decision machine chooses the simple controller and chooses it for subsequent outputs. Finally, indicated by (3), the dot is stopped safely at $x = 49.5$ meters.

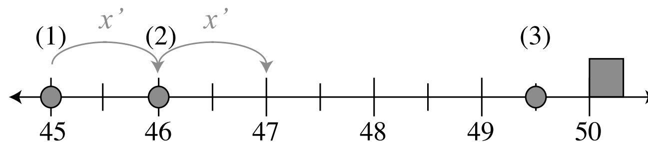


Figure 6.1: Transitioning from unreliable to trustworthy output.

6.2 Comparing three tools: Maude, AADL, A2M

I have precisely defined a collection of Simplex solutions in three different specification paradigms: Maude, AADL, and A2M. Maude is an executable spec-

ification language developed at SRI and the University of Illinois at Urbana-Champaign. AADL, or the Architecture and Analysis Description Language, is a standard architecture description language offered by the Society of Automotive Engineers. The third, A2M is an AADL interpreter implemented in Maude which offers a set of executable semantics for a small subset of AADL.

The table below summarizes my comparison of these three paradigms. Each approach is rank-ordered low, medium, or high. A high score indicates that the tool is the best among the three for the particular metric. Brave readers can continue with the next three chapters to uncover the details of my evaluation; I summarize it here for the reader's convenience.

Comparison	Maude	AADL	A2M
T.1. Component behavior	HI	LOW	MED
B.1. Hardware-software delineation	HI	MED	LOW
B.2. Component interface	LOW	HI	MED
B.3. Connections	MED	HI	LOW
B.4. Parameterization	HI	MED	LOW
S.1. Safety properties	HI	LOW	MED

Figure 6.2: A comparison of three tools for describing the Simplex architectures.

6.2.1 Maude

An executable specification language, Maude offers an expressiveness that gives developers significant power to describe models and execute them. With a really fantastic way for parameterizing modules, the ability to describe safety properties in linear temporal logic, and executable specifications, Maude stands out as the best approach among the three. Where it falls short is its lack of a domain-specific vocabulary; its rich expressiveness allows developers to describe almost anything, but this leads to an almost paralysis modeling systems. For more details, see Chapter 7.

6.2.2 AADL

The most industrial of the three, AADL is a dictionary of vocabulary evolved over many years by multiple teams of academics and industry partners. While its vocabulary is useful for describing component interfaces and connections, its lack of executability makes it difficul to do any kind of safety analysis. For more details, see Chapter 8.

6.2.3 A2M

An exciting joint project between the Real-Time Systems Integration and Formal Methods research groups at the University of Illinois, A2M offers the potential for an approach that has the apt dictionary of AADL but with the executability of Maude. At this time, it is only a prototype −− hence all the medium and low scores −− but as it is developed and adapts more of the AADL standard, I believe that this tool has the potential to offer developers a safety analysis tool with real utility. For more details, see Chapter 9.

7 Modeling Simplex in Maude

All kinds of word processing applications check spelling as I type. If I spell "farmar" instead of "farmer," a dashed red line underlines the word. Smarter applications advise on grammar rules. Given the phrase, "Message receive events," Microsoft Word suggests "*Messages* receive events." Spelling and grammar are the syntax of the English language; they are the rules by which one may structure words into a sentence.

Just like human languages, computer languages are defined by syntactic rules which define the structure of a program. Their dictionaries are smaller but their grammars are stricter. Every class in C++ must end with a semicolon. If a semicolon is missing, the C++ compiler gives an error; it may or may not be coherent. If I omit a period from the end of an English sentence, other people don't mind as much

There are a variety of computer languages available for a number of computable tasks. High-level imperative languages like C, C++ and Java have the vocabularies and structures necessary for a software developer to command a computer to execute the instructions necessary to play an .mp3 file, compute a least common multiple, or display a graphical interface. Other languages are functional; yet others are logical, while still others are descriptive.

The previous chapter outlined a very useful set of language constructs necessary to describe Simplex. So far, these constructs have been treated as the collection of words necessary to talk about Simplex: the necessary syntax. But there is more to a language than just its syntax, there is also its semantics: its meaning.

Semantics are important for one very big reason: *testable precision*. To conduct an early safety analysis of the Simplex architecture, one needs a description of the architecture in some computer language: one that can be read by other developers, but can *also* be evaluated by tools such as a model-checker.

Enter Maude. Maude is an executable specification language developed by José Meseguer of the University of Illinois, Steven Ecker and Carolyn Talcot at the Stanford Research Institute, and Manuel Clavel of Universidad Complutense de Madrid. Maude is based on rewriting logic which José Meseguer argues has, *"a simple and intuitive, yet precise, semantics. As a result, Maude has an expressiveness and ease of specification that allows for many different kinds of systems, and at different levels of abstraction"* [66].

I am certain not every developer would agree with this assessment, but

an executable specification language certainly has immense value. What does it mean for it to be executable? It is the same reason why Grigore Rosu's programming languages class always felt to me like an infomercial: *You get the interpreter for FREE!* That is, by defining a set of syntax and semantics in Maude, one has also implicitly defined an interpreter for the language or model. Define Peano arithmetic and with it comes a rudimentary calculator.

This chapter highlights my Maude Simplex model; it also evaluates Maude against the six metrics for Simplex description set out by Chapter 5.

7.1 A very brief introduction to Maude

To be clear, this dissertation should not be treated as a resource for learning Maude. For those interested in learning Maude there are a number of useful resources. My personal favorite is the "Maude Primer" written by a college student doing a summer internship at SRI [51]. The primer is direct, accessible and well-written, a well-received break from most academic gobbledeegook. There is also José Meseguer and friends' latest book, "All About Maude" [13]; it is an exhaustive description of the Maude language and all of its affiliated extensions and tools. Over 3 inches thick, it is also a weapon.

7.2 A Simplex model in Maude

I now describe the Simplex architecture for the simple dot example introduced in Chapter 6 as implemented in Maude. This model began as a pilot project to understand the minimum amount of information necessary to model-check Simplex in Maude. The focus was not on AADL architecture constructs like ports and components; the focus was on domain-specific knowledge necessary for analysis. As a result, the model is simple. I like it that way.

The fully executable Maude specification for the simple dot example is available in Appendix J as well as the Real-Time System Integration group's subversion code repository. The file name is `parameterized-dot.maude`. Rather than dump out the code here, allow me to give a brief tour.

I start the tour with how I parameterize the model. Ideally, a description language for Simplex should easily separate the generic from the application-specific. Maude does a beautiful job. This is done using a *theory*. In simple terms, a theory is just a set of statements that are true. For example, in the theory of linear algebra there is the statement: $x + y = y + x$.

Similarly, a Simplex theory is a statement of vocabulary. It is all the words I use to talk about Simplex and how they relate to each other: a system, a set of commands, a simple controller, a complex controller, an internal model of the physical world, an some boolean operator which evaluates if a particular complex command is allowed:

```
--- Theory for the Simplex Machine.  The Simplex Machine
--- Consists only of the simple controller, the complex
--- controller, and the checker.
fth SIMPLEX-MACHINE-TH is
  protecting BOOL .
  sort System .
  sort Command .

  --- Simple and Complex controllers
  op simple-controller : System -> Command .
  op complex-controller : System -> Command .

  --- On-line estimate of the system.  Maintain a system
  --- estimate after issuing the command.
  op step : Command System -> System .

  --- Checker
  op cmd-allowed? : Command System -> Bool .
endfth
```

Using the Simplex theory, I can parameterize the Simplex behavior. The key to the Simplex architecture's behavior is the decision machine. Based on the properties delivered by the modeling machine, it has to determine if the current plant state satisfies the high-confidence safety requirements. If so, it is free to choose the control command from the complex machine. Otherwise, if the plant state satisfies only the safety requirements, it must choose the control command from the simple machine. This functionality is generic to all implementations of Simplex, whether a diving controller, an inverted pendulum, or an F-16. This all boils down to a simple conditional equation seen below:

```
mod SIMPLEX-MACHINE{X :: SIMPLEX-MACHINE-TH} is

  op simple-controller : X$System -> X$Command .
  op complex-controller : X$System -> X$Command .

  op cmd-allowed? : X$Command X$System -> Bool .

  op simplex : X$System -> X$Command .

  var S : X$System .
  var Cmd : X$Command .

  ceq simplex(S) = Cmd
   if Cmd := complex-controller(S)
```

```
   /\ cmd-allowed?(Cmd, S) .
  eq simplex(S) = simple-controller(S) [owise].

  rl S => step(simplex(S), S) .
endm
```

So far, the model has no information about dots or walls; it is generic and can be used for modeling many applications. I define all of the dot-specific information in another Maude module, DOT-SIMPLEX-MACHINE.

For the simple dot example, a System is described by the position of the dot, the position of the wall, and the current velocity of the dot[1].

```
  --- System membership:  A well-founded system comprises a
  --- wall position (Wpos), car position, and a car velocity.
  --- Notice that a system configuration is made up either of
  --- assumptions about the environment or sensor observations.
  mb [wall-pos(WPos) ; car-pos(CPos) ; car-vel(Vel)]
     : Dot-System .
```

There are two possible Commands for the dot:

```
--- Possible commands for the controllers to execute.
  op accel : -> Dot-Command [ctor].
  op stop : -> Dot-Command [ctor].
```

The key to this module is the decision machine. It is implemented as dot-cmd-allowed?, a boolean predicate that takes a command and a system; it returns true if the command will not put the dot in harm's way.

```
--- The checker performs a ``lookahead'' to make sure that
  --- the command is allowed.  That is, given the maximum
  --- speed, can the car still decelerate in time to avoid
  --- colliding with the wall?
  op dot-cmd-allowed? : Dot-Command Dot-System -> Bool .
  eq dot-cmd-allowed?(Cmd, [car-pos(CPos) ; wall-pos(WPos) ; St])
   = CPos + (top-speed * top-speed) / (2 * - max-decel) < WPos .
```

With the semantics of dot-specific information defined, I use a Maude view to map the dot implementation to the generic parameters of the theory.

```
   --- Map the particular implementation
   --- of the dot simplex machine to the
   --- general simplex machine theory.
   view Dot from SIMPLEX-MACHINE-TH to DOT-SIMPLEX-MACHINE is
```

[1]In some cases, the "dot" is referred to as a "car" in the code, as in car-pos or car-vel. Terrible coding practice, but it is what it is.

```
  sort System to Dot-System .
  sort Command to Dot-Command .

  op simple-controller to dot-simple-controller .
  op complex-controller to dot-complex-controller .
  op step to dot-step .
  op cmd-allowed? to dot-cmd-allowed? .
endv
```

7.3 Model-checking the Simplex specification

Model-checking is the determination that a model satisfies a property. Thus, there are two input to model-checking: a system specification and a property specification.

With Maude comes the incredibly valuable ability to model-check a specification. I want to make sure that the system I have specified is safe: the dot must never hit the wall. I want to check that the dot position is always "before" or less than the wall position. First, I must define the property:

```
op before-wall : -> Prop .
eq [car-pos(CPos) ; wall-pos(WPos) ; Cfg] |= before-wall
= CPos < WPos .
```

Then I check the property against the model:

```
red modelCheck(init, [] before-wall) .
```

Maude yields a positive result.

```
reduce in DOT-MODEL-CHECK : modelCheck(init, []before-wall) .
rewrites: 194 in 0ms cpu (9ms real) (~ rewrites/second)
result Bool: true
```

7.4 Evaluation of modeling Simplex in Maude

7.4.1 Testable precision

Component behavior

It is not enough to describe just the components and connections of a software architecture. Boxes and lines are not telling. A software architecture description also needs behavior information for testable precision.

The fact that Maude is an executable specification language has immense value. By defining a set of syntax and semantics in Maude, one has also implicitly defined an interpreter for the language or model. Moreover, one can use

the Maude model-checker to determine if a given model satisfies a particular property. This is useful for the goals of automating the analysis of Simplex architectures, since it is necessary to show that a particular architecture maintains safety: the dot does not hit the wall, the pendulum does not fall, the car does not collide.

7.4.2 Boundary delineation

Hardware-software delineation, interfaces, and connections

For Maude, it does not make sense to discuss these three metrics individually: hardware-software delineation, component interfaces, and component connections. The same discussion will apply to each. I group them together in this section.

Return to José Meseguer's statement: Maude has, *"a simple and intuitive, yet precise, semantics. As a result, Maude has an expressiveness and ease of specification that allows for many different kinds of systems, and at different levels of abstraction"* [66].

Maude *is* expressive. But expressiveness and ease of specification do not naturally go hand in hand. This is because of human psychology. There is a myth that "the more choice the better" Yet researchers have empirically demonstrated just the opposite. Some of the first to do so were Sheena Iyengar and Mark Lepper. They asked, "What happens when the range of alternatives becomes larger and the differences among options become relatively small?" [37].

To answer this question, they went where people are faced with hundreds of choices: a grocery store in Menlo Park, California. They set up two tables featuring gourmet jellies: one table for each day. The first table featured 6 exotic gourmet jellies: from kiwi to lemon curd. The second featured 24. Almost 30% of shoppers who looked at the limited selection table bought some jelly. Only 3% of shoppers bought jelly from the extensive table [37].

This study was conducted in 2000, and subsequent research has demonstrated a similar theme; too much choice is paralyzing. How do I pick one mustard out of the 250 mustards offered at the grocery store? Maude is expressive; it is the grocery store of specification languages. It can do anything; it is paralyzing.

Maude does not offer a specific vocabulary for modeling hardware-software delineation. Yet because Maude is so expressive, it is possible to specify these things. One needs only to define the semantics for hardware and software interfaces, communication channels, and their integration. It can be done; it is just hard to know where to begin.

Parameterization

Parameterization is where I feel Maude shines. With its theory and view mechanisms, I can define any collection of concepts, parameterize a module using

that collection, and map a specific implementation to the collection. For Simplex, these are concepts like a `Command`, a `System` or the boolean predicate `cmd-allowed?`. Then I can use that theory to parameterize another module which defines the behavior.

I complained about Maude's expressiveness in an earlier section. I do not want to define my own semantics for communication channels and hardware interfaces before I begin work on specifying my own architecture. However, the expressiveness offered by the theory-view concept is incredibly useful. I can parameterize a module at whichever granularity that seems appropriate for the domain. I do not have to extend a collection of components one at a time using some arcane inheritance mechanism; I get to choose that Simplex is parameterized by a collection of concepts.

7.4.3 Safety definition

Safety properties

The model checker offered by Maude is a linear temporal logic model checker. Linear temporal logic, or LTL, is a temporal logic in which one may describe properties of a single path of execution. Thus, any safety property that can be expressed in terms of LTL may be checked by the Maude model checker. For the simple dot example LTL is plainly adequate. I want to check that globally along the path of execution the dot does not hit the wall.

7.5 Conclusion

My favorite kind of jelly is blackberry. If I were to go to a grocery store, and the only kind of jelly that they carried was *Simplify* brand blackberry pecan jelly, I would buy three jars. I eat a lot of peanut butter and jelly sandwiches given my graduate stipend. But if I were to go to a grocery store that carried anywhere from six to 250 kinds of jelly[2], but none of them were blackberry, I would likely not make a jelly purchase at that particular store.

There is a tradeoff between limited and extensive choice. Similarly, there is a tradeoff between expressiveness and ease of specification. Maude is expressive; it is an expressiveness that is sometimes paralyzing. Other languages may not be as expressive, but they could facilitate easier specifications given the right vocabulary. Still other, less appealing, languages may not be as expressive; they may not have the right vocabulary. They may not have any blackberry jelly.

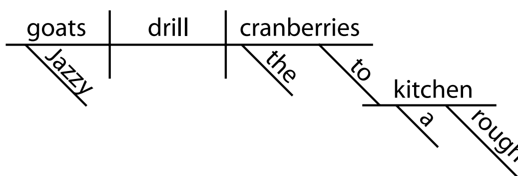
Despite this expressiveness challenge, Maude offers a lot for modeling Simplex architectures: executability and fantastic parameterization.

[2]This isn't crazy-talk. The grocery store in Menlo Park discussed in [37] carried over 250 kinds of mustard

8 Modeling Simplex in AADL

One candidate language considered by Lui Sha's research group for describing Simplex is the Architecture Analysis & Design Language Standard, commonly referred to as "AADL." Initiated by Sha's colleagues Jun Li and Peter Feiler in 1999 [45], this architecture description language standard was adopted by the International Society of Automotive Engineers in 2004. The language was created so that real-time and embedded systems designers had a more formal way to design and analyze software systems in a machine checkable format.

The key thing to note about AADL is that it is *just syntax*. It is a dictionary of words with an accompanying structure. If I write the sentence, "Jazzy goats drill the cranberries to a rough kitchen," no word processor would complain. It fulfills all of the syntactical rules of English. I can even diagram it:



But what is the meaning? How do goats drill and what makes them jazzy? The eerie feeling that I get from writing this weird and meaningless sentence is similar to the feeling I get when I am writing architecture descriptions in AADL. Because AADL is not executable−−because it has no compiler or formal semantics−−I am not at all sure if the architecture I have described makes any sense. While it is easy for human beings to spot weird sentences like mine about jazzy goats, its difficult to spot weirdness in the sometimes bulky AADL.

Back to Maude, the executable specification language developed by UIUC's own José Meseguer and others. Work is currently underway to define the formal semantics of AADL in Maude. With that in place, AADL specifications have both syntax and semantics, and tools like the Maude model-checker can be used to verify safety properties.

This chapter focuses on the vocabulary of AADL; how well AADL suffices against the six metrics for a language to describe Simplex. Chapter 9 describes current plans to define a formal semantics for AADL and to automatically model-check instances of Simplex described in AADL using the Maude model-checker.

8.1 A very brief introduction to AADL

To be clear, this dissertation should not be treated as a resource for learning AADL. For those interested in learning AADL there are a number of useful resources. Depending on taste, one might begin with the AADL standard [70] published by the Society of Automotive Engineers who generously sell it for $59.00. The folks at the Software Engineering Institute also provide a getting started guide [26] packed with more prose and more examples. Instead, this dissertation offers a brief introduction to AADL, sufficient to understand this chapter's discussion. The AADL code for the example in this section can be found in Appendix B.

8.1.1 History and motivation

AADL began humbly as the "Avionics Architecture Description Language" [2]; it was influenced by MetaH and evolved into AADL from a collaboration between Boeing, Honeywell, and the United States Army. The avionics domain sees very complex systems, and AADL's inventors wanted a modeling language to describe the architectures of these hard real-time, safety-critical embedded computer systems [2, pg 1].

One important characteristic the inventors wanted to support was the rather unique life cycles seen in the avionics domain. An airplane's life cycle is long, both in terms of its design and maintenance. The Boeing 747 was designed in 28 months, first deployed in 1970 and is still in flight today [74]. As a result, the key feature of AADL is partial specification. Over a lengthy design stage, not everything may be initially known about its various components; still, developers want to evaluate even their partial designs. Moreover, these "systems exist in many forms, such as instances of a system deployed in different contexts or a system evolving over time" [25, pg v]

However, as the language evolved and became a standard, the next generation of AADL language designers at the Software Engineering Institute chose to generalize it and market it for a wider audience; they renamed it the "Architecture and Analysis Description Language" [70] and offer it as a language to describe any component-based embedded system.

8.1.2 AADL core standard

Developers using AADL can model their systems as a collection of components. Components are categorized into the following: application software, hardware execution platform, and composite. These three span the following 10 subcategories which comprise the "core vocabulary" of the description language [24].

- **Application software**: thread, thread group, process, data[1], subprogram.
- **Hardware execution platform**: bus, memory, device, processor.
- **Composite**: system.

Remember, the key feature of AADL is partial specification. As a result, there are two separate descriptions of a component: its type and its implementation. The component type specifies the external interface of the component, expressed in terms of ports, subprograms, flows, and properties. Predefined properties are available−−such as a thread's `Execution_Time` or a flow's `Actual_Latency` [70]−−but new properties can be introduced into the language. Once a type is described, one may also specify one or more implementations of that type. The component implementation specifies the internal structure of a component.

Suppose, for example, one wanted to use AADL to describe a control loop architecture consisting of sensor, controller, and actuator. For the sensor, one would choose the `device` component category. A very generic component type declaration would describe an input port which reads the voltage value and an output port which reports the sensor data.

```
data voltageValue
   properties
      Source_Data_Size => 16 bits;
end voltageValue;

data sensorData
   properties
      Source_Data_Size => 16 bits;
end sensorData;

device dvcSensor
    features
      Input : in event data port voltageValue;
      Output: out event data port sensorData;
end dvcSensor;
```

Given this generic type declaration, one could then have multiple implementations of a sensor device. For example, one could have a temperature sensor that outputs ambient temperature along with units of measure. Expressing this can be done by creating an implementation of the `sensorData` type.

[1]Like Roy Fielding et. al. in [27], data is treated as a key aspect in an AADL system, rather than encapsulated and hidden within objects in the architecture

```
data implementation sensorData.temperature
   subcomponents
      temperature : data int;
      measurementUnits : data string;
end sensorData.temperature;
```

AADL also offers an `extends` mechanism that allows developers to define partial specifications or families of components: similar to what abstract classes provide in Java. This mechanism can be used either at the component type or component implementation level. Going back to the `dvcSensor` type declaration, one could let this component type represent a family of sensors. Every sensor in this family has an input and an output, but some are more specialized. One may use the `extends` mechanism to create a subtype, defining those sensors that also have an output that indicates an internal error.

```
device dvcSensorWithError extends dvcSensor
     features
        InternalError: out event data port errorData;
end dvcSensorWithError;
```

The AADL component implementation also specifies the component's connections to other components via ports. Ports are logical connection points which can pass data, events raised by subprograms, or both [70, pg 96]. There are three kinds of ports: `data`, `event` and the dubious `event data` port which provides an interface for message transmission with queuing [26, pg 56].

Suppose that a developer needs to connect the sensor component to a controller component. This is done via ports. First, consider a very simple control process with two ports:

```
process proController
   features
      controlInput: in event data port sensorData;
      controlOutput: out event data port controlCommand;
end proController;
```

Notice that because AADL has a global namespace, the names of the ports must differ; I cannot reuse the names `Input` and `Output`. To connect the sensor to the control, I can do so within an AADL system implementation, or what is also called a *system instance model*.

```
system sysControlLoop
end sysControlLoop;

system implementation sysControlLoop.simple
   subcomponents
```

```
      mySensor: device dvcSensor;
      myController: process proController;
   connections
      event data port
        mySensor.Output -> myController.controlInput;
end sysControlLoop.simple;
```

To organize an AADL specification, one has two alternatives. One can place the entire specification in a single file. Or one may use the `package` construct, which is a "named grouping of declarations and property specifications" [26]. Every package must be contained in its own file, and every package has its own namespace. The simple sensor example that has been presented here is contained in its own package in Appendix B:

```
package pkgSensor
   ...
end pkgSensor;
```

Other packages that may want to make use of the sensor specification use a double-colon construct. For example, suppose in another package, a developer extends the `dvcSensor` component type:

```
device mySensor extends pkgSensor::dvcSensor
   ...
end mySensor;
```

8.1.3 AADL behavior annex

Looking over the AADL standard, it seems to me that the developers of AADL might not exactly agree on my favorite software architecture definition from Bass and friends: connections, interfaces, and behaviors. While the key feature was partial specification, the original SAE standard had no way of describing component behavior. In 2004, an AADL specification simply amounted to components and their connections: boxes and lines.

Fortunately, AADL is extensible via annexes. An annex enables a user to extend the AADL language, allowing the incorporation of specialized notations within a standard AADL model [26, pg 10]. Any developer can write an annex to describe anything one desires, but some particular annexes have been adopted into the standard. One is the behavior annex. In 2006, French researchers at Ellidiss Technologies, FéRIA, and ENST de Bretagne partnered with Airbus France to develop and publish the behavior annex [57]. This annex allows one to describe thread and subprogram execution behavior using a Mealy finite state automata description language.

Behavior is described by a set of guarded state transitions. Upon a state transition, an action may take place. The syntax is:

```
<state id> -[ <guard> ]-> <state id> { <action> };
```

To continue our example, suppose that our temperature sensor can be modeled with a two-state finite automata. It has a “detect” and a “report” state.

```
data implementation sensorData.temperature
   subcomponents
      temperature : data int;
      measurementUnits : data string;
   annex behavior_specification {**
      state variables
           voltageValue : Behavior::boolean ;
           temperatureValue : Behavior::integer ;
      states
           detect : initial state;
           report : state;
      transitions
           --- If there is a value on the Input port, set
           --- the state variable equal to the value on
           --- the port.
           detect -[ on Input ? ]-> report
              { voltageValue := Input; };

           --- Translate the voltageValue into the
           --- temperature and send on the output port.
           report -[  ]-> detect
              { temperatureValue := voltageValue * 3 ;
                Output ! (temperatureValue);  };
   **};
```

8.2 A Simplex model in AADL

I now describe the generic Simplex architecture and the instantiation of that model for the simple dot example. The model is summarized in Fig. 8.1 using the graphical notation prescribed by the Software Engineering Institute.

I have delineated the two parts of the architecture using the AADL `system`. On the top of Fig. 9.1 is the software system, `sysSoftware`, which contains the modeling machine, the two controllers, and the decision machine. At the bottom of Fig. 9.1 is the hardware system, `sysRealWorld`, which consists of the physical plant and devices: the sensor and actuator.

The generic Simplex architecture specification is available in Appendix D as well as the Real-Time System Integration group’s subversion code repository. The instantiation of the generic architecture for the simple dot example is avail-

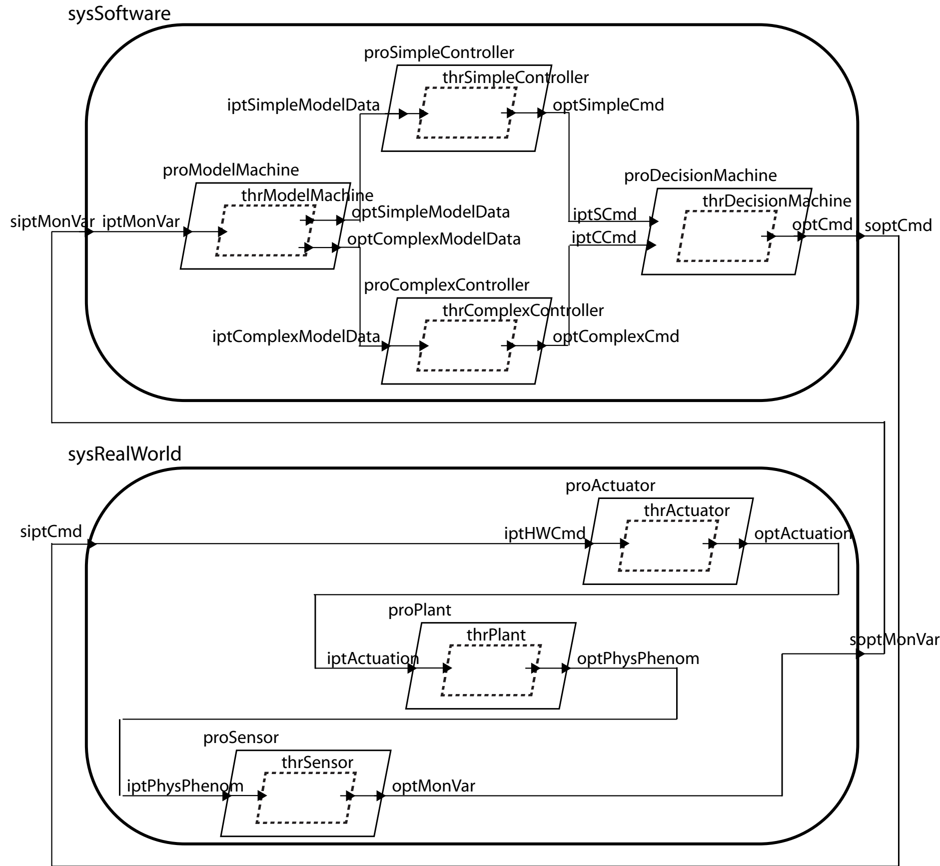


Figure 8.1: The architecture model for Simplex in the subset of AADL provided by the A2M Interpreter. A rounded rectangle represents a system. A solid parallelogram represents a process. A dashed parallelogram represents a thread. An arrowhead represents an event data port. Only two threads in the system are periodic: the `thrSimpleController` and the `proComplexController`. The remaining threads are aperiodic dispatch, reacting only to the messages they receive.

ble in Appendix E as well as the repository. Rather than dump out the code here, allow me to highlight a couple of noteworthy points.

The software system is the generic Simplex architecture. It provides generic definitions for a modeling machine, decision machine, and controller. It defines all of the components and their connections, but defines no behavior. For example, the component type for a very basic controller is just an input port and an output port.

```
package pkgController
public
  -- The Controller accepts model data from
  -- the model builder and issues a command.
  process proController
    features
```

```
      iptModelData: in event data port ;
      optCmd: out event data port ;
  end proController;
end pkgController;
```

A simple controller just extends this basic package:

```
process proSimpleController extends
          pkgController::proController
end proSimpleController;
```

For the developer who wants to instantiate the generic Simplex architecture, four parts must be specified:

1. Behavior for the modeling machine thread.
2. Behavior for the two controller threads.
3. Behavior for the decision machine thread.
4. The hardware system.

To instantiate the generic Simplex architecture requires use of the AADL `extends` construct. For the simple dot example, to define the behavior for the modeling machine thread first requires extending the component type defined by the generic specification. Essentially, the developer must use port refinement to define the types of the ports:

```
thread thrDotModelMachine
       extends simplex::thrModelMachine
 features
    ipthrMonVar :
       refined to in event data port Behavior::integer ;
    opthrSimpleModelData :
       refined to out event data port Behavior::integer ;
    opthrComplexModelData :
       refined to out event data port Behavior::integer ;
end thrDotModelMachine;
```

From there, I define the thread implementation as usual, specifying the behavior of the modeling machine thread using the behavior annex. The specification for the behavior of the controllers and decision machine is similar; developers must extend the component type and then define the component implementation.

For Simplex applications, the entire hardware system is domain-specific, and left up to the developer to define. To aid with specification, I have defined a collection of generic packages for actuator, sensor, and plant, as shown in Appendix C.

8.3 Evaluation of AADL for describing Simplex

I return to the six requirements for describing Simplex, summarized in Fig. 8.2. For each requirement, this section discusses what AADL provides such that it is good enough; it also describes what is missing and why it is not absolutely great.

AADL Evaluation

Testable Precision	Support precise and testable specifications.	Score
T.1. Component behavior	The behavior annex describes thread and subprogram behavior using a finite state machine description language.	**D**
Boundary Delineation	Support precise specification of system interfaces; support description of the system's environment.	Score
B.1. Hardware-software delineation	Provides three categories: application software, execution platform, and composite.	**D**
B.2. Component interface	The component type declaration specifies the component's external interface: ports, subprograms, flows, and properties.	**B**
B.3. Connections	The component implementation specifies the component's connections to other components via their ports.	**B**
B.4. Parameterization	Offers an extends mechanism; developers can define partial specifications and add details to inheriting components.	**C**
Safety Definition	Define what makes an architecture safe for the complex controller and the simple controller.	Score
C.1. Safety properties	A component's type declaration specifies its properties. Predefined types are available; developers may add new ones.	**D**

Figure 8.2: The six requirements for describing Simplex, and how AADL suffices.

8.3.1 Testable precision

Ideally, AADL ought to support the development of precise and testable specifications for real time, mission-critical, embedded systems. For describing Simplex architectures, I evaluate how well AADL describes component behavior.

Component behavior

It is not enough to describe just the components and connections of a software architecture. Boxes and lines are not telling. A software architecture description also needs behavior information for testable precision.

Why it's good enough. The AADL behavior annex allows description of thread and subprogram behavior using a finite state machine description language.

Why it's not great. The Simplex architecture boils down to a simple if-then construct which is very easy to express as a parameterized conditional equation in straight Maude:

```
ceq simplex(S) = Cmd
   if Cmd := complex-controller(S)
   /\ cmd-allowed?(Cmd, S) .
   eq simplex(S) = simple-controller(S) [owise].
```

These four lines express the following: choose the complex command if the command is "allowed"; choose the simple command otherwise[2].

Mimicking a parameterized if-then construct using only guarded transitions is clunky; to do so demonstrates the AADL's sometimes unnecessary bulk. The following are the state transitions taken from the simple dot example instantiation of the Simplex AADL decision machine:

```
transitions
   --- Consume any incoming complex command.
   --- Conduct the safety check on the incoming
   --- complex command.  Assess how far from a starting
   --- point of 0 the dot is, plus a safety envelope.
   sReceive -[ ipthrComplexCmd ? (inCCmd) ]-> sReceiveSimple
      { complexRcpt := complexRcpt + 1 ;
distanceFromStart := dotPos + inCCmd + safetyEnv ; } ;
   --- Consume any incoming simple command
   sReceiveSimple -[ ipthrSimpleCmd ? (inSCmd) ]-> s2
      { simpleRcpt := simpleRcpt + 1; } ;

   --- Check if the complex command is safe and
   --- send it if it is safe.
   s2 -[ on (distanceFromStart < wallPos) ]-> s4
      { dotPos := dotPos + inCCmd ;
        opthrCmd ! (inCCmd); } ;

   --- Double-check that the plant is still safe.
   s4 -[ on (dotPos < wallPos) ]-> sReceive { aux := 0 ; } ;

   --- If the plant is no longer safe,
   --- then the system has failed.
   s4 -[ on (dotPos >= wallPos) ]-> fail {aux := 0 ; } ;

   --- If the complex command is not safe,
```

[2]In case you skipped it, dear reader, there is more discussion on the parameterized Maude model of Simplex in Chapter 7.

```
   --- send the simple command
   s2 -[ on (distanceFromStart >= wallPos) ]-> s3
      { dotPos := dotPos - inSCmd ;
        opthrCmd ! (inSCmd) ; } ;

   --- Double-check that the plant is still safe.
   s3 -[ on (dotPos < wallPos) ]-> sReceive { aux := 0 ; } ;

   --- If the plant is no longer safe,
   --- then the system has failed.
   s3 -[ on (dotPos >= wallPos) ]-> fail {aux := 0 ; } ;
**};
```

8.3.2 Boundary delineation

Ideally, AADL ought to support the delineation of system boundaries, the precise specification of system interfaces, and the description of the system's environment. Users may indicate where the specification is internally incomplete, although the tool must allow users to isolate fuzzy or incomplete requirements and proceed with work on requirements that are well understood.

Hardware-software delineation

Simplex architecture descriptions would benefit from a description language providing categories which identify components in the physical world and the software one. This is because real-time and embedded systems very often control and monitor the physical world.

Why it's good enough. AADL provides three categories of components: application software, execution platform, and composite. As a result, developers can describe devices such as sensors, or software components such as threads.

Why it's not great. As I mentioned in Section 4.3.1 when discussing this family of CPS problems, it is important for Simplex developers to make a distinction between the plant and the environment. The plant consists of the physical devices under the machine's control; the environment consists of disturbances. Because of AADL's limited categories, developers ought to model a system's environment as a ... `device?`. A `process`? Unintuitive. Moreover, with the exception of the clunky behavior annex, there's no obvious way to describe how to maintain an internal model of the plant and environment; there's no good way to describe the solution to the model builder problem introduced in Section 4.2.4.

Moreover, larger real-time systems, such as avionics, cannot be cleanly divided into "application software" and "execution platform." For example, a

flight control system has a control application potentially running on top of some middleware, on top of some OS, on top of some hardware, in the house that Jack built. It is true that a single developer or development team might not work at all of these layers; they may not wish to specify all the details of these levels. Even so, there is no natural vocabulary in AADL for describing layered components.

Instead, developers have to make careful use of the `extends` mechanism, where an application component extends a middleware component. This means that a developer describing an application may very well have to look into the implementation of a middleware component in order to know what to extend. This is not good practice. Work done by Vergnaud and others [78] describes a loose mapping from middleware design to an AADL description: AADL packages, for example, map to reactive components in middleware. Their discussion is intriguing but the paper offers no enlightening examples.

Finally, not much of interest can be said about an AADL `device` from an application standpoint. While it might make sense to model an actuator as a device, the AADL vocabulary limits me to talk about only its ports and its properties. As a result, I have modeled all of the hardware for the Simplex architecture as processes and threads. In this way, I can use the behavior annex to describe their function.

Component interface

According to my chosen architecture definition, it is necessary to describe the abstracted functionality of one component insofar as it can be accessed by other components.

Why it's good enough. The component type declaration specifies the component's external interface in terms of ports, subprograms, flows, and properties.

Why it's not great. AADL differentiates between a component's type and its implementation such that, "a component implementation declaration specifies implementation-specific property values" [25]. However, the divide between what is defined in the component type and the implementation is weird; it does not correspond to what a developer must know to use a component without looking at the implementation.

In his book, *Domain Driven Design* Eric Evans talks about the necessity of "Intention-Revealing Interfaces" [22]. He writes, "If a developer must consider the implementation of a component in order to use it, the value of encapsulation is lost." Returning to the sensor example, the data type declaration for the `sensorData` reveals only the size of the data:

```
data sensorData
   properties
```

```
      Source_Data_Size => 16 bits;
end sensorData;
```

This is not exactly useful for the developer who must integrate this sensor component with his own. Instead, the developer must dig down into a particular component implementation to learn that sensor data consists of a temperature measurement and its units:

```
data implementation sensorData.temperature
   subcomponents
      temperature : data int;
      measurementUnits : data string;
   end sensorData.temperature;
```

However, it is possible to provide a more "Intention-Revealing Interface" in AADL by describing subprograms in a data's type declaration. For the sensor example, a developer may specify the subprograms that are called on `sensorData`:

```
subprogram getTemperature
end getTemperature;

subprogram getUnits
end getUnits;

data sensorData
   features
      getTemperature: subprogram getTemperature;
      getUnits: subprogram getUnits;
   properties
      Source_Data_Size => 16 bits;
end sensorData;
```

As a result, developers using the sensor package can gleam from just the `sensorData` type declaration that it consists of a temperature measurement and its accompanying units. But the particular data shared cannot be known without looking at the implementation.

Component connections

Again, according to Bass and friends' definition of software architecture, a good description language must describe how components are connected via their interfaces.

Why it's good enough. The AADL component implementation specifies the component's connections to other components via ports.

Why it's not great. First, port refinement means that port data types do not have to be named. Given that AADL's intent is to describe architectures in such a way that automatic analyses can be performed, leaving out these kinds of details and deferring them to the implementation seems out of line with the goal. How helpful is any kind of safety analysis if one cannot even perform type checking?

Fortunately, the behavior annex practically forces the developer to type her ports. Consider the following state transition:

```
idle -[ input ? (msg) when msg = 1 ]-> wait
```

The guard `input ? (msg)` checks if there is a message on the port `input`. If so, it places the value of the message in the state variable `msg`. State variables' types must be declared in the annex. If the state variable type does not match the port type, the AADL parser give the rather dubious error:

```
incompatibe [sic] argument: edu.cmu.sei.aadl.model.component
```

Second, port connections can only be made in the implementation. This causes problems that have surfaced before. Like Evans writes, "If a developer must consider the implementation of a component in order to use it, the value of encapsulation is lost." Unfortunately, unlike with the component interface, there is no way around this issue; ports can only be connected in an implementation.

Third, the port semantics described by the behavior annex have side-effects on the automata. Return to the example transition:

```
idle -[ input ? (msg) when msg = 1 ]-> wait
```

The guard conducts two tests. First it tests if there is any message availble on the port `input`. If so, the input is placed in the state variable `msg`. Second, the guard tests if the value in the state variable is equal to 1.

What is the caveat? Suppose that the value at the port `input` is 5. Even though this value fails the second test, the state variable is still assigned the value 5. To me, this is an undesirable side-effect. Why should the state change if no transition is made?

Finally, there is no way to express "no message available" in the guards. Consider the first transition in the automata for the decision machine:

```
sReceive -[ ipthrComplexCmd ? (inCCmd) ]-> sReceiveSimple
  { complexRcpt := complexRcpt + 1 ;
    distanceFromStart := dotPos + inCCmd + safetyEnv ;
  } ;
```

For this transition, if there is a message at the input port for the complex command, transition to the next state. Ideally, if the complex command is not available, I would like to consume the simple command instead, and move onto the next state. There is no expression for "no message available"; one alternative is something like this:

```
sReceive -[ ipthrComplexCmd ? (inCCmd) ]-> sReceiveSimple {...}
sReceive -[ ]-> sReceiveSimple {...}
```

But implementations of the behavior annex are free to choose state transitions non-deterministically. Even if there is a message available at the port, an interpreter could still choose the second transition during a given execution of the automata.

Another alternative is to use less readable specifications: encoding a program counter into the automata so that the appropriate transition is taken whether there is or is not a message at the port. Equally unsatisfying.

Parameterization

Why it's good enough. AADL offers an `extends` mechanism with which developers can define partial specifications and add details to inheriting components. For Simplex, developers need to extend four component type definitions and then define a particular component implementation for each one.

Why it's not great. Visually, I would like to be able to parameterize the Simplex architecture in this way:

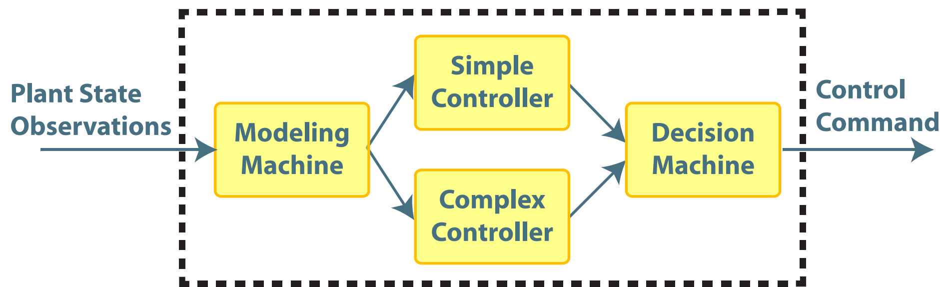


Figure 8.3: In the ideal case, parameterizing Simplex means clearly identifying the collection of concepts that must be defined for the specific application.

Simplex has a generic assembly with a collection of concepts that must be defined for the specific application. In the specification, I want to make it very obvious what is the *collection* of components that make up the architecture. With AADL, this is not possible. Because the only parameterization construct is `extends`, there is no way to group a collection of concepts as can be done so beautifully in Maude. Developers can only extend individual components one at a time; aside from putting all the concepts in a single file, there is no way to make this collection explicit. As a result, concepts that ought to be defined for the specific case could be easily forgotten.

8.3.3 Safety definition

The key goal of Simplex is safety; a Simplex architecture is defined by three sets of safety requirements:

- **Model safety requirements**. The model is required to provide an estimation of the real world within a specified tolerance.
- **Core safety requirements**. The simple machine is required to maintain plant safety.
- **High-confidence safety requirements**. The complex machine is required to maintain high-confidence plant safety.

And it is the Simplex architecture itself that teaches developers the principles and guidelines for keeping these requirements consistent using the decision machine, the complex controller, and the simple controller.

Safety properties

When describing the solution architecture for Simplex, developers must describe three sets of requirements: system safety requirements, core safety requirements, and high-confidence safety requirements. To describe or model the Simplex architecture, developers need a way of describing all of these safety requirements.

Why it's good enough. A component's type declaration specifies its properties. Predefined property types are available; developers may add new ones.

Why it's not great. So far, Linear Temporal Logic has been mentioned as a useful way to describe safety properties in a Simplex architecture. However, one may only use an LTL expression in one of two ways in AADL. First, developers can add any property they like to a specification; LTL would be most conveniently expressed as an `aadlstring`. For example, in the file `srmSafetySet`, I add the new property `CoreSafetyRequirement`:

```
property set srmSafetySet is
   CoreSafetyRequirement: aadlstring applies to (all) ;
end srmSafetySet;
```

Then, in the specification for the Simplex decision machine for the dot example, I describe this property as a string:

```
thread implementation thrDecisionMachine.dotExample
   properties
      srmSafetySet::CoreSafetyRequirement =>
         ''[] (CPos < WPos)'' ;
   annex behavior_specification {**
      state variables
 CPos : Behavior::integer ;
 WPos : Behavior::integer ;
         ...
end thrDecisionMachine.dotExample ;
```

However, using a string in this way decouples parsing the LTL expression from the AADL specification. This means that I could write a "correct" specification −− one that can be parsed by the AADL parser −− but contains an ill-formed LTL expression.

The only other alternative is to use an annex to define this collection of safety properties. For this, the developer must write her own parser. Very unsatisfying.

8.4 Conclusion

Despite its many drawbacks, AADL is one of the best among the available description languages for modeling Simplex. Unlike other description languages, like UML and RT-UML, which focus on modeling object-oriented software at the class level, AADL uses a more natural vocabulary for the real-time domain. AADL Words like *component* and *device* are a better fit than RT-UML's *ConcurrentUnit* and *SchedulableAction*.

Moreover, design decisions in research projects are not solely based on merit. Take a look at the history of two of the project's collaborators: Lui Sha and Peter Feiler. In a previous lifetime, Peter Feiler worked as Lui Sha's project supervisor during his time at the Software Engineering Institute. Collaborators are often difficult to secure in research, so many tend to stick with who they know. Sometimes−−more often than the naive graduate student might think−−research decisions are based on human relationships.

9 The AADL2Maude toolchain

Felix Ungar: I put order in this house. For the first time in months, you're saving money. You're sleeping on clean sheets. You're eating hot meals for a change and I did it.

Oscar Madison: Yes, you did. And after we've had your halibut steak and tartar sauce, I get to spend the rest of the evening watching you Saran Wrap the leftovers.

Neil Simon, *The Odd Couple*. 1965.

The Architecture and Analysis Description Lanugage, or AADL, is just syntax. By itself, it does not *do* anything. It has a parser, but it is not executable; it has no compiler. Both the core standard and the behavior annex include a "semantic narrative" which very briefly describes, for example, the periodic dispatch of a thread or the message delay on a port. There are missing details that might be relevant to the modeling task at hand. For example, what happens if a message arrives when a thread is in its completed state? Does it immediately become active? Without some kind of executable semantics, it is difficult for a developer like myself to write an architecture description without that eerie, uneasy feeling about goats and cranberries.

This dissertation is part of a much larger project dedicated to the automated verification of avionics. It is motivated by two needs. First, there is a need for an industrial strength tool to effectively analyze avionics architectures. Second, there is a need for formal tools to evolve such that they are accessible and useful to a Rockwell Collins engineer: the average guy who does not have three decades of training in temporal logic and category theory.

The larger project is a collaboration between Rockwell Collins, Lui Sha, and José Meseguer. One of their goals is to formalize AADL in rewriting logic, using Maude. With this formalization, The AADL description can be subjected to different kinds of formal analyses, including model-checking.

Dedicated moviegoers might remember the 1968 movie, "The Odd Couple" with Walter Mathau and Jack Lemmon. Mathau's Oscar is a fat and sloppy cigar smoking womanizer; Lemmon plays Felix, an uptight neat-freak with a menagerie of allergies. Felix moves into Oscar's apartment after his wife leaves him. Funny antics ensue.

The combination of AADL and Maude is another odd couple. I see AADL

as Oscar, sloppy with his partial specifications and somewhat unnecessary bulk. Maude is my Felix, dotting every i and lower-case j in his specifications, unable to cope with bulky models due to his "allergy" to infinite states. Even in group meetings Professors Sha and Meseguer seem a little bit like Oscar and Felix: Lui Sha with his shoes off, feet on the chair, and José Meseguer in his sweater vest.

Stepping away from my goofy metaphor, defining a formal semantics for AADL in Maude is a big challenge. The key to AADL is partial specification, but model-checking cannot do without details. For example, AADL's notion of refinement means that data, ports, and the like do not necessarily need to declare their types. Meanwhile, in Maude, it is impossible to model-check a specification without these kinds of details.

This chapter describes the AADL2Maude toolchain that has resulted from the Sha-Meseguer collaboration. It also describes my contribution to the project: a case-study of the toolchain from a Simplex architecture perspective. With this case study comes the first known example of a formal automated analysis of a Simplex architecture.

9.1 The big links in the long chain

The AADL2Maude toolchain, as it has been envisioned by the Sha-Meseguer collaboration, permits developers to write an AADL specification that can be machine-checkable by tools like Maude's model-checker. As of writing this chapter, the toolchain's full implementation is incomplete, so I comment here on its design.

There are a number of plug-ins that, integrated together, make the toolchain possible. They are:

- **OSATE**. Pronounced "Oh-Sah-Teh" to sound Japanese[1], this Eclipse plug-in was developed by TopCaseD and the AADL team at SEI; it offers a text and graphical editor, parser, and a small collection of analysis tools including the Cheddar resource and scheduling analysis tool [75]. There is also an OSATE-BA plug-in which allows developers a behaviour annex parser. In OSATE, AADL models are maintained as textual AADL files or as XML-based AADL model files, providing a convenient interface to other plug-ins, such as MOMENT.

- **MOMENT2AADL**. MOMENT is a formal model-management framework available in Eclipse for which the underlying algebraic specification lanugage is Maude. The MOMENT framework offers tools for a number of languages, including AADL. The MOMENT2AADL plug-in takes as

[1] Do not confuse the fake Japanese word "Oh-Sah-Teh" with the real Japanese word "Oh-Sah-Eh-Te" which is the imperative form of the verb "Oh-Sah-Eh-Ru" which means "to pin down" or "to grasp" as in "Tom got a grip on his emotions."

input an AADL model represented in XML and outputs a Maude term; simply put, it translates one kind of tree to another. Formal analysis can then be conducted on the Maude term.

- **MOMENT Maude daemon**. This daemon encapsulates a Maude process into a set of Java classes; it offers an API which can control the Maude process in batch or interactive mode. As a result, developers can conveniently invoke Maude from within the Eclipse IDE.

The result of integrating these plug-ins is a five-stage toolchain that begins with an AADL Eclipse plug-in and ends with the Maude model-checker. An envisioned use-case for this toolchain is:

1. Start up the Eclipse Integrated Development Environment.
2. Use the OSATE Eclipse plug-in to write the textual specification of an architecture in AADL.
3. Use the MOMENT2AADL plug-in to convert the AADL .aaxl file into a Maude term; in other words, convert the XML tree into a tree that is readable by Maude.
4. Translate the Maude term into the format expected by the AADL-Maude Interpreter.
5. Use the MOMENT Maude Daemon to invoke the Maude model-checker and model-check the architecture description.

9.2 What is ready now: Peter's interpreter and my examples

What is currently available for the toolchain, in addition to the plethora of plug-ins described above, is a small AADL interpreter implemented in Maude by Peter Olveczky. This means that developers can write in an AADL-like language to describe their architecture using a small subset of the AADL syntax: systems, processes, threads, ports, and a small portion of the behavior annex. I refer to it as the AADL2Maude interpreter, or the A2M interpreter.

Consider a very simple example; a single component which transmits a single integer output equal to 1 exactly once. To describe this architecture in AADL means that I need to describe a single system component; within that system there is a process and within that process there is a thread. I can use the behavior annex to describe the thread's behavior as a finite state automata with two states. Below shows a portion of this single thread example description in Appendix F; I include only the thread specification here:

```
  thread thrSimpleThread
     features
        prtThreadOut: out event data port Behavior::integer;
  end thrSimpleThread;

  thread implementation thrSimpleThread.impl
     subcomponents
        none ;
     connections
        none ;
     properties
        Dispatch_Protocol => periodic;
     annex behavior_specification {**
states
           s0 : initial state;
           s1 : state;
        transitions
         s0 -[]-> s1 { prtThreadOut!(1); };
  **};
  end thrSimpleThread.impl;
```

To describe this thread, I need to talk both about its type `thrSimpleThread` and its implementation `thrSimpleThread.impl`. In its type, I dictate its single output port, `prtThreadOut`. In its implementation, I indicate that its dispatch protocol is periodic and I detail its behavior using the annex.

If the toolchain were complete, I could automatically translate the above AADL specification into a format that can be analyzed by Maude. Instead, I must write for the A2M interpreter directly. Describing this same thread in the A2M interpreter demands a little bit different syntax. The following is a portion taken from the exectuable code listed in Appendix G:

```
eq thread(thrSimpleThread) =
  ports (prtThreadOut out event data thread port)
  dispatch periodic-dispatch(1) .

eq TI thread thrSimpleThread . impl =
  < TI : Thread | features :
          ports(thread(thrSimpleThread)),
     subcomponents : none,
     connections : none,
     properties : dispatch(thread(thrSimpleThread)),
     status : completed,
     behavior :
        (states
```

```
        initial: s0 complete: s0 s1
      transitions
         s0 -[]-> s1 {prtThreadOut ! (1)}) > .
```

The original developer of the A2M interpreter, Peter Olveczky, was dedicated to making the syntax of the two as identical as possible. I am impressed; he got about 90%. Yet, notice a major difference between the AADL syntax and the A2M syntax.

Features are part of the implementation, not part of the type.

This difference speaks. It yells out to the world that AADL and Maude are indeed an odd couple. For Maude, a component implentation is not optional; there is no partial specification here. There is no point in separating the component type and the component implementation because both are needed to conduct model-checking.

There are also other subtler differences in the syntax. Subcomponents, for example, are not separated by any delimiter, but connections are separated by a semi-colon. Consider this portion of code from the simple Simplex example discussed later in this chapter. The three components are separated by returns; a comma separates the subcomponents from the connections, but individual semi-colons separated the connections.

```
eq SI system sysDotSimplex . impl =
  < SI : System | features : system(sysDotSimplex),
     subcomponents :
        ( idPSC process proSimpleController . impl )
        ( idPCC process proComplexController . impl )
        ( idPDM process proDecisionMachine . impl ) ,
     connections :
        ( idPSC . optSimpleCmd --> idPDM . iptSimpleCmd ) ;
        ( idPCC . optComplexCmd --> idPDM . iptComplexCmd )
  > .
```

These subtler differences speak to Maude operators and syntax design. While I appreciate the original designer's attempt to keep the two in step, the subtle differences were such that I felt I had to relearn the lanugage; 90% is not enough. Given that I had to learn another "AADL-like" language, I think time would perhaps be better spent on providing more functionality, even if the syntax is not identical. Moreover, given the effort to provide an automatic translation between the two, it seems that the need for identical syntax is less important.

With the A2M version of the system specification complete, I can conduct model-checking. Model-checking requires two parts: a system specification and a property specification. To create the property specification for the A2M interpreter, I must write a separate module indicating a target state of interest, as shown in Appendix H.

The initial state is specified in the behavior annex via keywords like `initial` and assignments to `state variables`. The target state is specified in a separate, above-mentioned Maude module. With this pair, one can perform a timed search on the model, using Real-Time Maude tools:

```
(tsearch [1] init =>* {C:Configuration}
   such that outputValue(C:Configuration) with no time limit .)
```

Again, I am reminded what an odd couple these two are. To conduct model-checking, one needs a system specification and a property specification. The system specification is a statement of how a system ought to behave. The property specification is a statement of the system's desired behavior. Model-checking is about making sure the two match; yet, there is no notion in AADL of these separate statements of what something ought to do and what something is desired to do.

In the end, the A2M interpreter offers *executable* semantics for a working subset of AADL. One can write an AADL-like specification using simple guards, simple actions, and integer messages. Model-checking takes place outside the realm of AADL, with a property specification specified in a separate Maude module. While the setup may be a bit clunky now, the Sha-Meseguer collaboration has the potential for a very powerful toolchain which automatically checks the safety of critical architectures.

9.3 A Simplex model for the A2M interpreter

I now describe the Simplex architecture for the simple dot example introduced in Chapter 6. I use only the implemented AADL subset in the A2M interpreter: systems, processes, threads, ports, and some of the behavior annex. The model is summarized in Fig. 9.1 using the graphical notation prescribed by the Software Engineering Institute.

I have delineated the two parts of the architecture using the AADL `system`. On the top of Fig. 9.1 is the software system, `sysSoftware`, which contains the modeling machine, the two controllers, and the decision machine. At the bottom of Fig. 9.1 is the hardware system, `sysRealWorld`, which consists of the physical plant and devices: the sensor and actuator.

The neatly depicted control loop in Fig. 5.1 is echoed here in SEI's messier and bulkier graphical notation. Starting from the bottom-center of the figure is the plant process, denoted `proPlant`. Within the plant process is the plant thread which describes the behavior of the plant. Notice that every process in the diagram has an accompanying thread; there is admittedly little use for the processes in this architecture except that the AADL semantics demand that threads belong inside processes.

Continuing with the diagram, a sensor monitors the plant and sends the monitored variable out of the system via the `soptMonVar` port. This variable is

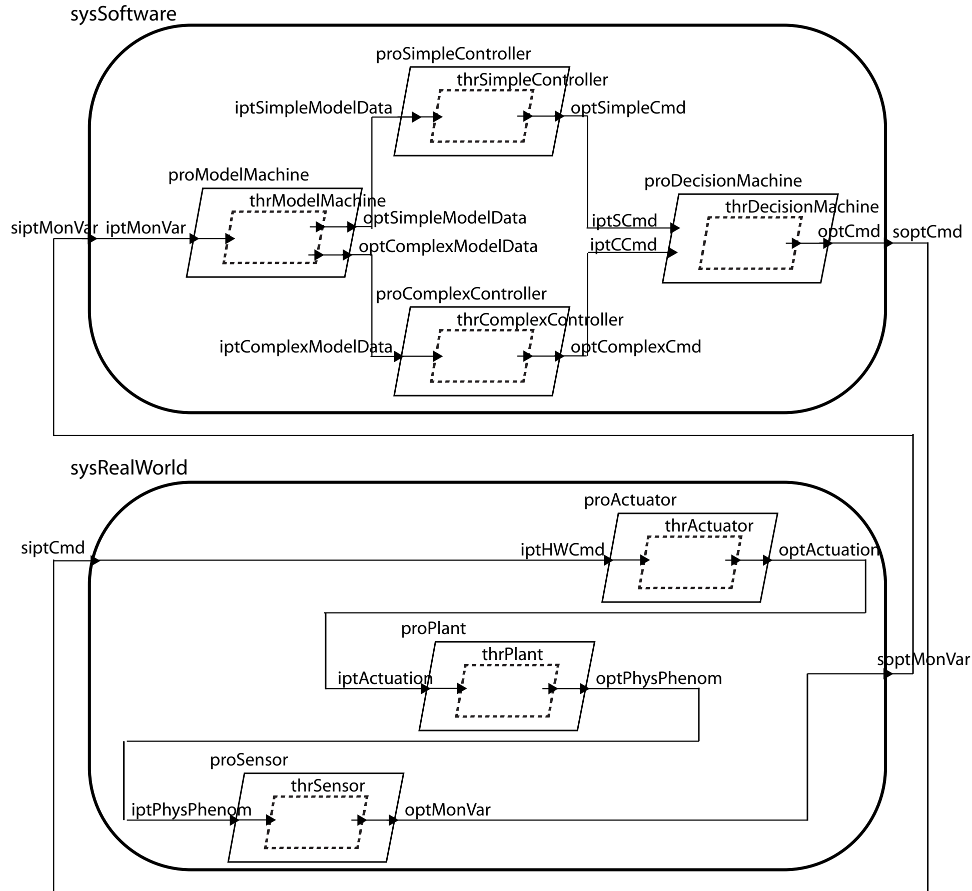


Figure 9.1: The architecture model for Simplex in the subset of AADL provided by the A2M Interpreter. A rounded rectangle represents a system. A solid rhombus represents a process. A dashed rhombus represents a thread. An arrowhead represents an event data port. Only two threads in the system are periodic: the `thrSimpleController` and the `proComplexController`. The remaining threads are aperiodic dispatch, reacting only to the messages they receive.

received by the modeling machine process, `proModelMachine` which maintains the software's internal view of the external world. The modeling machine delivers modeling events to both the simple controller and the complex controller. Each controller uses the event information to calculate an appropriate control command. Each controller forwards its control command to the decision machine; for each pair of control commands, the decision machine process forwards the selected control command to the actuator in the hardware system. Finally, the actuation command reaches the plant, and the control loop is complete.

So far, I have only talked generically about the architecture loop, omitting the details of the simple dot example. This could be misleading since there is nothing generic about this architecture description. The A2M subset does not have any support for parameterization. As a result, all of the behavior described is specific to the simple dot example.

The fully executable A2M specification for the simple dot example is available in Appendix I as well as the Real-Time System Integration group's subversion code repository. Rather than dump out the code here, allow me to highlight a couple of noteworthy points.

9.3.1 The reactive plant

Consider the thread behavior for `thrPlant`:

```
eq TI thread thrPlant . impl =
   < TI : Thread | features : ports(thread(thrPlant)),
      subcomponents : none,
      connections : none,
      properties : dispatch(thread(thrPlant)),
      status : completed ,
      behavior :
         (states
            initial: sReceive complete: sReceive
          state variables
             (inCmd |-> 0)
             (realDotPos |-> 0)
             (pRecpt |-> 0)
          transitions
          --- Receive an incoming command;
          --- update the position of the dot, and
          --- ‘‘send’’ the dot position to the sensor.
             (sReceive -[ ipthrCmd ? (inCmd) ]-> sReceive
                  { (pRecpt := pRecpt + 1) ;
                    (realDotPos := realDotPos + inCmd) ;
                    (opthrPhysPhenom ! (realDotPos)) } ) ) > .
```

The plant is described by a single-state automata which is executed aperiodically; it is activated whenever it receives an `inCmd` from the actuator on the port `ipthrCmd`. Upon activation, it executes three simple actions; it receives the new command, updates the position of the dot, and sends the new position to the sensor.

9.3.2 The two-phase, periodic controllers

All but two of the threads in the architecture have aperiodic dispatch; like the plant, these threads are simply reacting to messages received on their ports. However, the architecture is not purely reactive. The simple controller and complex controller are both periodic. Consider the following behavior for the complex controller's thread:

```
eq TI thread thrComplexController . impl =
  < TI : Thread | features :
       ports(thread(thrComplexController)),
     subcomponents : none,
     connections : none,
     properties : dispatch(thread(thrComplexController)),
     status : completed,
     behavior :
        (states
           initial: s0 complete: s0 s1
         state variables
           (cXmit |-> 0)
           (complexCommand |-> 2)
           (cModelData |-> 0)
         transitions
            (s0 -[ ]-> s1
     {
             (opthrComplexCmd ! (complexCommand)) ;
             (opthrComplexCmdForSimple ! (complexCommand))
             }) ;
            (s1 -[ ipthrComplexModelData ? (cModelData) ]-> s0
             {
              (cXmit := cXmit + 1)
             } ) ) > .
```

The complex controller operates in two phases. Upon activation at state `s0`, the controller transmits its latest command to the decision machine and the simple controller via its two output ports. The `complexCommand` is 2, which means the complex controller always directs the dot to go two meters-per-second. After transmission, the controller goes to sleep. Upon its next activation, it increments the transmission count, `cXmit`, and goes to sleep again.

How do I know the automata goes to sleep at both state 0 and 1? This two phase behavior is specified by the set of “complete” states. Upon reaching these states, the automata goes to sleep until its periodic dispatch makes it active again. It is this code snippet that makes it so:

```
complete: s0 s1
```

That’s how, but why? Why is the automata two phases, when all of that functionality could easily be packed into a single transition? It is due to the port semantics defined in the A2M interpreter.

There are three things to know about the port semantics in the A2M interpreter.

- Event data input ports are buffered.

- Event data output ports are *not* buffered.
- If a thread has a value in its input port, even if it has reached complete state, it will remain active.

To understand the impact of these semantics, consider the connection between the complex controller and the decision machine. Upon every other activation, the complex controller transmits a complex command. The decision machine is a five-state automata with aperiodic dispatch and a single complete state. This means that it begins in its initial state; it executes until it returns to its initial state, and then it goes to sleep. It does not become active again until it has a message on its input ports; it is reactive.

These port semantics have two significant impacts on specifying systems for the A2M interpreter. First, consider the lack of buffering on the output ports. This means that if the controller output were not connected to anything, the first message transmitted by the controller would sit at the port, it would not be consumed by any other process, and the system would be unable to progress. This is due to the following rewrite rule in the file `time.maude`:

```
--- For ports, if any transfer not yet dealt with: mte is 0:
eq mte
  (< P : Port | buffer : ML :: transfer(ML') :: ML''  >) = 0 .
```

This is just one equation of the collection that defines the "mte," or the *maximum time elapsed*; it is the amount of time that can elapse between two events in the specified system. An mte of zero means that time does not progress. So, if an output port generates an output that is not consumed, the system does not execute any further.

Second, if the controllers did not execute in their two-phase fashion, there would always be a pair of commands available to the decision machine, and it would stay permanently active; once again, time would never progress. Executing the two controllers in two phases means that the decision machine has time to consume the control commands, execute, and go to sleep before the next pair of commands become available.

9.3.3 Making a choice: The five-state decision machine

The decision machine must choose between the two control commands. The complex command always instructs the dot to go at two meters per second. The simple command always instructs the dot to stop. The decision machine chooses the control command based on the state of the dot. If it is in a high-confidence safety state−−if it is within stopping distance of the wall−−it chooses the complex command. Otherwise, it chooses the simple one.

The decision machine behavior is very simplified due to the limited functionality of the A2M interpreter. Actions consist of simple integer expressions:

addition, subtraction, and multiplication. Thus, to assess if the complex command can be chosen is a simple matter of addition. The decision machine reads in the complex command, increments the number of times a complex command has been received (very useful for debugging) and calculates the dot's distance from its starting point.

```
(sReceive -[ ipthrComplexCmd ? (inCCmd) ]-> sReceiveSimple
   { (complexRcpt := complexRcpt + 1) ;
     (distanceFromStart := dotPos + inCCmd + safetyEnv)
   } ) ;
```

The calculation includes a safety envelope, or `safetyEnv` which accounts for stopping distance. In subsequent states, the decision machine determines if the `distanceFromStart` is less than the wall position. If so, then choose the complex command.

9.4 Executing and model-checking the Simplex specification

With the A2M interpreter comes the incredibly valuable ability to *execute* a specification. Given the A2M interpreter is implemented in Real-Time Maude, I can perform a *timed fair rewrite* or a `tfrew` on the specification.

```
(tfrew init in time < 100 .)
```

I execute for 100 time units; that is long enough such that if my specification is incorrect, there is substantial time for the dot to accidentally hit the wall. At the end of the `tfrew`, I get an output with the following highlights:

- The position of the dot is 44 meters. The last command received was the simple command. The plant has received 50 commands. That is,

  ```
  (inCmd |-> 0)(pRecpt |-> 50)realDotPos |-> 44 .
  ```

- The complex controller and simple controller have each transmitted an output 50 times. That is,

  ```
  (simpleCommand |-> 0)transmissions |-> 50
  ```

  and

  ```
  (cXmit |-> 50)complexCommand |-> 2
  ```

These highlights are fantastic. By executing the specification, I can surmise that the decision machine begins choosing the simple command long before the dot is in danger of hitting the wall.

Not only can I execute the specification, I can also use the Maude model-checker to evaluate any safety properties. Model-checking requires two parts: a system specification and a property specification. To create the property specification for the A2M interpreter, I must write a separate module indicating a target state of interest, as shown in Appendix J. For example, I can define a state called `crashing` which yields a dot position equal to the wall position. This is defined by:

```
>) = VAL:Valuation ==
    (inCmd |-> 0)(pRecpt |-> 50)realDotPos |-> 50 .
```

Then I use a timed search to determine if the model I have defined ever reaches this crashing state.

```
(tsearch [1] init =>* {C:Configuration}
such that crashing(C:Configuration) with no time limit .)
```

A bit of warning, this particular timed search takes multiple hours on my iBook G4.

9.5 Analyzing dependency inversions in A2M

While my focus on Simplex has been safety, another interesting characteristic that has long been lauded by Lui Sha is this architecture's ability to eliminate dependency inversions. A dependency inversion occurs when the functionality of a highly-critical component depends on a lesser critical one. By separating the control task into two, the safety of the plant depends on a verifiably safe, simple controller instead of a potentially unreliable monolithic controller.

Given that A2M can take advantage of the tools made available by Maude, the fact that Simplex can eliminate dependency inversion can be systematically demonstrated. No matter how slow, buggy, or hostile I can imagine the complex controller to be, executing and model-checking the model demonstrates how it has been decoupled from the plant.

As with any modeling approach, this does have its limitations. It is limited by two things: developer paranoia and expressibility. First, a system can only be robust against the faults and failures that a developer anticipates. For example, in the early inverted pendulum prototype, Simplex could not have prevented pendulum instability that resulted from a faulty or corrupt sensor. Second, even if developer paranoia is high, it is useless if the given hostile complex controller cannot be modeled in the given language.

While the A2M toolchain is effective at demonstrating the elimination of dependency inversions in *models* of software and hardware, Chapter 11 demonstrates how one must turn a skeptical eye at the physical world to insure that even the plant and the sensors work reliably so not to compromise the critical path of system execution.

9.6 Evaluation of the A2M interpreter for Simplex

I return to the six requirements for describing Simplex, summarized in Fig. 9.2. For each requirement, this section discusses what A2M provides such that it is good enough; it also describes what is missing and why it is not absolutely great.

A2M Evaluation

Testable Precision	Support precise and testable specifications.	Score
T.1. Component behavior	A subset of the behavior annex is executable: simple guards, simple actions, integer messages.	B
Boundary Delineation	Support precise specification of system interfaces; support description of the system's environment.	Score
B.1. Hardware-software delineation	Hardware and software components may be specified in separate systems.	D
B.2. Component interface	Specifiy the component’s external interface in terms of ports.	B
B.3. Connections	Specifiy the component’s connections to other components via their ports.	B
B.4. Parameterization	No support.	F
Safety Definition	Define what makes an architecture safe for the complex controller and the simple controller.	Score
S.1. Safety properties	Specifiy the desireable safety properties in a separate model-checking module.	B

Figure 9.2: The six requirements for describing Simplex, and how A2M suffices.

9.6.1 Testable precision

It is not enough to describe just the components and connections of a software architecture. Boxes and lines are not telling. A software architecture description also needs behavior information for testable precision.

Component behavior

Why it's good enough. Enjoying the executable semantics offered by the A2M interpreter is an invaluable experience and really shows off the potential of this tool.

Why it's not great. At this time, only a small subset of AADL is currently available. The only thing that stands between a decent tool and a great tool is adding more AADL semantics to it. Future case studies, such as the ongoing project to model Rockwell's synchrnous bus, or PALS, will definitely help to create an even more useful interpreter.

9.6.2 Boundary delineation

Ideally, a specification language for Simplex ought to support the delineation of system boundaries, the precise specification of system interfaces, and the description of the system's environment. Users may indicate where the specification is internally incomplete, although the tool must allow users to isolate fuzzy or incomplete requirements and proceed with work on requirements that are well understood.

Hardware-software delineation

Why it's good enough. Because the AADL `system` construct is supported in the A2M interpreter, the simple dot example separates the physical world from the software one by putting each in a separate system.

Why it's not great. The A2M supports only a handful of AADL constructs; yet even if it supported all of AADL it would still have the same hardware-software delineation challenges as plain AADL. It is important for Simplex developers to make a distinction between the plant and the environment. But AADL's categories are limited; is a system's environment as a `device` or a `process`? Moreover, with the exception of the clunky behavior annex, there is no obvious way to describe how to maintain an internal model of the plant and environment; there's no good way to describe the solution to the model builder problem introduced in Section 4.2.4.

In addition, the same criticism I made of the divide between application software and the execution platform applies here. A flight control system has a control application running on top of some middleware, on top of some OS, on top of some hardware. It is true that a single developer or development team might not work at all of these layers; they may not wish to specify all the details of these levels. Even so, there is no natural vocabulary in AADL for describing layered components.

Instead, developers have to make careful use of the `extends` mechanism, where an application component extends a middleware component. This means

that a developer describing an application may very well have to look into the implementation of a middleware component in order to know what to extend. This is not good practice. Work done by Vergnaud and others [78] describes a loose mapping from middleware design to an AADL description: AADL packages, for example, map to reactive components in middleware. Their discussion is intriguing but the paper offers no enlightening examples.

Component interface

According to my chosen architecture definition, it is necessary to describe the abstracted functionality of one component insofar as it can be accessed by other components.

Why it's good enough. Because features are not separate from the component implementation in A2M, the component interface is specified simply as a list of ports associated with a given component. Consider this list of ports for the modeling machine thread:

```
eq thread(thrModelMachine) =
       ports (ipthrMonVar in event data thread port)
             (opthrSimpleModelData out event data thread port)
             (opthrComplexModelData out event data thread port)
       dispatch aperiodic-dispatch .
```

Why it's not great. In his book, *Domain Driven Design* Eric Evans talks about the necessity of "Intention-Revealing Interfaces" [22]. He writes, "If a developer must consider the implementation of a component in order to use it, the value of encapsulation is lost." Looking at my own list of ports for the modeling machine, it is not very easy to determine the intention of this thread; this list says nothing about the functionality of the thread that may be accessed by others. In comes a monitored variable, and out go two model data. Are the model data calculated from the monitored variable? Is the model data simply a forwarded monitored variable? Something else? Without looking at the thread behavior, it is impossible to tell.

Component connections

Why it's good enough. Similar to AADL, components are connected via three kinds of ports: event, data, and the event data port.

Why it's not great. The port semantics defined in the A2M present a real challenge. In their current form, the port semantics do not offer any way to model distributed systems. This is due to two facts:

- Output ports are not buffered.

- If there is an output port that is generating messages but is not connected to any input, the entire system comes to a screeching halt, the mte is zero, and time cannot progress.

I admit for the simple dot example that it is simply erroneous to have an output port that generates a message but has no complementing input port to consume it. But what about a more realistic system? Consider two devices communicating with each other over a bus. Suppose that one wants to analyze the behavior of the system given that the bus is faulty. It is impossible to do this with the current port semantics; yet it is quite possible for an actual system to progress just fine for a short time despite a faulty bus. As a result, there is no practical way to truly model a distributed system in the current A2M interpreter.

Parameterization

Why it's good enough. It's not. There is currently no support available for parameterization in the A2M interpreter.

9.6.3 Safety definition

The key goal of Simplex is safety; a Simplex architecture is defined by three sets of safety requirements:

- **Model safety requirements**. The model is required to provide an estimation of the real world within a specified tolerance.
- **Core safety requirements**. The simple machine is required to maintain plant safety.
- **High-confidence safety requirements**. The complex machine is required to maintain high-confidence plant safety.

Safety properties

Why it's good enough. The model checker offered by Maude is a linear temporal logic model checker. Linear temporal logic, or LTL, is a temporal logic in which one may describe properties of a single path of execution. Thus, any safety property that can be expressed in terms of LTL may be checked by the Maude model checker. For the simple dot example LTL is plainly adequate. I want to check that globally along the single path of execution the dot does not hit the wall.
Why it's not great. While for the simple dot example, LTL is adequate, expressing safety properties for the general case of Simplex requires a more expressive temporal logic. Chapter 10 details why Simplex may need more than LTL and it describes an alternate logical framework not checkable in Maude.

A note on the usability of Maude

Not many may realize this: there are two separate implementations of the Maude lanugage. There is "Core Maude" which is the Maude language implemented in C++. For instance, my simple dot example described in Chapter 7 was created in Core Maude. Then there is "Full Maude"; this is the Maude language implemented in Core Maude, with a few additional features. The A2M interpreter was written by Peter Olveczky in Real-Time Maude; a further specialized implementation made up from Full Maude and a few global clocks [55].

The brutally honest truth is that starting out with Full Maude is a miserable experience; this is due to its very poor parser. Writing specifications is a slow process; even expert Maude developers must work with Full Maude one line at a time so not to introduce a syntax error for which the parser gives no coherent error message.

For example, one of the first steps to writing a specification is to declare the names of the ports. This is done like so:

```
op opthrCmd : -> PortId [ctor]
```

However, if one omits the space between ":" and "-" all full Maude can say is that there is a syntax error *somewhere* in the file. There are even more dire cases than this. Consider this transition:

```
(s1 -[ on (aux == true) ]-> s2
      {  (isSafe := (carPos + 40 < wallPos)) })  ;
```

Accidentally add a space between "]" and "-" and Maude fails silently; she reports nothing.

That said, it took only two two months for me to adjust to the challenges presented by the Full Maude parser. I learned to write my specifications just a couple of lines at a time; I became an expert at very incremental refactoring. To my own surprise, I am willing to put up with a painful parser for the benefit of executability. I would much rather use A2M than plain AADL.

Other caveats when using the A2M interpreter

Because of the way the A2M specifications are defined, it is very easy to make dangerous mistakes that go unreported; these mistakes can take some hours and days to debug, depending on one's experience with the tool. I include this section for the future A2M developers hoping I can save them valuable debugging time.

Consider the begining of the simple dot example in Appendix I:

```
--- Declare all the names and ids of things.
ops sysSimplex sysSoftware sysRealWorld :
   -> SystemName [ctor] .
```

```
ops idSS idSRW MAIN : -> SystemId [ctor] .
op impl : -> ImplName [ctor] .

--- Process Names and IDs
op proComplexController : -> ProcessName [ctor] .
...
```

An A2M specification begins with the model's signature. It is the model's syntax: a declaration of all the names that are to be used in the model's semantics. Because of this organization, if a name is used in a wrong component, no error is reported.

Let me give an example. Consider the following transition in the component `thrDecisionMachine`:

```
---  The complex command is not safe. If the simple one is
---  available, send it and update the internal model of
---  the car's location.
(s3 -[ ipthrSimpleCmd ? (inSCmd)]-> sReceive
   {
      (carPos := carPos - inSCmd) ; (opthrCmd ! (inSCmd))
   } ) ) > .
```

This transition uses two of the five states for the decision machine. However, suppose that I replace `sReceive` by `s0`. This is an error, but A2M does not report it; the AADL parser would. In A2M, `s0` has been declared in the signature of the model, but it is not one of the states intended for the decision machine. Maude does not complain because the state name has been declared at the top of the file. Instead, when executing the model, execution cannot progress after time 0 because there is no transition from `s0` to any other state in the decision machine thread.

Let me give another example. Consider this *incorrect* portion of the simple dot example, the specification for the modeling machine process and thread:

```
 eq PRI process proModelMachine . impl =
    < PRI : Process | features : process(proModelMachine),
       subcomponents :
        (tidMM thread thrModelMachine . impl) ,
       connections :
        (iptSimpleMonVar --> tidMM . ipthrSimpleMonVar ) ;
        (tidMM . opthrSimpleModelData --> optSimpleModelData) ;
        (tidMM . opthrComplexModelData --> optComplexModelData)
    > .

eq thread(thrModelMachine) =
   ports (ipthrSimpleMonVar in event data thread port)
```

```
        (opthrSimpleModelData out event data thread port)
  dispatch aperiodic-dispatch .
  ...
```

The process `proModelMachine` specifies three connections between the model machine process and thread. Meanwhile, the thread `thrModelMachine` declares only two ports. However, because the name `opthrComplexModelData` has been declared in the model signature, there is no error reported. Once again, when executing the model, execution cannot progress after time 0 because there is a discrepancy between these ports and their connections.

All told, whenever I found a model specification that could not progress past time 0, it was usually due to a discrepancy in the ports and their connections.

9.7 Conclusion

Despite its limited functionality, quirky port semantics, and Full Maude's poor parser, the A2M interpreter stands high above plain AADL for one simple reason: *executability*. The A2M offers a decent level of testable precision; I can write a specification, execute it, and iterate. I learn significantly more about a system specification when I get to execute it than when I just put it down on paper or stuff it into a parser. I look forward to great things from the A2M interpreter.

10 Simplex in higher order logic

The Simplex architecture is designed to combine two redundant components or order to achieve a particular goal with the greatest possible performance while making safety a priority. On the one hand, an unreliable component is designed for high performance; it is a complex, possibly unverifiable component in my system that yields safe output most of the time. On the other hand, a trustworthy component is designed for safety. The trustworthy component has fewer features, but has been verified that it yields safe output all of the time.

So far, the focus on this dissertation has been on the architecture specification: drawing boxes and lines, describing behavior, and conducting analysis. For the simple dot example, safety properties expressed in linear temporal logic were enough; but what about more complicated examples? Consider the general statement of control systems described in [4]:

> *Given a state machine whose transitions are partitioned into controllable and uncontrollable, a set of safe states, the control problem asks for the construction of a controller that chooses the controllable transitions so that the machine always stays within the safe set.*

For the general case, I want to express the current plant state in terms of the most recently received control command and the most recently monitored control variable. Yet at the same time, the next control command partially depends on the currently monitored control variable. This kind of reactivity cannot be expressed in linear temporal logic, because one cannot relate state variables across different states. To express reactivity within the Simplex architecture, I want to express that there must exist at least one simple controller output that will always keep the plant in a safe state. Expressing this alternation of “any possible input” and “one control command” is a challenge.

In this section, I describe the logical reference model I developed with committee member Elsa Gunter for reasoning about Simplex. In particular, I present a logical framework for a “recoverable” or high-confidence safety state, and describe how the decision machine uses this framework to uphold the safety requirements.

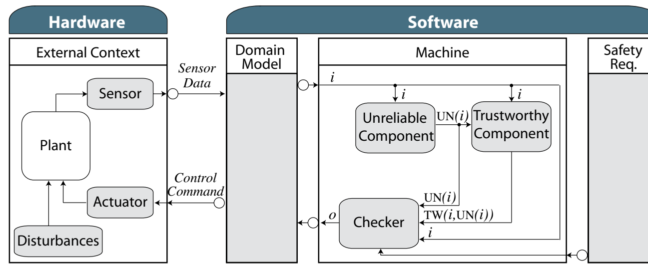


Figure 10.1: The Simplex reference model.

10.1 Decomposition of the logical reference model

Shown in Fig. 10.1, my logical Simplex reference model consists of the following decomposition:

External Context

Consists of all the physical devices external to the computing machine. This includes the *Plant* being controlled, as well as the *Environment* in which the *Plant* operates. The *Environment* includes *Disturbances* to the *Plant*, actuator dynamics, and sensor inaccuracies. The *External Context* contains no software, at least from the perspective of the development team. Perhaps there is a sensor that contains embedded code, but it is not under the discretion of development. A *Sensor* provides *Sensor Data* and an *Actuator* receives *Control Commands*.

Domain Model

Represents everything that is known about the *External Context* that can be used to estimate the behavior, or dynamics, of the *Plant*. It maintains an estimate of the current *Plant* state given *Sensor Data* and *Control Commands*. It can project the *Plant* state one time-step into the future. The *Domain Model* interprets the *Sensor Data* and sends it to the *Machine*. The *Machine* replies with an output which the *Domain Model* translates to a *Control Command* for the *Environment*.

Machine

Consists of all of the control logic used to generate control actions applied to the *Plant*. The *Machine* comprises multiple redundant subsystems designed to achieve the same minimal performance goals. A *Checker* chooses between the outputs of the redundant components depending on the current *Plant* state.

Safety Requirements

Safety requirements are a statement of what must always be true in the *Plant* to achieve safety. The *Safety Requirements* are embedded in the *Machine* so that it can choose an output from its redundant subsystems that will keep the *Plant* in a safe state, both now and in the future.

This decomposition I describe, notably the separation of the *Machine* functionality from the *Domain Model* is based on the principle of separation [10, 80]. To summarize, an optimal state estimate and optimal control action can be computed separately, and are still optimal when combined. Since this reference model depends on this separation result providing for such equivalent optimal control, it is not applicable to the CPS domain when the theorem is not valid [79].

I delve into greater detail for each entity in my Simplex reference model. Shown in Fig. 10.1, the *Machine* comprises an *Unreliable Component*, *Trustworthy Component*, and *Checker*, which I call UN, TW, and CHKR. Given interpreted *Sensor Data*, the CHKR selects output from the UN or the TW; preference is given to the UN to take advantage of its full features, but the output must always maintain the *Plant* in a safe state with respect to the *Safety Requirements* expressed in the CHKR.

Given the UN and the TW, the CHKR must be able to: i) Determine if the UN output will place the *Plant* in a state that satisfies the *Safety Requirements* at the current time. It must also determine if TW can continue to satisfy the requirements in the future; ii) Choose the TW output if the UN output places the *Plant* in a state that must eventually lead to a state that does not satisfy the *Safety Requirements*; iii) Choose the TW output in a timely manner such that the *Plant* is always in a safe state.

Suppose that my *Safety Requirement* is that a given object does not hit a particular obstacle. Just because the object is currently not hitting the obstacle at this moment does not mean that it is in a safe state. It may be moving at too high a velocity to stop before it hits the obstacle. The object can be "safe at the moment" and still not "always safe." Thus, I must differentiate between three kinds of *Plant* states.

- **Recoverable**. Satisfies the *Safety Requirements* to such a degree that the *Plant* can tolerate "aggressive" commands.
- **Safe**. Satisfies the *Safety Requirements*, yet cannot tolerate "aggressive" commands.
- **Unsafe**. Does not satisfy the *Safety Requirements*.

To say that the *Plant* is in a recoverable state makes a statement about two things: the *Plant* and the TW component. First, a recoverable state is one that satisfies the *Safety Requirements*. Second, to say that a *Plant* state

is recoverable requires that if UN output is used in the current time-step, the TW component output must be able to satisfy the *Safety Requirements* in the next time-step. To state it simply, "In a recoverable state, if you choose the unreliable command now, the trustworthy command must ensure safety later."

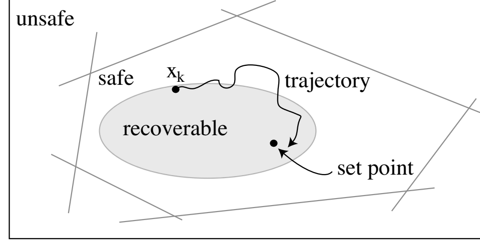


Figure 10.2: The Simplex state space.

I define these states for the control domain. Referring to Fig. 10.2 I denote the state of the plant at time k as $\mathbf{x}_k$. As time passes, the state of the plant evolves, forming a trajectory, T. The trajectory is dependent on the *Plant* dynamics, control policy and initial condition. Hence, we represent a point in the system trajectory at time j, under control policy U, starting with initial condition $\mathbf{x}_k$ as $T(\mathbf{x}_k, U, j)$. A state is considered *safe* if it lies within the safety set, which is denoted by $\mathcal{S}$. Unsafe states are denoted as $\bar{\mathcal{S}}$. I define a state as being "recoverable" if the future trajectory lies within the safety set:

$$T(\mathbf{x}_k, U, j) \in \mathcal{S} \quad \forall \quad j \geq k \tag{10.1}$$

Thus, the set of all states $\mathbf{x}_k$ that satisfy (10.1) represents the recoverable set $\mathcal{R}$:

$$\mathcal{R} := \{\mathbf{x}_k : T(\mathbf{x}_k, U, j) \in \mathcal{S} \quad \forall \quad j \geq k\}$$

Note that $\mathcal{R}$ is dependent on the particular control policy U employed. If I use the trustworthy controller U_{TW} to define $\mathcal{R}$, then I can stipulate the following two requirements for the unreliable controller U_{UN} to be accepted at time k:

$$T(\mathbf{x}_k, U_{\text{UN}}, k+1) \in \mathcal{R}$$

This implies that:

$$\mathbf{x}_{k+1} = T(\mathbf{x}_k, U_{\text{UN}}, k+1) \in \mathcal{S}$$

and

$$T(\mathbf{x}_{k+1}, U_{\text{TW}}, j) \in \mathcal{S} \quad \forall \quad j \geq k+1$$

The notion of recoverable can also be expressed as "stability in the sense of Lyapunov" [10].

10.2 Logical framework

The reference model requires its logical framework to express reactivity. Given any possible input from the *External Context*, subject to the *Domain Model*, there must exist at least one TW output that will always keep the *Plant* in a safe state. Expressing this alternation of "any possible input" and "one control command" is a challenge.

On the one hand, I want to express the current *Plant* state in terms of the most recently received input and the most recently transmitted output. Yet at the same time, I would require recursion such that the next input partially depends on the current output.

Streams. I assume that I are given the streams of input and output a priori. Practically, it seems unrealistic to know all inputs at once, but I argue that, for my purposes, is mathematically equivalent to the reactive nature I wish to capture.

- Each stream is a sequence of inputs or outputs for each point in time. The input stream, ins, indicates the input, i, for each time k. I denote the input at time k as $i_k = ins(k)$.
- I denote the kth prefix of an infinite stream, s, as $s^k = (s(0), \ldots, s(k))$. Consider a finite prefix of the input stream, $ins^k = (ins(0), \ldots, ins(k))$. The function $out(ins^k)$ produces the kth output in response to the first k inputs, meaning that I do not allow the output stream to look into the future. To denote the output at time k, I use $o_k = out(ins^k)$.
- The current state of the plant, constructed from all previous inputs and outputs, is defined by the current input/output pair, (i_k, o_k).

Safety. The ultimate goal of Simplex is for the *Machine* output to satisfy the *Safety Requirements*. To express this notion of safety, I must differentiate between two safety predicates: "safe at the moment" and "always safe."

- I denote "safe at the moment" as $\texttt{sf}(i_k, o_k)$. For $\texttt{sf}(i_k, o_k)$ to be satisfied means that the *Plant* satisfies the *Safety Requirements* in the current state (i_k, o_k).
- To define "always safe" or `Safe`, I use a predicate $\texttt{pn}(ins^k, out(ins^k))$ that yields the set of "possible next" inputs at time $k+1$ given the input and output prefixes up to time k. Essentially, this `pn` predicate is equivalent to my reference model's *Domain Model*, which dynamically predicts the future state of the plant based on past and current sensor input and command output.

- I define `Safe` with respect to: i) the `sf` predicate; ii) the current *Domain Model* of the *Plant*, `pn`; and iii) the output stream function. Thus, `Safe` is evaluated dynamically with respect to the current state and the current output.

Recoverable. In order to guarantee that the *Plant* is safe with respect to the TW, I require that the *Plant* is in a recoverable state when choosing any output.

- The output of the UN component is $\text{UN}(i)$. My only assumption of UN is that if it produces output, the output is of the correct type.

- TW uses both the original input as well as the unreliable output to calculate its own output, $\text{TW}([i_0, \ldots, i_k], [\text{UN}(i_0), \ldots, \text{UN}(i_k)])$.

- I define the recoverable predicate, $\mathtt{r}(i, o)$. This predicate, like `pn` and `Safe`, can be evaluated dynamically to determine if the *Plant* is in a recoverable state. A *Plant* is recoverable if: i) at the current time-step the current input/output pair satisfies the *Safety Requirements*, $\mathtt{sf}(i, o)$; and ii) for all the possible next states, a sequence of outputs exist which keeps the *Plant* `Safe`.

$$
\begin{aligned}
&\mathtt{r}\langle \mathtt{sf}, \mathtt{pn}, \text{TW}\rangle([i_0, \ldots, i_k], [o_0, \ldots, o_k]) = \\
&\mathtt{sf}(i_k, o_k) \wedge \text{let } out([x_0, \ldots, x_m]) = \\
&\text{TW}([i_o, \ldots, i_k, x_0, \ldots, x_m], \\
&\quad [o_0, \ldots, o_k, out([x_0]), \ldots, out([x_0, \ldots, x_{m-1}])]) \\
&\text{in } \mathtt{Safe}\langle \mathtt{sf}, \mathtt{pn}, \mathtt{out}\rangle
\end{aligned}
$$

- **Criteria:** If the CHKR does not choose the TW output, then the UN output must be recoverable.

$$
\begin{aligned}
&o_k \neq \text{TW}(ins^k, \text{UN}(ins^k)) \Rightarrow \\
&\mathtt{r}\langle \mathtt{sf}, \text{PN}, \text{TW}\rangle([i_0, \ldots, i_k], [o_0, \ldots, o_k])
\end{aligned}
$$

Discussion. To evaluate the recoverable predicate requires knowledge of the *Plant's* state. Given that the system is interacting with the physical world, the system must have three sets of information to make this evaluation. Foremost, it needs *Sensor Data* interpreted by the *Domain Model*. Next, it requires knowledge of the entire *External Context*: the mass of the device, the speed of the vehicle, the accuracy of the sensor. Finally, it requires an estimate of the *Plant's* current state.

Though I have shown a fairly strenuous framework, I recognize that in some instances calculating the set of recoverable states is a more intuitive decision procedure. For example, in the original Simplex prototype, the Lyapunov function [50, 47] was used to calculate the set of recoverable states for the inverted

pendulum. In this paper, I do not claim to estimate a recoverable state, rather we define what the recoverable state must imply. It is up to domain experts to determine how to calculate the recoverable state.

10.3 Conculsion

The Simplex architecture is difficult to formally describe in the general case because of its reactivity. Developers must express the current plant state in terms of the most recently received control command and the most recently monitored control variable. Yet at the same time, the next control command partially depends on the currently monitored control variable.

Reactivity is difficult to express in industrial strength logics such as linear temporal logic or computational tree logic. This challenge has been recognized by the formal methods community; newer logics and approaches have emerged to address this problem [4, 59]. The logical Simplex reference model presented here is a higher-order logical framework to help developers describe Simplex architectures in the general case.

11 Simplex improves safety in the convergence laboratory

This chapter describes one more example of the Simplex architecture. It is not a historical prototype like the inverted pendulum or the diving controller; it is my own prototype neatly inserted into the convergence laboratory's testbed, an existing platform for distributed networked control.

The convergence lab architecture is a control loop executing across distributed components in a wired and wireless network. But, as I identified in multiple experiments, the vision sensor performs unreliably. In the worst case conditions, the vision sensor only identifies cars correctly about half of the time. The key safety property in the lab is collision avoidance, yet it is difficult to achieve in the presence of unreliable sensor data.

Because of this unreliable sensor, my Simplex prototype is the first that pays significant attention to the modeling machine. Never before had any Simplex prototypes seen this kind of unreliable data. The bulk of my prototype is the set of strategies that I had to develop to main the software's internal view of the physical world so that the decision machine could better choose between the simple and the complex controller. As a direct result of my work, the vision subsystem identifies cars correctly at least 99% of the time. Moreover, the vision subsystem implicitly indicates when it is uncertain about car locations. Silence means, "I don't know."

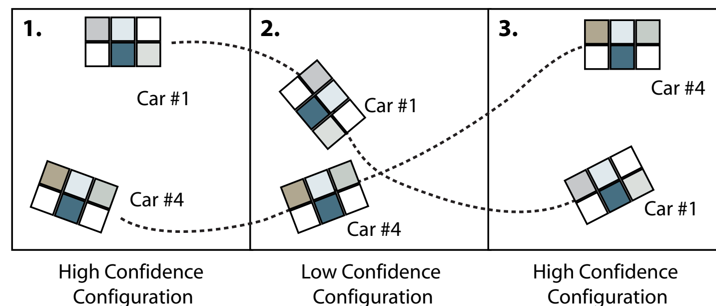


Figure 11.1: The trajectories of cars #1 and #4 cross; the car locations and identities go from a configuration of high confidence to a configuration of low confidence and back again.

There is a frustrating incongruity to creating the Simplex modeling machine for the convergence laboratory: I can *easily* explain why tracking cars and managing their identities with two cameras is incredibly *difficult.* Consider Fig. 11.1: Cars #1 and #4 are traveling about the platform. I choose these

two cars for the example because Car #1 is sometimes mistaken as Car #4 and vice versa. Note in the figure that their travels are broken into three frames. In frame 1, Cars #1 and #4 are a safe distance apart; it is fairly easy to maintain a correct identity and location for each car. Even if reports of their identity are incorrect, it is fairly easy to deduce that a mistake has been made by using a validation region. To anthropomorphize the vision subsystem, "In the last observation, Car #1 was way over there. Now you are telling me it's way over here!? Give me a break!"

Here comes trouble. In frame 2, Cars #1 and #4 are now in close proximity. What happens in the worst case; Car #1 is reported as #4 and #1 is reported as #4. It is very difficult to tell if a mistake has been made in the reporting. Even with a set of past observations, both cars are reasonably close to the last observation. It is easy to believe the mistaken reports. But mistaken reports lead to weird behavior; with the cars so close together, weird behavior can quickly lead to collisions.

The uninitiated quickly list a set of ad-hoc heuristics to solve these problems. "Stop the car when you don't know!" But in the second frame's worst case, how does the system *know* that it does not know? Even worse, stopping the cars leaves them both in a low confidence configuration. "Don't let the cars get close together!" This can be done to some extent, but how impressive is a collision avoidance demonstration if the cars are always 5 feet apart? "Don't drive these particular two cars at the same time!" Maybe. But it is currently impossible to come up with a reasonably large set of cars which are not mistaken as any other.

People in the radar community use a variety of Bayesian correlation approaches to solve these kind of tracking and identity management problems. Ideally, this means calculating all possible target positions for all target identities and finding the likeliest match, a task with exponential complexity in the number of targets. More efficient algorithms for tracking multiple targets date back as early as 1979 and were developed with radar applications in mind. MHT [58] maintains a set of associations which are pruned over time based on a rank function. At each iteration, JPDA [8] computes an association matrix which is updated by a combination of the latest observations and their marginal probabilities.

Today, radar is not the only domain for tracking targets. The introduction of video cameras into domains like surveillance for airport security and traffic management have made it necessary to track not just blips but also color images. BraMBLe, or the Bayesian Multiple-Blob Tracker [35], combines two technologies to perform multiple object tracking with a static camera, even in cases where the number of objects being tracked is unknown. First, BraMBLe uses a blob tracker, or statistical appearance models to track foreground objects despite their deformation across frames. As a result, both background and foreground models are necessary to track the blobs. Second, a particle filter is used

to calculate a posterior distribution over the number and configuration of objects. The algorithms presented in [17] and [46] are variations of the BraMBLe approach, and use particle filters to achieve their goals.

Wireless sensor networks have also entered the tracking arena. Sensor networks must be able to track multiple objects across multiple sensors. An interesting algorithm for managing target identities in a distributed fashion over a wireless sensor network offers $O(N^2)$ complexity in the number of targets [68]. It includes an approach for how sensors can update each other with their locally known information.

All of these algorithms are suitable for tracking targets in a variety of domains, but they all seem like too much firepower for the testbed where the problem is to use the data from two cameras to associate at most nine toy car locations with nine known identities. One must balance goofy ad-hoc heuristics with the complexity of updating and transposing doubly-stochastic matrices.

Thankfully, Simplex is "using simplicity to control complexity"; it is an inspiring tag line. My approach to building the modeling machine was founded on a simple mantra,

Be as accurate as possible; never keep secrets

By focusing my skeptical eye on the physical world of the testbed, I have uncovered that it is possible to track cars and manage their identities with reasonable accuracy and performance. I have done this by eliminating dependency inversions between the cars themselves and the vision sensor. As demonstrated in Chapter 9, the Simplex architecture decouples the safety of the plant from an unreliable, monolithic controller. It does so by removing dependency inversions between a highly critical controller and other less-critical components. But that alone is not enough. The testbed is an entire system; it is hardware and software, plant and controller. The safety of the plant depends on all of the components in the path between it and the decision machine: even the plant itself.

11.1 Applications in the convergence laboratory

Simplex in the convergence laboratory is integrated into the set of control applications which sit at the application level in the testbed. An applications is a collection of components which communicate with each other over Etherware to control a fleet of remote-controlled cars. Different applications direct the cars towards different goals:

- **Trajectory**. A simple trajectory application where cars travel in a large circle;

- **Traffic**. Cars adhere to the rules of the road in a network of roads. Cars drive on the right side of the road and stop at intersections;

- **Police**. A policed traffic scenario. One set of "civilian cars" follows traffic as normal which another set of "police cars" chase a rogue car. Police cars are given priority over civilian cars in traffic, as the civilians are stopped to allow the police through;

- **Avoidance**. An extension, or additional set of components, which the above applications may include to avoid collisions between cars.

At a high level, the key software components of all these applications are the Vision Sensor, Trajectory Planner, Controller and Actuator. Applications vary by choosing a particular Trajectory Planner. Far more components are involved in the actual implementation, but these four are the core.

Networked together, the four core components execute a control loop, as shown in Fig 11.2. The Vision Sensor observes the color panels in the plant and converts them to car information: a list of car identities with location and orientation information. Based on this information, the Trajectory Planner revises the path that each car must take to achieve its goals; this path is expressed as a set of waypoints to the Controller. The Controller uses the car information and the next waypoint to calculate the next command for the Actuator which transmits the commands to the physical plant and completes the loop.

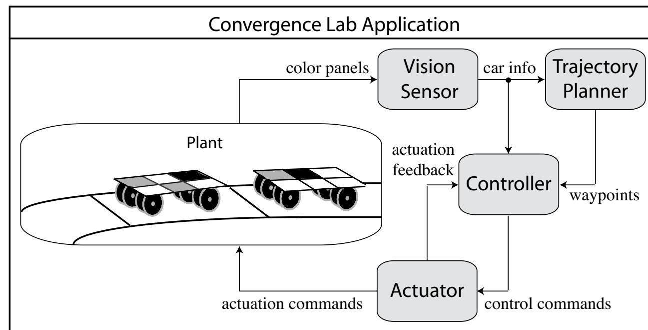


Figure 11.2: Four key components make up the control loop which directs a fleet of remote-controlled cars.

11.1.1 The vision sensor subsystem

Fig. 11.3 summarizes the collection of components used to implement the vision subsystem for the testbed. From the left, the subsystem begins with a camera−−left or right−−mounted to the ceiling of the testbed. The camera captures the systems monitored variables: the set of color panels mounted to the car rooftops, their colors, x-position, y-position, and size.

The raw camera feed is processed by a Matrox Imaging Library that extracts a set of color blobs. Which blobs are extracted depends on a configuration file

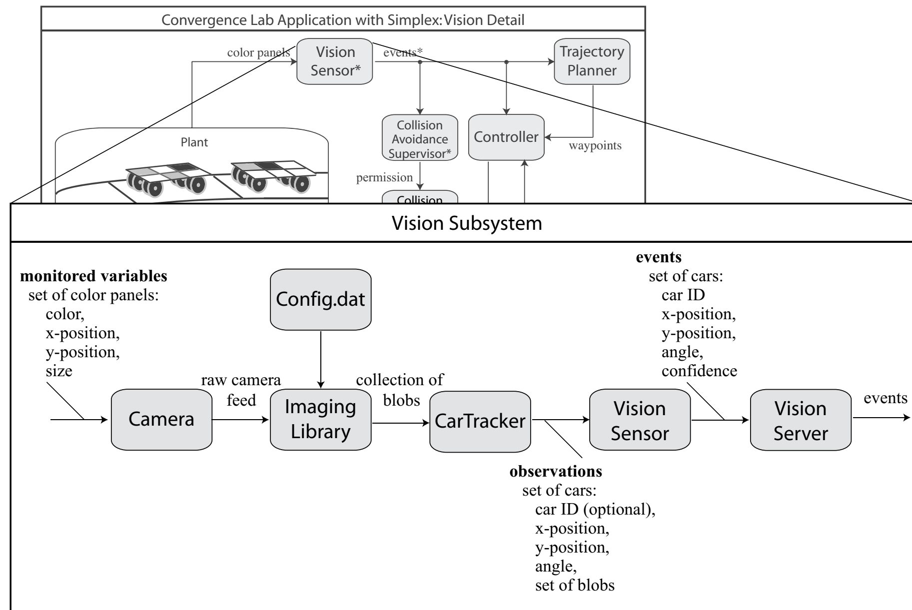


Figure 11.3: Logical architecture for the vision subsystem

which provides a list of blob types to locate. Simply put, I tell the library "look for blue blobs about this big" and the library says, "I saw 2 blue blobs, here and here." For example, here is the line of the configuration file which defines the color dark blue.

```
144 184 1 255 15 255 5 200 //DarkBlue
```

The library is configured by hue, saturation, luminance and size. The first pair of numbers indicates the low and high hue values, the second pair describe the saturation range, the third describe luminance. The fourth pair describes the size range. The comment `//DarkBlue` is simply for readability of the file. Eight colors are used in the testbed and are all described in this configuration file.

The collection of blobs is interpreted by the CarTracker component. Using a simple least squares algorithm, the CarTracker component reorganizes the collection of blobs into sets that could possibly make up a car. Each sets's orientation and center position are calculated. If possible, a reasonable guess at the car's identity is made, based on a list known car identities and how well the set's colors match the list. Keep in mind that colors may have been observed incorrectly, so this guess could very well be wrong. The calculations and guess are then packaged up as a list of observations, one for each set of blobs.

Together, the CarTracker component and Imaging Library are one of the weird frontiers of the testbed. The majority of the testbed executes in Java, as components on top of Etherware. However, because the imaging library is a C++ library, the CarTracker is a stand-alone component which invokes the

library and translates the collection of blobs into observations. The observations are snuck into Etherware via a file buffer, where they safely reach civilization.

The Vision Sensor component, executing in Etherware, takes the observations and uses both previous observations and a plant model to make more intelligent guesses at the car's identities and locations. Finally, the events are sent to the Vision Server which distributes them to the other interested components.

11.1.2 Flaws in the original vision subsystem

This section describes the algorithm and flaws of the original vision subsystem. It is not intended as a therapeutic laundry list of complaints; rather, key flaws have been selected to motivate the design decisions of the current vision subsystem.

At a high level, the vision algorithm works in three steps. At each step, a flaw in the original implementation contributed to the incorrect reporting of car identities. These are summarized here, and detailed in the following subsections.

1. **Extract color blobs from the camera feed.** Unattractive Mead paper made the blob colors difficult to identify. Dark green was reported as light green; purple is hardly ever reported.

2. **Group color blobs and identify cars.** Color blobs grouped into cars are matched against a set of known car identities. Due to mistaken color observations, a group of blobs may not match any of the known identities. These groups are not reported.

3. **Filter car list.** The Vision Server expects a list of unique cars and locations. Due to mistaken identity, if two cars are observed as Car #1, only one of them is reported; worse, the car that is reported may not actually be Car#1.

What these three steps point to is a vicious cycle of dependency inversions. The plant's safety depends on commands from the controller that keep the cars at a safe distance from each other. The controller depends on the vision subsystem to give it useful information to calculate these commands. But the plant itself is not amenable to observation. Bad paper makes the cars hard to identify. This interferes with the safety of the plant. The vision subsystem keeps some observations secret. This interferes with the safety of the plant. The safety of the plant depends on all of the components in the path between it and the decision machine. To eliminate all these dependencies, they must first be understood.

Step 1: Unattractive paper makes blobs hard to identify

The first flaw was the Mead construction paper. This paper was poorly dyed and prone to fading, bad visibility, and mistaken identity.

To uncover this, a set of color patch experiments were conducted. Each experiment was conducted with a car equipped with a single color patch on its rooftop and manually driving it randomly around the left end of the platform. Debug information was added to the CarTracker component which yielded all the blobs identified at every iteration. A small python script, clm.py, was used to extract the statistical performance of each blob[1].

Two metrics were used to evaluate patch performance: Absolute success and relative success. Absolute success refers to what percentage of the iterations the given color blob was observed correctly. For example, if the experiment runs for 10 iterations, one expects to observe 10 color blobs. If 10 color blobs of the correct color are observed, then a color has 100% absolute success. The graph in Fig. 11.4 summarizes the absolute success of each color patch. The top performing colors are the fluorescents; the two fluorescent oranges and green have enviable success. The Mead colors performed badly, with purple as the dunce.

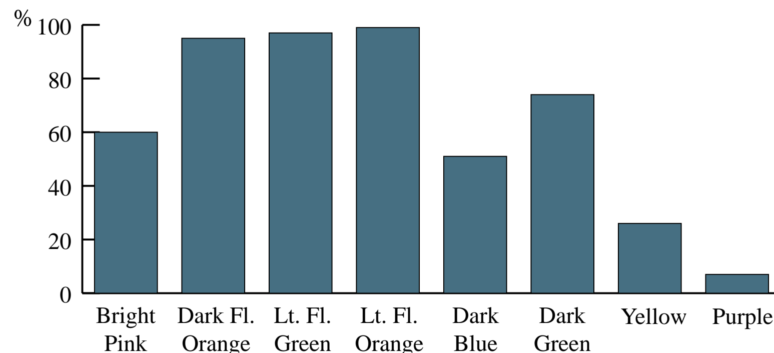


Figure 11.4: Absolute success of observing color patches using the CarTracker C++ module configured with the LeftCarConfigTracker_1.dat configuration seen in Appendix A.

"Relative success" refers to how many times, out of the total number of blobs observed, the given color blob was observed correctly. For example, if the experiment runs for 10 iterations, but only 7 blobs are observed, and the color is observed correctly 7 times, then a color has 100% relative success. All the colors but dark blue and dark green have about 100% relative success.

Relative success for these colors was poor because the dark blue and dark green colors were often mistaken as other colors, as shown in Fig. 11.5. If a dark blue patch was successfully captured by the camera and imaging library, 59%

[1]For those interested in repeating these experiments, information and source code are available on the convergence laboratory wiki under the "Color Patches" link. Visit the internal wiki at https://agora.cs.uiuc.edu/display/convergencelab/Home. Members only.

of the time, a purple patch was reported. Similarly, the dark green patch was thought to be a light fluorescent green patch 23% of the time.

These cases of mistaken identity sometimes resulted in "extra" blobs being reported. For example, during the first experimental trial for dark blue, 132% of the total number of expected blobs were observed; this means that a single patch was often reported as *both* a blue and purple blob. The reason for this was a configuration file whose definition of "purple" and "blue" overlapped. See LeftCarConfigTracker_1.dat in Appendix A and notice how the ranges of values for hue, saturation, and luminance for purple and blue overlap.

Thus, the first motivation for redesigning the vision system was eliminate the dependency inversion between the plant and the camera: to select new paper and reconfigure the imaging library so that colors were observed with greater success, and no color was mistaken for another.

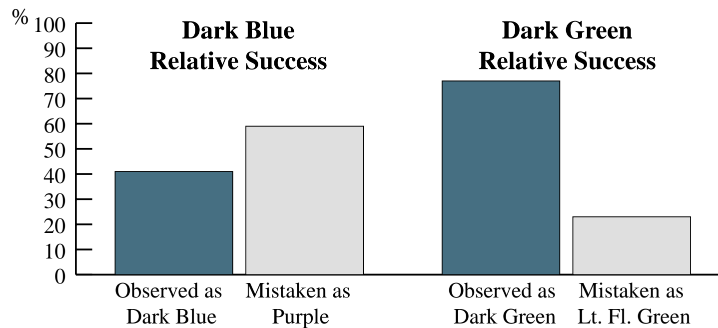


Figure 11.5: Relative success of observing dark blue and dark green patches using the LeftCarConfigTracker_1.dat configuration seen in Appendix A.

Step 2: Unnecessary statelessness and secrecy results in car under-reporting and mistaken identity

The original CarTracker component was secretive and stateless. If it observed a group of colors that it did not expect, it did not report that group. Moreover, the original Vision Sensor component did not keep track of any state or make intelligent guesses based on the previous observations or knowledge of the plant. It just reported the exact observations as received from the CarTracker. These design choices led to car under-reporting and mistaken identity.

To uncover the impact of these choices, a set of car identity experiments were conducted with Cars 1, 4, and 5. Each experiment was conducted with a single car equipped with rooftop 1, 4, or 5; the car was manually driven about randomly on the left end of the platform. Debug information was added to the CarTracker component which yielded each blob set and car identity identified at every iteration. A small python script, carId.py, was used to extract the statistical performance of each car[2].

[2]For those interested in repeating these experiments, information and source code are

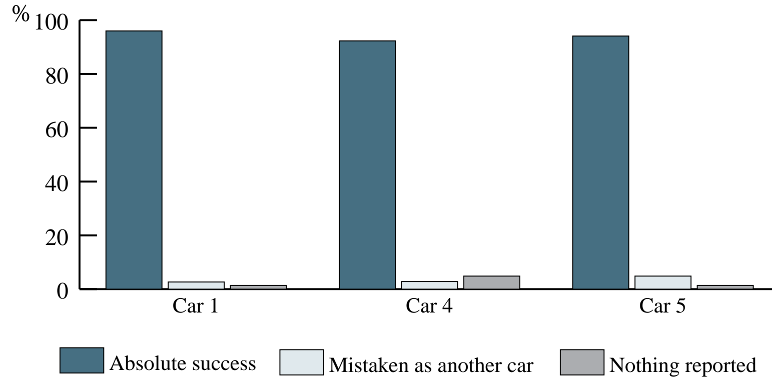


Figure 11.6: The success rate for car identification in the original vision subsystem. In the original CarTracker component, when Car 1 is the only car driving, it is reported correctly about 96% of the time. However, 3% of the time, it is mistakenly reported as another car. This leaves 1% of the time when nothing is reported at all.

Fig. 11.6 summarizes the performance of identifying a single car at a time. Car #1 was successfully identified 96% of the time. Car #4, 92%. Car #5, 94%. At the surface, mistaken identities might seem reasonable. It is only 4% of the time, right? No. Bad doggie, no biscuit. Fault-tolerant engineers would scoff at these kinds of numbers. These people demand "five nines" out of their reliable systems.

Worse, Car #1 was mistaken to be as many as *eleven* other cars. These mistaken identities lead to less stability in the control of the cars.

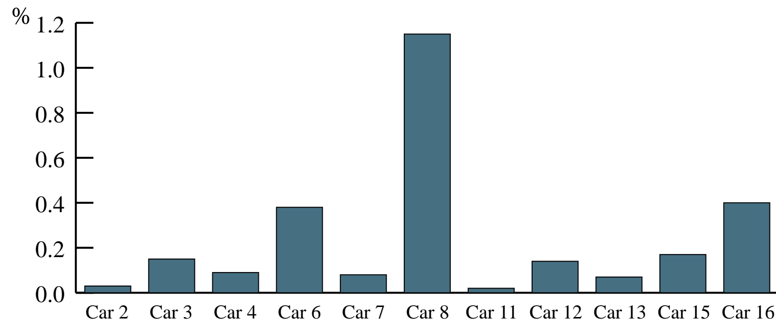


Figure 11.7: The many faces of Car 1. It is mistakenly reported as one of eleven other cars approximately 3% of the time when using the LeftConfigTracker_1.dat shown in Appendix A.

Thus, the second motivation for redesigning the vision subsystem was to remove the dependency inversion between the vision subsystem and the controller: to use previous observations and knowledge of the plant to more intelligently interpret the current observations and avoid as many mistaken identities as possible. Doing so would give the controller as useful information as possible so

available on the convergence laboratory wiki under the "Color Patches" link. Visit the internal wiki at https://agora.cs.uiuc.edu/display/convergencelab/Home. Members only.

that it might calculate safe control commands.

Step 3: Aggressive filtering of the car list results in unsafe silence

The original CarTracker component chose silence above doubt. For a lonely Car #4 driving solo around the network of roads, the CarTracker component reported no observations almost 5% of the time. For a mission-critical system like the testbed, this 5% is unacceptable. Yet, the third step of the vision algorithm further reduced the rate at which cars were reported.

The Vision Server expects a list of unique cars and locations; due to mistaken identities, it is possible to have a list of observed cars which contains multiple entries for the same car. Thus, the final step is to filter the list of observed cars, and keep only the unique entries that have the greatest confidence. As a result, if two cars are observed as Car #1, only one of them is reported.

To uncover filtering's impact, an experiment was performed with Cars 1 and 4. Using the configuration file LeftCarTrackerConfig_1.dat, Car 4 is observed 95% of the time. It is mistaken as Car #1 4.7% of the time. With Car #1 in an easily identifiable location on the driving platform and Car #4 driving randomly, I measured the number of cars *observed* after step 2 and compared it to the number of cars that are *reported* after step 3. Keep in mind that this experiment did not test the correctness of cars that were actually reported; it was only meant to examine how the number of cars reported was reduced.

Despite under-reporting due to step 2, the percentage of cars that were observed after step 2 was 97.7%. However, list filtering reduced further the number of cars that were reported to 96.9%, and this was only for two cars. As the number of cars active in the testbed increase, so can the amount of underreporting.

Thus, the third and final motivation for redesigning the vision subsystem was to remove the dependency inversion between the vision subsystem and the controller: to avoid under-reporting. Resolving the mistaken identity problem partially solves this problem; but even in cases where mistaken identity could not be resolved correctly, the aim was to report each time that some car was observed. Once again, the controller deserves as much information as possible to do its job.

11.2 Be as accurate as possible

Remember the mantra from the beginning of this chapter:

Be as accurate as possible; never keep secrets

The first part of my mantra, and the first step to stable control, is making accurate observations. Bad observations can quickly lead to instability. Observations are made in the testbed with two cameras mounted to the ceiling. These

two cameras are the left and right eyes of the testbed, tracking the location and orientation of every car driving in the testbed.

Making accurate observations in the testbed is not straightforward. The cameras, and the software that supports them, each comprise one Vision Sensor. This pair must observe up to 54 color patches moving about, and intelligently interpret those color patches as nine different vehicles, each with its own orientation and position.

Correcting step 1: Buying better paper yields better observations

The Model Builder problem is to build a model which approximates the real world. The resulting model acts as the software's internal view of the real world. I pose that solving this problem is simplified if the real world is easy to observe.

For the original testbed, this was not the case. The little paper quilt on top of every car was made from two kinds of paper: one very good and one very bad. The very good paper was observable 95-99% of the time. The very bad paper was observable 7-74% of the time, with many cases of mistaken color identification.

By replacing the bad paper with more brilliant colors found at Kinko's and Michael's[3], and making some minor changes to the configuration file, Fig. 11.8 shows the significant improvements that were made to rates of successful observation for each color. Even better, both dark blue and dark green are no longer mistaken as any other color.

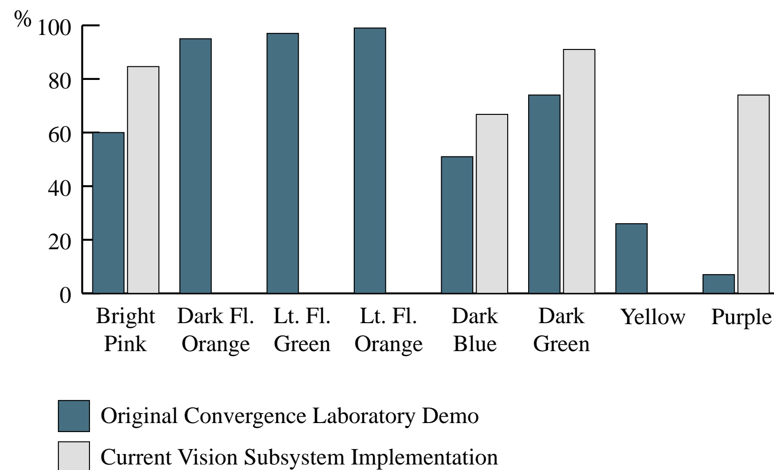


Figure 11.8: Absolute success comparison of observing color patches using the CarTracker C++ module as configured with LeftConfigTracker_2.dat seen in Appendix A.

What Fig. 11.8 also shows, however, is that no improvements have been made to the yellow color. I was not able to find a color that could replace the

[3]The convergence laboratory wiki describes all of the latest colors with instructions on where to buy them.

old yellow color and not be mistaken as any of the existing colors. The small solution to this small problem is not clear to me at this time.

11.3 Never keep secrets

Accurate observations are great, but alone they are not enough. With the new paper, the bright pink color is observed correctly about 80% of the time. What happens to observations during the other 20% of the time and how should the controller react? Ideally, there must be some way of indicating a problem with the observations and if there is a problem, it should never be kept secret.

In the original implementation, there were two problems:

- **A stateless and secretive Vision Sensor**. The Vision Sensor did not use past observations when interpreting sets of color patches as cars. What was interpreted as Car #1 in one moment would just as easily be interpreted as Car #4 in the next. Even worse, if two cars were interpreted as Car #4, the Vision Sensor would only report one of them.

- **A Controller in the dark**. Because the Vision Sensor did not report confidence with car information, the Controller had to accept as fact every observation it was handed and plough forward to the next waypoint.

Despite the significant improvements made to the success of color patch observations, cars would still be victims of mistaken identity if not for further improvements; color patches are not observed 100% of the time, and the CarTracker's guesses at identity are sometimes wrong.

Correcting step 2: Using the laws of physics to throw out bad observations

Fortunately, the laws of physics must prevail in the testbed. I employed two simple heuristics to filter out the possible mistaken identity observations that were presented by the CarTracker. These heuristics were based the two most frequent situations in which mistaken identity took place.

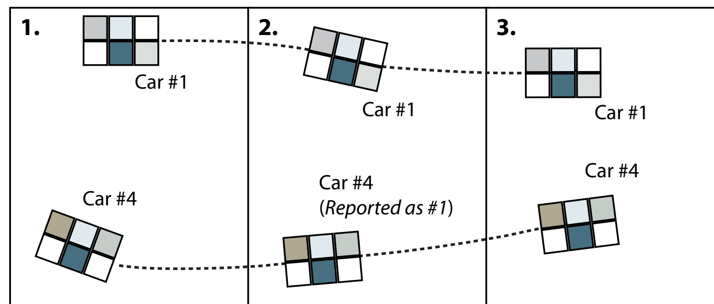


Figure 11.9: A car cannot travel faster than its maximum velocity.

Heuristic 1: A car cannot travel faster than its maximum velocity. One of the most frequent situations of mistaken identity in the testbed is depicted in Fig. 11.9. Two cars, #1 and #4, are traveling a safe distance apart. In frame 2, Car #1 is correctly identified and reported as Car#1. However, Car #4 is also reported as Car #1. Yet, for that second observation to be correct, Car #1 would have had to travel faster than its maximum velocity. Impossible. I throw out the second observation.

In C++, this heuristic is implemented as follows in CarTrackerProxyWith-Model.java:

```
while (prevObsvCarIter.hasNext()) {

    // Get the latest observation
    int thisIndex = prevObsvCarIter.next().intValue();
    BlobInfo thisObsv = parsedBlobObsv.get(thisIndex);

    // Get the most recent observation for this same car.
    BlobInfo lastObsv =
        __lastObsvHashMap.get(thisObsv.getCarID());

    // Calculate the actual distance traveled by this car
    // since the last location observation.
    double actualDistance =
        calculateDistance(thisObsv.getX(), lastObsv.getX(),
                          thisObsv.getY(), lastObsv.getY());

    long timeSinceLastObservation =
        timeStamp - __lastProbeTimeStamp.getValue();

    // Calculate the maxiumum possible distance traveled
    // by this car since the last location observation.
    double maximumPossibleDistance =
        timeSinceLastObservation * __MAX_MM_PER_MILLISECOND;

    // Confirm that the actual distance traveled by the car is
    // less than the maximum possible distance traveled.
    if (actualDistance >= maximumPossibleDistance) {

      // The car is too far from where it should be.
      // Throw it out.
      parsedBlobObsv.remove(thisIndex);
    }
}
```

Heuristic 2: Two cars cannot be in the same place at the same time. The other frequent situation for mistaken identity is depicted in Fig. 11.10. Car #1 is reported correctly in frame 1. In the next observation, frame 2, it is reported as Car #2. Car #2 is currently not driving on the testbed, so this is the very first observation of this car. Yet, this observation of one car is just a few millimeters away from the last observation of an entirely different car: Again, impossible. Throw out the second observation.

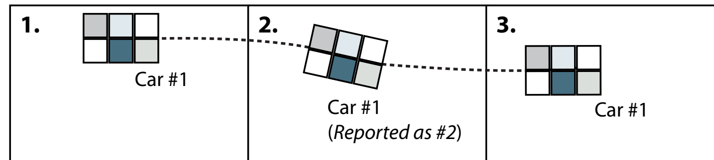


Figure 11.10: Two cars cannot be in the same place at the same time.

In C++, this heuristic is implemented as follows in CarTrackerProxyWithModel.java:

```
Iterator<String> lastObsvIter =
   __lastObsvHashMap.keySet().iterator();

while (newCarIter.hasNext()){

  // Get new observation
  int thisIndex = newCarIter.next().intValue();
  BlobInfo thisObsv = parsedBlobObsv.get(thisIndex);

  // Compare each new observation to every
  // car's last observation.
  Iterator<String> lastObsvIter =
    __lastObsvHashMap.keySet().iterator();

  while (lastObsvIter.hasNext()) {

    String lastIndex = lastObsvIter.next();

    double distance = calculateDistance(thisObsv.getX(),
        __lastObsvHashMap.get(lastIndex).getX(),
        thisObsv.getY(),
        __lastObsvHashMap.get(lastIndex).getY());

    double delta =
        thisObsv.getTimestamp() -
           __lastObsvHashMap.get(lastIndex).getTimestamp();
```

```
    if (distance < (__MAXIMUM_DISTANCE_BETWEEN_OBSERVATIONS)) {

      // Double check that IDs are different.
      if (! thisObsv.getCarID()
        .equals(__lastObsvHashMap.get(lastIndex).getCarID())){

// Throw it out
parsedBlobObsv.remove(thisIndex);

} } }
  }
```

Correcting step 3: Reporting all valid observations avoids unnecessary underreporting

In the original implementation, the final step is to filter the list of observed cars, and keep only the unique entries that have the greatest confidence. As a result, if two cars are observed as Car #1, only one of them is reported.

In the current implementation, this problem is solved simply by omitting this step. The C++ module, CarTracker, is no longer allowed to censor its own observations.

Extra Step: Mitigating an asynchronous mode of communication between CarTracker and Vision Sensor

The CarTracker is a C++ module. The Vision Sensor is a Java component executing on top of Etherware. The CarTracker communicates its observations to the Vision Sensor via a BufferedStream. This mode of communication is completely asynchronous. Moreover, it's polling-based. As a result, a brief section from an experiment's log file shows how observations can build up:

```
0.  * * *
1.  Time: 1206841461023 ms
2.  blob 01 1360 790 72 114 2 4 0 1 3 5 1206841461093
3.  Reported: 01
4.  * * *
5.  Time: 1206841461173 ms
6.  * * *
7.  Time: 1206841461173 ms
8.  blob 01 1365 837 77 114 2 4 0 1 3 5 1206841461243
9.  blob 01 1365 861 81 113 2 -1 0 1 3 5 1206841461304
10. Reported: 01
```

Each iteration of Vision Sensor is marked by the * * *. Thus, on line 2, CarTracker reports a single observation of Car #1 to the Vision Sensor. On line 3, the Vision Sensor reports Car #1. However, line 5 shows that no observations were provided in the next iteration. But lines 8 and 9 show an "extra" observation for that particular iteration.

The previous heuristics would incorrectly throw out multiple cars being reported in the same location; one last heuristic must be introduced. It is simply, "One car can be in the same place at the same time."

The results of following the mantra

By following the mantra, the vision subsystem now provides observations that are as accurate as possible; moreover it does not keep secrets.

Car	Reported Correctly	Reported As Another Car	Unreported	Not Observed	Mistaken As Another Car
#1	99.91%	Never	0.09%	0.02%	0.07%
#4	99.91%	Never	0.09%	0.01%	0.10%
#5	99.66%	Never	0.34%	0.015%	0.325%

Figure 11.11: Improving colors, maintaining state, and using some simple heuristics yields at least 99% success for car identity reporting.

Between the new paper and the new two heuristics, the identification success of cars in the lab is happily at least 99%, as shown in Fig. 11.11. The vicious cycle of dependency inversions has been broken. The plant is amenable to observation. The vision sensor gives as much useful information as possible to the controller. The controller uses this information to issue control commands to the plant. No one component in this path of critical components is an unreliable one

One might wonder if throwing out observations is a good idea. Thankfully, as shown in Fig. 11.11, cars equipped with new colors are not victims of mistaken identity all that often. In fact, Car#5 is mistaken as other cars the most, but that happens only 0.325% of the time. As a result, these heuristics are not only simple, they do not need to be employed very often either. This means that throwing out observations, while at first a potentially harmful risk, is just fine for the testbed. Bad observation? Throw it out; another will be along in about 34 milliseconds.

11.4 The resulting Simplex architecture

With my lengthy discussion of the vision subsystem, it might be easy to forget the main purpose of this chapter: Simplex in the convergence lab. The identification success rate is at least 99%, but what does this mean for Simplex at the architectural level?

Again, remember my mantra:

Be as accurate as possible; never keep secrets

The dependency inversions have been eliminated. The vision subsystem reports everything it observes as accurately as possible. If an observation does not make sense according to the laws of physics, it just throws it out. This means that other components receiving reports from the vision sensor can rely on this important fact: Silence means, "I don't know." As a result, other components can take it upon themselves to act in an appropriate manner when reports have not been received from the vision subsystem.

At first it may seem like a bad idea to use radio silence as a way of indicating uncertainty. But for Cyber-Physical Systems it is actually ideal. It aids in schedulability. Radio silence means no extra messages clogging up precious resources in a real-time system. It aids in reducing complexity. Radio silence means that even if the connection between the vision subsystem and the rest of the testbed is completely severed, the reaction subsequent components is exactly the same; the safety mechanism is a passive one and there is no extra code to test.

With that in mind, I return the decomposed trio of Simplex problem frames and explain what each means for the testbed:

- **Model Builder**. Build a machine which constructs a safe approximation of the environment. The vision subsystem must provide an accurate approximation of the cars and their locations. If an observation defies the law of physics, do not report it.

- **Connection Domain for Minimal Behavior**. Build a machine which controls some part of the physical world; implement a set of minimal functionalities while guaranteeing core safety requirements. From a pure safety perspective, there is no minimal functionality. The cars can sit perfectly still. The core safety requirement is that the cars must not collide.

- **Connection Domain for Desired Features**. Build a machine which implements the set of desired features only if the high-confidence safety guarantees are met. If car locations and identities have been received recently, then a controller may direct cars to their next waypoint.

The Simplex architecture is a solution-creating technique for combining two algorithms such that a system retains the safety of the first while gaining the features of the second. For the testbed, the first algorithm is the original control algorithm used by convergence laboratory to get the cars from one waypoint to the next. The second is also from the original platform: the Avoidance application extension used to divert the original control commands from the actuator and edit them with stop commands when a collision is imminent.

Choosing between these two algorithms is easy. Because the vision subsystem uses radio silence to indicate uncertainty, the decision machine only needs to make the following choice, “Go if there has been a report. Stop if there has not.”

The resulting architecture depicted in Fig. 11.12 summarizes the solution [4]. The collection of components make up a control loop. The Vision Sensor observes the color panels in the plant. This time, color panels are converted into events. Each event is a list of most recently observed cars intelligently interpreted by the sensor. This list reports their identities, locations, and orientations. Events are received by three components: the Trajectory Planner, Controller, and Collision Avoidance Supervisor.

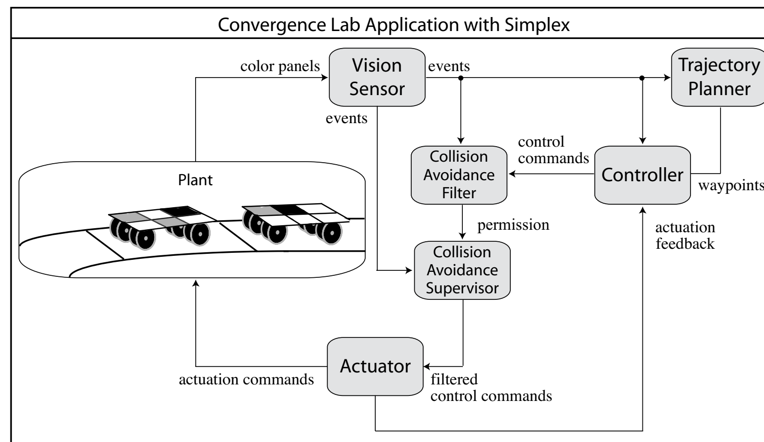


Figure 11.12: Logical architecture for the Simplex solution in the convergence laboratory testbed.

The functionality of the Trajectory Planner and Controller is identical to the original implementation. The Trajectory Planner uses the car information to revise the overall path that each car must take to achieve its goals. This overall path is maintained as a set of waypoints. The Controller also uses the car information in order to calculate a shorter path: the path from a car’s current location to the next waypoint determined by the Trajectory Planner.

The Controller sends control commands to the Actuator. These control commands are intercepted by a Collision Avoidance Filter. As a result, any time that a car’s location has not been reported in a recent time window, the Collision Avoidance Supervisor gives permission to the Collision Avoidance Filter to alter the control commands to stop commands; this avoids potential collisions that could result. The Actuator receives these filtered control commands; it transmits the commands to the physical plant and completes the loop.

Reorganizing the components of Fig. 11.12 into Fig. 11.13 makes the architecture look more like Simplex. The Simplex decision machine, or the Collision

[4]Please keep in mind that the full solution depicted here has not been implemented in the testbed. Because of the 99% identification success rate, I was very hesitant to add more code to a complex testbed already in transition between an old fleet of cars and a new one.

Avoidance Supervisor, chooses between the control command of the car Controller or the filtered command from the Collision Avoidance Filter.

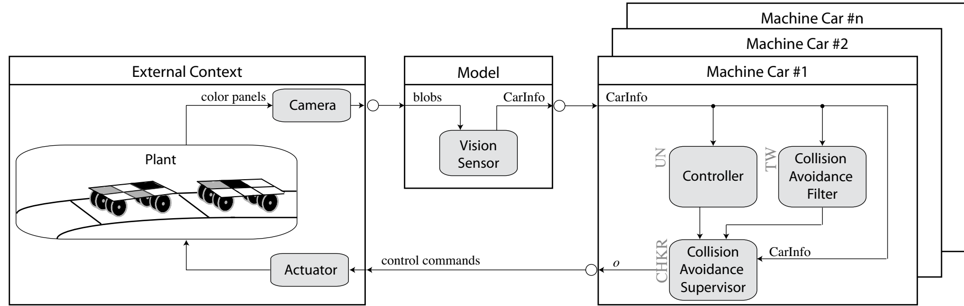


Figure 11.13: The "Simplex Looking" logical architecture. The Simplex decision machine, or the Collision Avoidance Supervisor, chooses between two commands based on the confidence of the car information sent by the Vision Sensor. The Supervisor's mantra, "Go if you know, stop of if you don't."

11.5 Conclusion

For a time, the convergence laboratory suffered a secretive and inaccurate vision subsystem; safety was at risk. Thanks to the lessons of Simplex and a simple mantra, the vision subsystem now yields an identification success rate of at least 99%. Because it is well established that the vision subsystem is radio silent when car locations are unknown, other subsystems can react appropriately in a Simplex fashion. Other Simplex prototypes have focused on control laws and decision machines; this is the very first to illuminate the challenge of building a modeling machine.

12 Related work

Simply put, my work is as much about Simplex as it is about describing Simplex architectures. My work provides a collection of precise, logical descriptions of the Simplex architecture in four different modeling paradigms. As a result, developers now have a formulaic design approach to utilizing and analyzing the Simplex architecture for their own applications.

In the early days, Simplex began as a prototype, and then I developed it into a design pattern [15]. While prototypes and design patterns are helpful for learning about examples, they do not facilitate the automatic analysis of software architectures demanded by Cyber-Physical Systems and the avionics domain.

My solution looks to architecture description languages and formal methods to more precisely describe the Simplex architecture and automate its analysis. The marriage of design patterns and formal methods is not a new idea, but it is a very good one. The key goal of this union is to bridge the gap between an intuitive understanding of a software architecture and the rigorous mathematics which can yield proofs of it. It is a challenge to balance this union such that developers have tools which are "rigorous but practical" [53].

There are a number of related areas to my work. I discuss these in turn.

12.0.1 Patterns

The notion of software patterns was introduced in [29]. Three from this original catalog are present in the Etherware middleware and are evaluated in Chapter 3. The *Memento* pattern records the internal state of an object. This record can be used to later restore the state of the object. *Façade* provides a simplified, high-level interface for a set of objects. Clients invoking these objects' services don't have to be concerned with the differences between their varying interfaces, and can just invoke services using the *Façade*. A *Proxy* is used to instantiate an object in place of another. In many cases, a *Proxy* creates a local place holder for a remote object. The use of a fourth pattern, the *Filter* [32], is also evaluated. A *Filter* allows dynamic compositions of objects to perform transformations on streams of data.

A real-time system is a reactive system whose correctness depends on satisfying both functional and temporal requirements [48]. Some of the earliest design patterns related to real-time systems are described in [62]; including *Pedestal*, a layered approach to organizing mechanical-process systems. In *Pedestal*, the

lowest layer is the **Real World**, consisting of actuators, solenoids, and sensors. Higher layers consist of the software used to model, manage, and interface to the **Real World** layer.

A real-time system is safety critical when its incorrect behavior can directly or indirectly lead to a state hazardous to human life [44]. Decisions which shape the software architecture for safety-critical, real-time systems are driven in part by three qualities; availability, reliability, and robustness [44, 9].

Availability, α, can be quantified by the probability that a system is available when needed. Availability is defined in [9] as,

$$\alpha = \frac{\text{mean time to failure}}{\text{mean time to failure} + \text{mean time to repair.}}$$

To increase availability in software, developers can use fault-detection or fault-recovery tactics [9]. For example, one fault-detection tactic is to employ a *heartbeat*, a periodic signal emitted by one component and monitored by another.

Reliability can be quantified by the probability that a component will perform its intended function satisfactorily for a prescribed time and under stipulated environmental conditions [44]. Reliability is important to safety-critical, real-time systems for it insures that components correctly execute to completion and meet their deadlines. Reliability patterns for real-time systems are documented in [19]. A *Watchdog* pattern can be used to monitor the internal, time-dependent computational progress of a subsystem. If the progress does not proceed to specification, the *Watchdog* can issue a shutdown or restart signal to the component.

Robustness is the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions [54]. One approach to robustness is to ensure that components or subsystems of components enter a fail-safe state under some conditions. For some systems, a fail-safe state is a depowered mode. For autonomous unmanned aerial vehicles, it is a low-energy descent. For radio-controlled boats which have lost their radio signal, it is turning in a large circle. The *Safety Executive* [19] is one pattern which describes how to coordinate a component's entry into its fail-safe state.

There are software patterns documented specifically for middleware. *QoS Contract* [49], uses three components to specify quality of service expectations, measure quality of service conditions, and adapt system behavior for measured quality of service. Used together in the QuO framework, these three components create a mechanism for distributed decision-making about resource management in real-time, embedded middleware.

Snapshot [49] describes how middleware may obtain an approximate view of the system's state for use in real-time, adaptive systems. *Snapshot* disperses a set of system condition objects throughout the system. These objects are queried for their object values and collected by an aggregator to construct an approximate view.

The *Quality Connector* architectural pattern [16] describes how to decouple application components from infrastructure components so that applications need not be concerned with non-standard quality of service interfaces. Instead, applications can make quality of service configurations and these configurations are adapted to the possibly evolving infrastructure components by the *Quality Connector*, even when the system enters various mode changes.

This dissertation presents a case study of patterns applied to one software system in the real-time, safety-critical domain. Other case studies in similar domains have been published. A broad road map of real-time patterns is provided in [81] which highlights multiple case studies of patterns in the real-time domain including those done on a microntroller-based fire alarm system and a vessel control system. A case study of patterns applied to a large-scale embedded applications for military mission planning is available in [67]. My work complements these, further advancing the understanding of patterns' utility in real-time, safety-critical systems. Moreso than the other case studies, this paper advances the depth of understanding by detailing four patterns' implementation and impact on one middleware for networked control.

Aside from Etherware, other middleware architectures have been proposed for networked control systems and focus on different aspects of the problem domain. A middleware that modifies the controller gain based on network load conditions while leaving the original controller unaffected is described in [76, 77]. In [72], a middleware layer is inserted into the IP stack in order to develop a fault-tolerant Ethernet for use in networked control applications. The need for design objectives and problems associated with building embedded real-time middleware applications with COTS are summarized in a tutorial paper [63], and [11] provides an overview of some of the middleware applications available for distributed systems with an emphasis on control systems.

12.0.2 Model-checking real-time systems

For my investigation into describing the Simplex architecture, I chose Maude as one candidate for clear political reasons. There are other formalisms that support finite automata, linear temporal logic, and model-checking, such as NuSMV [12] or SPIN [34]. Still, there are many other powerful and novel formalisms available for modeling real-time systems. Among these are [36, 41, 61, 64].

Message Sequence Charts [36] were adapted by the International Telecommunications Union in 1996. They are a visual and textual formalism for expressing message passing between a collection of independent "instances" or processes. Multiple extensions have been created, including Triggered Message Sequence Charts [64] and Symbolic Message Sequence Charts [61]. One characteristic of MSC is that developers describe scenarios of messages passed between processes. This is very apt for the telecommunications domain, but it is a challenge

to describe a reactive system such as a pacemaker using only a collection of scenarios.

The quantitative verification methodology [41] allows developers to check probabilistic properties about a system. Based on Markov chains, developers describe their systems using a state automata with additional probabilistic transitions. This kind of examination is valuable for real-time systems, whose safety properties are often measured quantitatively: *the probability that the pacemaker battery will expire after 5 years is* 1×10^{-7}. While there is certainly value in verifying such properties, I have not opted to explore probabilistic models.

12.0.3 Formalizing patterns

Various approaches to the formalization of software patterns have been studied, including [53, 73]. In 1998, Mikkonen introduced formalizing temporal behaviors of design patterns at a high level of abstraction [53]. Based on Leslie Lamport's temporal logic of actions, developers may specify patterns using the primitives *class*, *relation*, and *action*. In 2004, Soundarajan et al. introduced another paradigm for formalizing design patterns; they borrow the concepts introduced by Design by Contract [52] and describe a pattern's *responsibilities* and the *rewards* of abiding by them.

One limitation of my work is the language in which developers must express an architecture's properties. When using Maude, for example, developers are faced with expressing properties in the sometimes limited Linear Temporal Logic, or LTL. Other work uses patterns to simplify the formulation of a system's properties. In 1998, Dwyer et al. offered a collection of well-understood ways for specifying properties of a finite state machine. These "property specification patterns" [21] recognize the steep learning curve for specifying formal properties; 500 property specifications were surveyed to result in eight patterns. Since 1998, Dwyer's patterns have become incorporated into the Bandera project, an integrated toolset for model checking concurrent Java software [20].

Others have extended property specification patterns. Propel, is an approach for expressing properties in both finite-state automata and a disciplined natural language; it is a set of templates based on Dwyer's collection [69]. Since its inception, Propel has become a question tree-based system for guiding developers through the options that must be considered for using the patterns for their own system [5]. Other extensions of Dwyer's work for the real-time domain are also available. Templates for real-time temporal logic specifications are provided by Konrad and Cheng in three different temporal logics [40].

My work may benefit from these kinds of property specification patterns and certainly future work out to consider these.

A Vision configuration files

These are the configuration files for the Matrox Imaging Library to define the 8 colors used in the testbed.

```
SERVERNUM
2
CALIBRATION(hueL,hueH,satL,SatH,lumL,lumH,patchSizeL,patchSizeH)
211 245 1 255 78 255 15 250 //BrightPink
0 19 60 255 50 255 10 250 //DeepFlourescentOrange
41 60 80 255 30 255 7 350 //LightFlourescentGreen
20 31 70 140 50 255 10 350 //LightFlourescentOrange
144 184 1 255 15 255 5 200 //DarkBlue
45 100 1 90 20 160 7 300 //DarkGreen
31 42 80 120 60 255 20 300 //Yellow
183 226 2 90 18 125 5 250 //Purple
```

Figure A.1: LeftCarTrackerConfig_1.dat

```
SERVERNUM
2
CALIBRATION(hueL,hueH,satL,SatH,lumL,lumH,patchSizeL,patchSizeH)
198 255 1 255 53 255 15 200 //BrightPink
0 18 60 255 38 255 10 300 //DeepFlourescentOrange
53 67 75 255 45 255 7 300 //LightFlourescentGreen
20 32 70 140 50 255 10 300 //LightFlourescentOrange
142 177 1 255 15 255 7 200 //DarkBlue
51 97 33 90 16 130 7 255 //DarkGreen
36 47 34 120 29 175 15 310 //Yellow
164 205 28 138 18 111 5 175 //Purple
```

Figure A.2: LeftCarTrackerConfig_2.dat

B Introductory sensor example in AADL

```
--------------------------------------------------------
-- Sensor Package
--------------------------------------------------------
package pkgSensor
public
  data int
  end int;

  data string
  end string;

  data errorData
  end errorData;

  data voltageValue
      properties
         Source_Data_Size => 16 bits;
   end voltageValue;

   data sensorData
      properties
         Source_Data_Size => 16 bits;
   end sensorData;

   device dvcSensor
       features
         Input : in event data port voltageValue;
         Output: out event data port sensorData;
   end dvcSensor;

  device dvcSensorWithError extends dvcSensor
     features
        InternalError: out event data port errorData;
  end dvcSensorWithError;
```

```
  data implementation sensorData.temperature
      subcomponents
         temperature : data int;
         measurementUnits : data string;
  end sensorData.temperature;

end pkgSensor;
```

C Generic packages to aid with instantiating Simplex architectures in AADL

```
-----------------------------------
-- Actuator Package
-----------------------------------
package pkgActuator
public

  -- The generic actuator takes a control
  -- command as input and transmits an
  -- actuation command as output.
  process proActuator
    features
      iptCmd: in event data port ;
      optActuation : out event data port ;
  end proActuator;

  process implementation proActuator.impl
     subcomponents
        tidA : thread thrActuator ;
     connections
        event data port iptCmd -> tidA.ipthrCmd ;
        event data port tidA.opthrActuation -> optActuation ;
  end proActuator.impl;

  thread thrActuator
     features
        ipthrCmd: in event data port ;
        opthrActuation : out event data port ;
   end thrActuator;

end pkgActuator;

--------------------------------------------------------
-- Controller Package
--------------------------------------------------------
package pkgController
```

```
public

  -- The Controller accepts model data from the model builder,
  -- and issues a command.
  process proController
    features
      iptModelData: in event data port ;
      optCmd: out event data port ;
  end proController;
end pkgController;

-----------------------------------
-- Plant Package
-----------------------------------
package pkgPlant
public

  -- The generic plants accepts an actuation
  -- command and ''transmits'' a physical
  -- phenomenon.
  process proPlant
    features
      iptActuation: in event data port ;
      optPhysPhenom : out event data port ;
  end proPlant;

  process implementation proPlant.impl
     subcomponents
        tidP : thread thrPlant ;
     connections
        event data port iptActuation -> tidP.ipthrActuation ;
        event data port tidP.opthrPhysPhenom -> optPhysPhenom ;
  end proPlant.impl;

  thread thrPlant
     features
        ipthrActuation: in event data port ;
        opthrPhysPhenom : out event data port ;
   end thrPlant;

end pkgPlant;

-----------------------------------
```

```
-- Sensor Package
-----------------------------------
package pkgSensor
public

  -- The generic sensor takes a physical
  -- phenomenon as input and transmits a
  -- monitored variable as output.
  process proSensor
    features
      iptPhysPhenom: in event data port ;
      optMonVar : out event data port ;
  end proSensor;

  process implementation proSensor.impl
     subcomponents
        tidS : thread thrSensor ;
     connections
        event data port iptPhysPhenom -> tidS.ipthrPhysPhenom ;
        event data port tidS.opthrMonVar -> optMonVar ;
  end proSensor.impl;

  thread thrSensor
     features
        ipthrPhysPhenom: in event data port ;
        opthrMonVar : out event data port ;
   end thrSensor;

end pkgSensor;
```

D Generic Simplex model in AADL

```
-----------------------------------
-- Generic Simplex Reference Model
-----------------------------------
package simplex
public

-----------------------------------
-- DECISION MACHINE
-----------------------------------

  --- The decision machine chooses
  --- between the simple control command
  --- and the complex control command.

  process proDecisionMachine
    features
       iptSCmd : in event data port ;
       iptCCmd : in event data port ;
       optCmd : out event data port ;
  end proDecisionMachine;

  process implementation proDecisionMachine.impl
     subcomponents
        tidDM : thread thrDecisionMachine ;
     connections
        event data port iptSCmd ->
           tidDM.ipthrSimpleCmd ;
        event data port iptCCmd ->
           tidDM.ipthrComplexCmd ;
        event data port tidDM.opthrCmd -> optCmd ;
  end proDecisionMachine.impl;

  thread thrDecisionMachine
    features
      ipthrSimpleCmd : in event data port ;
```

```
     ipthrComplexCmd : in event data port ;
     opthrCmd: out event data port ;
  end thrDecisionMachine;

-----------------------------------
-- MODELING MACHINE
-----------------------------------

  --- The modeling machine maintains
  --- the software's internal view of
  --- the hardware's state.

  process proModelMachine
     features
        iptMonVar : in event data port ;
        optSimpleModelData : out event data port ;
        optComplexModelData : out event data port ;
  end proModelMachine;

  process implementation proModelMachine.impl
     subcomponents
        tidMM : thread thrModelMachine ;
     connections
        event data port iptMonVar -> tidMM.ipthrMonVar ;
        event data port tidMM.opthrSimpleModelData
           -> optSimpleModelData ;
        event data port tidMM.opthrComplexModelData
           -> optComplexModelData ;
  end proModelMachine.impl;

  thread thrModelMachine
     features
        ipthrMonVar : in event data port ;
        opthrSimpleModelData : out event data port ;
        opthrComplexModelData : out event data port ;
  end thrModelMachine;

-----------------------------------
-- CONTROLLERS
-----------------------------------

  --- Simple Controller
  process proSimpleController
```

```
    extends pkgController::proController
  end proSimpleController;

  process implementation proSimpleController.impl
    subcomponents
      tidSC : thread thrSimpleController ;
    connections
      event data port tidSC.opthrSCmd -> optCmd ;
      event data port iptModelData -> tidSC.ipthrModelData ;
  end proSimpleController.impl;

  thread thrSimpleController
    features
      ipthrModelData : in event data port ;
      opthrSCmd : out event data port ;
  end thrSimpleController;

  --- Complex Controller
  process proComplexController
     extends pkgController::proController
  end proComplexController;

  process implementation proComplexController.impl
    subcomponents
      tidCC : thread thrComplexController ;
    connections
      event data port tidCC.opthrCCmd -> optCmd ;
      event data port iptModelData -> tidCC.ipthrModelData ;
  end proComplexController.impl;

  thread thrComplexController
    features
      ipthrModelData : in event data port ;
      opthrCCmd : out event data port ;
  end thrComplexController;


-----------------------------------
-- ASSEMBLING IT ALL TOGETHER
-----------------------------------

  --- Simplex Control Architecture Assembly
  --- The system takes as input a monitored variable;
```

```
  --- likely from some sensor monitoring the plant.
  --- The system transmits an output of a control
  --- command, likely to some actuator controlling
  --- the control variables in the plant.
  system Simplex
     features
        siptMonVar : in event data port ;
        soptCmd : out event data port ;
  end Simplex;

  system implementation Simplex.impl
    subcomponents
      idDM : process proDecisionMachine ;
      idMM : process proModelMachine ;
      idSC : process proSimpleController ;
      idCC : process proComplexController ;
    connections
      event data port siptMonVar -> idMM.iptMonVar ;
      event data port idMM.optSimpleModelData
         -> idSC.iptModelData ;
      event data port idMM.optComplexModelData
         -> idCC.iptModelData ;
      event data port idSC.optCmd -> idDM.iptSCmd ;
      event data port idCC.optCmd -> idDM.iptCCmd ;
      event data port idDM.optCmd -> soptCmd ;
  end Simplex.impl;

end simplex;
```

E Simple dot example in AADL

```
package simplexDotExample
public

-----------------------------------
-- THE ENTIRE SYSTEM ASSEMBLY
-----------------------------------

system sysDotSimplexExample
end sysDotSimplexExample;

system implementation sysDotSimplexExample.impl
   subcomponents
      idSDS : system sysDotSimplex ;
      idRW : system sysRealWorld ;
   connections
      event data port idSDS.soptCmd -> idRW.siptCmd ;
      event data port idRW.soptMonVar -> idSDS.siptMonVar ;
end sysDotSimplexExample.impl;

-----------------------------------
-- SOFTWARE
-----------------------------------

   system sysDotSimplex extends simplex::Simplex
      features
         soptCmd :
          refined to out event data port Behavior::integer;
         siptMonVar :
          refined to in event data port Behavior::integer;
   end sysDotSimplex;

   --- The software assembly is just an instance of the
   --- simplex assembly in simplex.aadl
   system implementation sysDotSimplex.impl
      extends simplex::Simplex.impl
```

```
   end sysDotSimplex.impl;

   --- Modeling Machine
   thread thrDotModelMachine extends simplex::thrModelMachine
      features
         ipthrMonVar :
            refined to in event data port Behavior::integer ;
 opthrSimpleModelData :
            refined to out event data port Behavior::integer ;
 opthrComplexModelData :
            refined to out event data port Behavior::integer ;
   end thrDotModelMachine;

   thread implementation thrDotModelMachine.impl
     properties
        Dispatch_Protocol => Aperiodic;
 annex behavior_specification {**
       states
          s0 : initial state ;
       state variables
         monVar : Behavior::integer ;
   mmXmit : Behavior::integer ;
       transitions
       --- Receive the monitored variable
          s0 -[ ipthrMonVar ? (monVar) ]-> s0
  { mmXmit := mmXmit + 1 ;
      opthrSimpleModelData ! (monVar) ;
        opthrComplexModelData ! (monVar) } ;
**} ;
   end thrDotModelMachine.impl;

   --- Decision Machine
   thread thrDotDecisionMachine
      extends simplex::thrDecisionMachine
      features
         ipthrSimpleCmd :
            refined to in event data port Behavior::integer;
         ipthrComplexCmd :
            refined to in event data port Behavior::integer;
         opthrCmd :
            refined to out event data port Behavior::integer;
   end thrDotDecisionMachine;
```

```
  thread implementation thrDotDecisionMachine.impl
   properties
       Dispatch_Protocol => Aperiodic;
annex behavior_specification {**
      states
            sReceive : initial state ;
         sReceiveSimple : state ;
       s1 : state ;
         s2 : state ;
         s3 : state ;
         s4 : state ;
         fail : state ;
      state variables
         aux : Behavior::integer;
         inSCmd : Behavior::integer;
         inCCmd : Behavior::integer;
         simpleRcpt : Behavior::integer;
         complexRcpt : Behavior::integer;
         distanceFromStart : Behavior::integer;
         safetyEnv : Behavior::integer;
         dotPos : Behavior::integer;
         wallPos : Behavior::integer;
       transitions
       --- Consume any incoming complex command.
       --- Conduct the safety check on the incoming
       --- complex command.  Assess how far from a starting
       --- point of 0 the dot is, plus a safety envelope.
          sReceive -[ ipthrComplexCmd ? (inCCmd) ]->
             sReceiveSimple
  { complexRcpt := complexRcpt + 1 ;
      distanceFromStart := dotPos + inCCmd + safetyEnv ; } ;
          --- Consume any incoming simple command
          sReceiveSimple -[ ipthrSimpleCmd ? (inSCmd) ]-> s2
       { simpleRcpt := simpleRcpt + 1; } ;

       --- Check if the complex command is safe and
       --- send it if it is safe.
  s2 -[ on (distanceFromStart < wallPos) ]-> s4
        { dotPos := dotPos + inCCmd ;
      opthrCmd ! (inCCmd); } ;

       --- Double-check that the plant is still safe.
  s4 -[ on (dotPos < wallPos) ]-> sReceive { aux := 0 ; } ;
```

```
        --- If the plant is no longer safe,
        --- then the system has failed.
   s4 -[ on (dotPos >= wallPos) ]-> fail {aux := 0 ; } ;

        --- If the complex command is not safe,
        --- send the simple command
   s2 -[ on (distanceFromStart >= wallPos) ]-> s3
         { dotPos := dotPos - inSCmd ;
        opthrCmd ! (inSCmd) ; } ;

        --- Double-check that the plant is still safe.
   s3 -[ on (dotPos < wallPos) ]-> sReceive { aux := 0 ; } ;

        --- If the plant is no longer safe,
        --- then the system has failed.
   s3 -[ on (dotPos >= wallPos) ]-> fail {aux := 0 ; } ;
**};
   end thrDotDecisionMachine.impl;

   --- Simple Controller
   thread thrDotSimpleController
    extends simplex::thrSimpleController
      features
         ipthrModelData :
            refined to in event data port Behavior::integer;
         opthrSCmd :
            refined to out event data port Behavior::integer;
   end thrDotSimpleController;

  thread implementation thrDotSimpleController.impl
     properties
        Dispatch_Protocol => Periodic;
 annex behavior_specification {**
       states
          s0 : initial state ;
          s1 : state ;
     state variables
           transmissions : Behavior::integer;
     simpleCommand : Behavior::integer;
       modelData : Behavior::integer;
       transitions
       --- Send the command and update the
```

```
      --- number of transmissions.
         s0 -[on true]-> s1
           { opthrSCmd ! (simpleCommand); }  ;
  s1 -[ipthrModelData ? (modelData) ]-> s0
          { transmissions := transmissions + 1; } ;
**} ;
  end thrDotSimpleController.impl;


   --- Complex Controller
   thread thrDotComplexController
     extends simplex::thrComplexController
      features
         ipthrModelData :
            refined to in event data port Behavior::integer;
         opthrCCmd :
            refined to out event data port Behavior::integer;
   end thrDotComplexController;

  thread implementation thrDotComplexController.impl
     properties
        Dispatch_Protocol => Periodic;
 annex behavior_specification {**
      states
         s0 : initial state ;
         s1 : state ;
      state variables
         cXmit : Behavior::integer;
         complexCommand : Behavior::integer;
         cModelData : Behavior::integer;
      transitions
      --- Send the command and update the
      --- number of transmissions.
         s0 -[on true]-> s1
            { opthrCCmd ! (complexCommand); }  ;
  s1 -[ipthrModelData ? (cModelData) ]-> s0
            { cXmit := cXmit + 1; } ;
**} ;
  end thrDotComplexController.impl;

-----------------------------------
-- HARDWARE
-----------------------------------
```

```
--- Hardware system assembly
system sysRealWorld
   features
      soptMonVar : out event data port Behavior::integer;
      siptCmd : in event data port Behavior::integer;
end sysRealWorld;

system implementation sysRealWorld.impl
   subcomponents
      idA : process pkgActuator::proActuator ;
      idS : process pkgSensor::proSensor ;
      idP : process pkgPlant::proPlant ;
   connections
      event data port siptCmd -> idA.iptCmd ;
      event data port idA.optActuation -> idP.iptActuation;
      event data port idP.optPhysPhenom -> idS.iptPhysPhenom;
      event data port idS.optMonVar -> soptMonVar ;
end sysRealWorld.impl;

--- Actuator
thread thrDotActuator
  extends pkgActuator::thrActuator
   features
      ipthrCmd :
         refined to in event data port Behavior::integer ;
      opthrActuation :
         refined to out event data port Behavior::integer ;
end thrDotActuator;

thread implementation thrDotActuator.impl
   properties
      Dispatch_Protocol => Aperiodic;
annex behavior_specification {**
     states
        sReceive : initial state ;
   state variables
     inRWCmd : Behavior::integer ;
           aRecpt : Behavior::integer ;
     transitions
     --- Receive an incoming command, forward
     --- it to the plant
     sReceive -[ ipthrCmd ? (inRWCmd) ]-> sReceive
```

```
         { aRecpt := aRecpt + 1 ;
   opthrActuation ! (inRWCmd) } ;
**} ;
  end thrDotActuator.impl;


  --- Plant
  thread thrDotPlant extends pkgPlant::thrPlant
     features
        ipthrActuation :
           refined to in event data port Behavior::integer ;
        opthrPhysPhenom :
           refined to out event data port Behavior::integer ;
  end thrDotPlant;

  thread implementation thrDotPlant.impl
     properties
        Dispatch_Protocol => Aperiodic;
 annex behavior_specification {**
       states
         sReceive : initial state;
       state variables
        inCmd : Behavior::integer ;
 realDotPos : Behavior::integer ;
             pRecpt : Behavior::integer ;
        transitions
        --- Receive an incoming command, update
        --- the position of the dot, and ‘‘send’’
        --- the dot position to the sensor.
        sReceive -[ ipthrActuation ? (inCmd) ]-> sReceive
          { pRecpt := pRecpt + 1 ;
            realDotPos := realDotPos + inCmd ;
            opthrPhysPhenom!(realDotPos) }  ;
**} ;
  end thrDotPlant.impl;

   --- Sensor
   thread thrDotSensor extends pkgSensor::thrSensor
      features
         ipthrPhysPhenom :
            refined to in event data port Behavior::integer ;
         opthrMonVar :
            refined to out event data port Behavior::integer ;
```

```
   end thrDotSensor;

   thread implementation thrDotSensor.impl
      properties
         Dispatch_Protocol => Aperiodic;
   annex behavior_specification {**
         states
            sReceive : initial state ;
     state variables
     inSignal : Behavior::integer ;
     sRecpt : Behavior::integer ;
         transitions
         --- Detect physical phenomenon
         --  in the plant.
            sReceive -[ ipthrPhysPhenom ? (inSignal) ]-> sReceive
       { sRecpt := sRecpt + 1 ;
 opthrMonVar ! (inSignal) } ;
 **} ;
   end thrDotSensor.impl;

end simplexDotExample;
```

F Single-thread example in AADL

```
---
--- A simple example demonstrating the connection
--- between AADL and the A2M interpreter.
---
--- The description below is parseable in the OSATE AADL
--- plug-in for Eclipse
---

package pkgSimpleExample

   public

--- The entire system.

   system sysWhole
      features
         none ;
   end sysWhole;

   system implementation sysWhole.impl
      subcomponents
         idSS: system sysSubsystem.impl;
      connections
         none ;
   end sysWhole.impl;

--- The subsystem .

   system sysSubsystem
      features
         prtEventOut: out event data port Behavior::integer;
   end sysSubsystem;

   system implementation sysSubsystem.impl
      subcomponents
```

```
      idSP: process proSimpleProcess.impl;
   connections
    event data port idSP.prtProcessOut -> prtEventOut;
end sysSubsystem.impl;

--- The process

process proSimpleProcess
   features
      prtProcessOut: out event data port Behavior::integer;
end proSimpleProcess;

process implementation proSimpleProcess.impl
    subcomponents
       tidST: thread thrSimpleThread.impl;
    connections
       event data port tidST.prtThreadOut -> prtProcessOut;
end proSimpleProcess.impl;

--- The thread

thread thrSimpleThread
   features
      prtThreadOut: out event data port Behavior::integer;
end thrSimpleThread;

thread implementation thrSimpleThread.impl
   subcomponents
      none ;
   connections
      none ;
   properties
      Dispatch_Protocol => periodic;
   annex behavior_specification {**
states
         s0 : initial state;
         s1 : state;
      transitions
       s0 -[]-> s1 { prtThreadOut!(1); };
**};
end thrSimpleThread.impl;

end pkgSimpleExample;
```

G Single-thread example in A2M

```
(tomod VERYSIMPLE is including AADL .

  --- Declare all the names and ids of things.
  ops sysWhole sysSubsystem : -> SystemName [ctor] .
  op impl : -> ImplName [ctor] .
  op proSimpleProcess : -> ProcessName [ctor] .

  ops prtEventOut prtProcessOut prtThreadOut : -> PortId [ctor] .

  ops idSS MAIN : -> SystemId [ctor] .
  op idSP : -> ProcessId [ctor] .
  ops s0 s1 : -> Location [ctor] .

  --- Thread id and name
  op tidST : -> ThreadId [ctor] .
  op thrSimpleThread : -> ThreadName [ctor] .

  --- -------------------------------------------------------
  --- -------------------------------------------------------

  var SI : SystemId .
  var PRI : ProcessId .
  var TI : ThreadId .

  --- The specification

  eq system(sysWhole) = none .

  eq SI system sysWhole . impl =
  < SI : System | features : system(sysWhole),
      subcomponents :
      ( idSS system sysSubsystem . impl ),
          connections : none
   > .
```

```
--- ----------------------------------------------------------
--- The Subsystem
--- ----------------------------------------------------------

  eq system(sysSubsystem) = (prtEventOut out event data port) .

  eq SI system sysSubsystem . impl =
  < SI : System | features : system(sysSubsystem),
      subcomponents :
      ( idSP process proSimpleProcess . impl ),
          connections :
  ( idSP . prtProcessOut --> prtEventOut )
  > .

--- ----------------------------------------------------------
--- The Process
--- ----------------------------------------------------------
  eq process(proSimpleProcess) =
      (prtProcessOut out event data port) .

  eq PRI process proSimpleProcess . impl =
       < PRI : Process | features : process(proSimpleProcess),
       subcomponents :
                    (tidST thread thrSimpleThread . impl),
               connections :
                    (tidST . prtThreadOut --> prtProcessOut )
       > .

--- ----------------------------------------------------------
--- The Thread
--- ----------------------------------------------------------
  eq thread(thrSimpleThread) =
  ports (prtThreadOut out event data thread port)
  dispatch periodic-dispatch(1) .

  eq TI thread thrSimpleThread . impl =
  < TI : Thread | features : ports(thread(thrSimpleThread)),
  subcomponents : none,
  connections : none,
  properties : dispatch(thread(thrSimpleThread)),
  status : completed,
          behavior :
                   (states
```

```
                        initial: s0 complete: s0 s1
                     transitions
                         s0 -[]-> s1 {prtThreadOut ! (1)})
   > .

 --- ----------------------------------------------------------
 --- The Whole Thing
 --- ----------------------------------------------------------

   op init : -> GlobalSystem .
   eq init = {MAIN system sysWhole . impl} .

endtom)
```

H Model-checking the single-thread example in A2M

```
(tomod MODEL-CHECK-SIMPLE is including VERYSIMPLE .

   op outputValue : Configuration ~> Bool [frozen (1)] .

   eq outputValue
   (< MAIN : System |
   subcomponents :
      (C1:Configuration
      < idSS : System |
      subcomponents :
            (C2:Configuration
            < idSP : Process |
            subcomponents :
               (C3:Configuration
               < tidST : Thread |
               behavior :
                  states
                     current: L:Location
                     complete: LS:LocationSet
                     others: LS1:LocationSet
                  state variables
                      VAL:Valuation
                  transitions TS:TransitionSet
                 >)
             >)
         >)
     >) = L:Location == s1 .

endtom)


(tsearch [1] init =>* {C:Configuration}
   such that outputValue(C:Configuration) with no time limit .)
```

I The simple dot example in the A2M

```
(tomod DOT-SIMPLEX is including AADL .

  --- Declare all the names and ids of things.
  ops sysSimplex sysSoftware sysRealWorld :
      -> SystemName [ctor] .
  ops idSS idSRW MAIN : -> SystemId [ctor] .
  op impl : -> ImplName [ctor] .

  --- Process Names and IDs
  op proComplexController : -> ProcessName [ctor] .
  op proSimpleController : -> ProcessName [ctor] .
  op proDecisionMachine : -> ProcessName [ctor] .
  op proPlant : -> ProcessName [ctor] .
  op proActuator : -> ProcessName [ctor] .
  op proSensor : -> ProcessName [ctor] .
  op proModelMachine : -> ProcessName [ctor] .

  op idSP : -> ProcessId [ctor] .
  op idSC : -> ProcessId [ctor] .
  op idCC : -> ProcessId [ctor] .
  op idDM : -> ProcessId [ctor] .
  op idP : -> ProcessId [ctor] .
  op idA : -> ProcessId [ctor] .
  op idS : -> ProcessId [ctor] .
  op idMM : -> ProcessId [ctor] .

  --- System level port names
  op siptCmd : -> PortId [ctor] .
  op soptCmd : -> PortId [ctor] .
  op soptMonVar : -> PortId [ctor] .
  op siptMonVar : -> PortId [ctor] .

  --- Process level port names
  op optSimpleCmd : -> PortId [ctor] .
  op iptSimpleCmd : -> PortId [ctor] .
```

```
op optComplexCmd : -> PortId [ctor] .
op iptComplexCmd : -> PortId [ctor] .
op optComplexCmdForSimple : -> PortId [ctor] .
op iptComplexCmdForSimple : -> PortId [ctor] .
op optCmd : -> PortId [ctor] .
op iptCmd : -> PortId [ctor] .
op iptRWCmd : -> PortId [ctor] .
op optActuation : -> PortId [ctor] .
op iptPhysPhenom : -> PortId [ctor] .
op optPhysPhenom : -> PortId [ctor] .

op optMonVar : -> PortId [ctor] .
op iptMonVar : -> PortId [ctor] .
op optSimpleModelData : -> PortId [ctor] .
op iptSimpleModelData : -> PortId [ctor] .
op optComplexModelData : -> PortId [ctor] .
op iptComplexModelData : -> PortId [ctor] .

--- Thread port names.
op opthrSimpleCmd : -> PortId [ctor] .
op ipthrSimpleCmd : -> PortId [ctor] .
op opthrComplexCmd : -> PortId [ctor] .
op ipthrComplexCmd : -> PortId [ctor] .
op opthrComplexCmdForSimple : -> PortId [ctor] .
op ipthrComplexCmdForSimple : -> PortId [ctor] .
op opthrCmd : -> PortId [ctor] .
op ipthrCmd : -> PortId [ctor] .
op ipthrRWCmd : -> PortId [ctor] .
op opthrActuation : -> PortId [ctor] .
op ipthrPhysPhenom : -> PortId [ctor] .
op opthrPhysPhenom : -> PortId [ctor] .
op opthrMonVar : -> PortId [ctor] .
op ipthrMonVar : -> PortId [ctor] .
op opthrSimpleModelData : -> PortId [ctor] .
op ipthrSimpleModelData : -> PortId [ctor] .
op opthrComplexModelData : -> PortId [ctor] .
op ipthrComplexModelData : -> PortId [ctor] .

--- State names
op s0 : -> Location [ctor] .
op s1 : -> Location [ctor] .
op sReceive : -> Location [ctor] .
op sReceiveSimple : -> Location [ctor] .
```

```
op s2 : -> Location [ctor] .
op s3 : -> Location [ctor] .
op s4 : -> Location [ctor] .
op fail : -> Location [ctor] .

--- Thread id and name
op tidST : -> ThreadId [ctor] .
op tidSC : -> ThreadId [ctor] .
op tidCC : -> ThreadId [ctor] .
op tidDM : -> ThreadId [ctor] .
op tidP : -> ThreadId [ctor] .
op tidA : -> ThreadId [ctor] .
op tidS : -> ThreadId [ctor] .
op tidOS : -> ThreadId [ctor] .
op tidCOS : -> ThreadId [ctor] .
op tidMM : -> ThreadId [ctor] .

op thrComplexController : -> ThreadName [ctor] .
op thrSimpleController : -> ThreadName [ctor] .
op thrDecisionMachine : -> ThreadName [ctor] .
op thrPlant : -> ThreadName [ctor] .
op thrActuator : -> ThreadName [ctor] .
op thrSensor : -> ThreadName [ctor] .
op thrModelMachine : -> ThreadName [ctor] .

--- Variables
op cModelData : -> IntVarId [ctor] .
op mmXmit : -> IntVarId [ctor] .
op modelData : -> IntVarId [ctor] .
op simpleMonVar : -> IntVarId [ctor] .
op pRecpt : -> IntVarId [ctor] .
op simpleRcpt : -> IntVarId [ctor] .
op complexRcpt : -> IntVarId [ctor] .
op cXmit : -> IntVarId [ctor] .
op transmissions : -> IntVarId [ctor] .
op receptions : -> IntVarId [ctor] .
op simpleCommand : -> IntVarId [ctor] .
op complexCommand : -> IntVarId [ctor] .
op inCCmd : -> IntVarId [ctor] .
op inSCmd : -> IntVarId [ctor] .
op inCmd : -> IntVarId [ctor] .
op inRWCmd : -> IntVarId [ctor] .
op aRecpt : -> IntVarId [ctor] .
```

```
 op safetyEnv : -> IntVarId [ctor] .
 op dotPos : -> IntVarId [ctor] .
 op realDotPos : -> IntVarId [ctor] .
 op wallPos : -> IntVarId [ctor] .
 op distanceFromStart : -> IntVarId [ctor] .
 op aux : -> IntVarId [ctor] .
 op inSignal : -> IntVarId [ctor] .
 op sRecpt : -> IntVarId [ctor] .

 --- -------------------------------------------------------
 --- -------------------------------------------------------

 var SI : SystemId .  var PRI : ProcessId . var TI : ThreadId .

 --- The specification

 eq system(sysSimplex) = none .

 eq SI system sysSimplex . impl =
     < SI : System | features : system(sysSimplex),
        subcomponents :
           ( idSS system sysSoftware . impl )
           ( idSRW system sysRealWorld . impl ) ,
        connections :
           ( idSS . soptCmd --> idSRW . siptCmd ) ;
           ( idSRW . soptMonVar --> idSS . siptMonVar ) > .

--- ---------------------------------------------------------
--- The Software subsystem
--- ---------------------------------------------------------
  eq system(sysSoftware) =
     (soptCmd out event data port)
     (siptMonVar in event data port) .

  eq SI system sysSoftware . impl =
     < SI : System | features : system(sysSoftware),
           subcomponents :
              ( idSC process proSimpleController . impl )
              ( idCC process proComplexController . impl )
              ( idDM process proDecisionMachine . impl )
              ( idMM process proModelMachine . impl ) ,
           connections :
               ( siptMonVar --> idMM . iptMonVar ) ;
```

```
              ( idMM . optSimpleModelData -->
                 idSC . iptSimpleModelData) ;
              ( idMM . optComplexModelData -->
                 idCC . iptComplexModelData) ;
              ( idSC . optSimpleCmd --> idDM . iptSimpleCmd ) ;
              ( idCC . optComplexCmd -->
                 idDM . iptComplexCmd ) ;
              ( idCC . optComplexCmdForSimple -->
                 idSC . iptComplexCmdForSimple) ;
              ( idDM . optCmd --> soptCmd ) > .

--- ------------------------------------------------------
--- The Modeling Machine Process
--- ------------------------------------------------------
  eq process(proModelMachine) =
     (iptMonVar in event data port)
     (optSimpleModelData out event data port)
     (optComplexModelData out event data port) .

  eq PRI process proModelMachine . impl =
       < PRI : Process | features : process(proModelMachine),
         subcomponents :
            (tidMM thread thrModelMachine . impl) ,
         connections :
            (iptMonVar --> tidMM . ipthrMonVar ) ;
            (tidMM . opthrSimpleModelData -->
               optSimpleModelData)
            (tidMM . opthrComplexModelData -->
               optComplexModelData) > .

--- --------------------------------------------------------
--- The Modeling Machine Thread
--- --------------------------------------------------------
  eq thread(thrModelMachine) =
     ports (ipthrMonVar in event data thread port)
           (opthrSimpleModelData out event data thread port)
           (opthrComplexModelData out event data thread port)
     dispatch aperiodic-dispatch .

  eq TI thread thrModelMachine . impl =
     < TI : Thread | features : ports(thread(thrModelMachine)),
         subcomponents : none,
         connections : none,
```

```
         properties : dispatch(thread(thrModelMachine)),
         status : completed,
         behavior :
            (states
               initial: s0 complete: s0
             state variables
               (simpleMonVar |-> 0)
               (mmXmit |-> 0)
             transitions

             --- Receive the monitored variable
              ( s0 -[ ipthrMonVar ? (simpleMonVar) ]-> s0
                { (mmXmit := mmXmit + 1) ;
                  (opthrSimpleModelData ! (simpleMonVar)) ;
                  (opthrComplexModelData ! (simpleMonVar)) } )
              )
        > .

--- -------------------------------------------------------------
--- The Simple Controller Process
--- -------------------------------------------------------------
  eq process(proSimpleController) =
     (optSimpleCmd out event data port)
     (iptComplexCmdForSimple in event data port)
     (iptSimpleModelData in event data port) .

  eq PRI process proSimpleController . impl =
       < PRI : Process | features :
           process(proSimpleController),
          subcomponents :
             (tidSC thread thrSimpleController . impl) ,
          connections :
             (tidSC . opthrSimpleCmd --> optSimpleCmd ) ;
             (iptSimpleModelData -->
                tidSC . ipthrSimpleModelData ) ;
             (iptComplexCmdForSimple -->
                tidSC . ipthrComplexCmdForSimple)
       > .

--- ------------------------------------------------------------
--- The Simple Controller Thread
--- ------------------------------------------------------------
  eq thread(thrSimpleController) =
```

```
    ports (opthrSimpleCmd out event data thread port)
          (ipthrSimpleModelData in event data thread port)
          (ipthrComplexCmdForSimple in event data thread port)
    dispatch periodic-dispatch(1) .

  eq TI thread thrSimpleController . impl =
    < TI : Thread | features :
         ports(thread(thrSimpleController)),
       subcomponents : none,
       connections : none,
       properties : dispatch(thread(thrSimpleController)),
       status : completed,
       behavior :
         (states
            initial: s0 complete: s0 s1
          state variables
            (transmissions |-> 0)
            (simpleCommand |-> 0)
            (modelData |-> 0)
          transitions
          --- Send the command and update
          --- the number of transmissions.
            ( s0 -[ ]-> s1 {
               (opthrSimpleCmd ! (simpleCommand) ) } ) ;
            ( s1 -[ipthrSimpleModelData ? (modelData) ]-> s0 {
               (transmissions := transmissions + 1) } ) ) > .
--- ----------------------------------------------------------------
--- The Complex Controller Process
--- ----------------------------------------------------------------

  eq process(proComplexController) =
    (optComplexCmd out event data port)
    (optComplexCmdForSimple out event data port)
    (iptComplexModelData in event data port) .

  eq PRI process proComplexController . impl =
      < PRI : Process | features :
          process(proComplexController),
        subcomponents :
                   (tidCC thread thrComplexController . impl),
              connections :
                   (tidCC . opthrComplexCmd -->
                      optComplexCmd ) ;
```

```
                  (tidCC . opthrComplexCmdForSimple -->
                     optComplexCmdForSimple ) ;
                  (iptComplexModelData -->
                     tidCC . ipthrComplexModelData)
     > .

--- ----------------------------------------------------------
--- The Complex Controller Thread
--- ----------------------------------------------------------
  eq thread(thrComplexController) =
     ports (opthrComplexCmd out event data thread port)
           (ipthrComplexModelData in event data thread port)
           (opthrComplexCmdForSimple out event data thread port)
     dispatch periodic-dispatch(1) .

  eq TI thread thrComplexController . impl =
    < TI : Thread | features :
       ports(thread(thrComplexController)),
      subcomponents : none,
      connections : none,
      properties : dispatch(thread(thrComplexController)),
      status : completed,
      behavior :
        (states
           initial: s0 complete: s0 s1
         state variables
           (cXmit |-> 0)
           (complexCommand |-> 1)
           (cModelData |-> 0)
         transitions
            (s0 -[ ]-> s1
               { (opthrComplexCmd ! (complexCommand)) ;
                 (opthrComplexCmdForSimple !
                    (complexCommand)) }) ;
            (s1 -[ ipthrComplexModelData ? (cModelData) ]-> s0
               { (cXmit := cXmit + 1) } ) ) > .

--- -----------------------------------------------------
--- The Decision Machine Process
--- -----------------------------------------------------
  eq process(proDecisionMachine) =
     (iptSimpleCmd in event data port)
     (iptComplexCmd in event data port)
```

```
     (optCmd out event data port) .

  eq PRI process proDecisionMachine . impl =
       < PRI : Process | features : process(proDecisionMachine),
         subcomponents :
                    (tidDM thread thrDecisionMachine . impl),
               connections :
                    (iptSimpleCmd --> tidDM . ipthrSimpleCmd ) ;
                    (iptComplexCmd -->
                       tidDM . ipthrComplexCmd ) ;
                    (tidDM . opthrCmd --> optCmd ) > .

--- -----------------------------------------------------
--- The Decision Machine Thread
--- -----------------------------------------------------
  eq thread(thrDecisionMachine) =
     ports
          (ipthrSimpleCmd in event data thread port)
          (ipthrComplexCmd in event data thread port)
          (opthrCmd out event data thread port)
     dispatch aperiodic-dispatch .

  eq TI thread thrDecisionMachine . impl =
     < TI : Thread | features :
          ports(thread(thrDecisionMachine)),
        subcomponents : none,
        connections : none,
        properties : dispatch(thread(thrDecisionMachine)),
        status : completed,
behavior :
                   (states
                        initial: sReceive complete: sReceive
    state variables
(aux |-> 0)
     (inSCmd |-> 0)
(inCCmd |-> 0)
(simpleRcpt |-> 0)
    (complexRcpt |-> 0)
(distanceFromStart |-> 0)
(safetyEnv |-> 2)
(dotPos |-> 0)
(wallPos |-> 5)
                    transitions
```

```
 --- Consume any incoming complex command.  Conduct the safety
 --- check on the incoming complex command.
 --- Assess how far from a starting point of 0 the dot is,
 --- plus a safety envelope.
   (sReceive -[ ipthrComplexCmd ? (inCCmd) ]-> sReceiveSimple
     { (complexRcpt := complexRcpt + 1) ;
       (distanceFromStart := dotPos + inCCmd + safetyEnv)
     } ) ;
 --- Consume any incoming simple command
   (sReceiveSimple -[ ipthrSimpleCmd ? (inSCmd) ]-> s2
     { (simpleRcpt := simpleRcpt + 1) } ) ;
 --- Check if the complex command is safe and
 --- send it if it is safe.
   (s2 -[ on (distanceFromStart < wallPos) ]-> s4
     { (dotPos := dotPos + inCCmd) ;
       (opthrCmd ! (inCCmd)) } ) ;

 --- Double-check that the plant is still safe.
   (s4 -[ on (dotPos < wallPos) ]-> sReceive { aux := 0 } ) ;

 --- If the plant is no longer safe, then the system has failed.
   (s4 -[ on (dotPos >= wallPos) ]-> fail {aux := 0 } );

 --- If the complex command is not safe, send the simple command
   (s2 -[ on (distanceFromStart >= wallPos) ]-> s3
      { (dotPos := dotPos - inSCmd) ;
        (opthrCmd ! (inSCmd))
      } ) ;

 --- Double-check that the plant is still safe.
   (s3 -[ on (dotPos < wallPos) ]-> sReceive { aux := 0 } ) ;

 --- If the plant is no longer safe, then the system has failed.
   (s3 -[ on (dotPos >= wallPos) ]-> fail {aux := 0 } )
) > .

 --- ---------------------------------------------------------
 --- ---------------------------------------------------------
 --- The Hardware subsystem
 --- ---------------------------------------------------------
   eq system(sysRealWorld) =
(siptCmd in event data port)
(soptMonVar out event data port) .
```

```
   eq SI system sysRealWorld . impl =
       < SI : System | features : system(sysRealWorld) ,
           subcomponents :
( idP process proPlant . impl )
( idS process proSensor . impl )
( idA process proActuator . impl ) ,
           connections :
                ( siptCmd --> idA . iptRWCmd ) ;
( idA . optActuation --> idP . iptCmd ) ;
( idP . optPhysPhenom --> idS . iptPhysPhenom ) ;
( idS . optMonVar --> soptMonVar ) > .

 --- -----------------------------------------------------------
 --- The Sensor Process
 --- -----------------------------------------------------------
   eq process(proSensor) =
  (iptPhysPhenom in event data port)
        (optMonVar out event data port) .

   eq PRI process proSensor . impl =
        < PRI : Process | features : process(proSensor),
          subcomponents :
                    (tidS thread thrSensor . impl),
                connections :
     (iptPhysPhenom --> tidS . ipthrPhysPhenom) ;
     (tidS . opthrMonVar --> optMonVar ) > .
 --- -----------------------------------------------------------
 --- The Sensor Thread
 --- -----------------------------------------------------------
   eq thread(thrSensor) =
ports
           (ipthrPhysPhenom in event data thread port)
     (opthrMonVar out event data thread port)
dispatch aperiodic-dispatch .

     eq TI thread thrSensor . impl =
< TI : Thread | features : ports(thread(thrSensor)),
subcomponents : none,
connections : none,
properties : dispatch(thread(thrSensor)),
status : completed,
        behavior :
```

```
                    (states
                         initial: sReceive complete: sReceive
     state variables
      (inSignal |-> 0)
 (sRecpt |-> 0)
                     transitions
 --- Detect physical phenomenon in the plant.
   (sReceive -[ ipthrPhysPhenom ? (inSignal) ]-> sReceive
      { (sRecpt := sRecpt + 1) ;
        (opthrMonVar ! (inSignal)) } ) ) > .

 --- -----------------------------------------------------------
 --- The Actuator Process
 --- -----------------------------------------------------------
   eq process(proActuator) =
  (iptRWCmd in event data port)
(optActuation out event data port) .

   eq PRI process proActuator . impl =
        < PRI : Process | features : process(proActuator) ,
          subcomponents :
                     (tidA thread thrActuator . impl) ,
               connections :
     (iptRWCmd --> tidA . ipthrRWCmd ) ;
     (tidA . opthrActuation --> optActuation) > .

 --- -----------------------------------------------------------
 --- The Actuator Thread
 --- -----------------------------------------------------------
   eq thread(thrActuator) =
        ports
           (ipthrRWCmd in event data thread port)
           (opthrActuation out event data thread port)
dispatch aperiodic-dispatch .

    eq TI thread thrActuator . impl =
         < TI : Thread | features : ports(thread(thrActuator)) ,
               subcomponents : none,
               connections : none,
               properties : dispatch(thread(thrActuator)) ,
               status : completed,
               behavior :
                    (states
```

```
                        initial: sReceive complete: sReceive
     state variables
                        (inRWCmd |-> 0)
                        (aRecpt |-> 0)
                    transitions
 --- Receive an incoming command, forward it along to the plant
    (sReceive -[ ipthrRWCmd ? (inRWCmd) ]-> sReceive
       { (aRecpt := aRecpt + 1) ;
         (opthrActuation ! (inRWCmd))
       } )
     ) > .

 --- -----------------------------------------------------------
 --- The Plant Process
 --- -----------------------------------------------------------
   eq process(proPlant) =
      (iptCmd in event data port)
      (optPhysPhenom out event data port) .

   eq PRI process proPlant . impl =
        < PRI : Process | features : process(proPlant),
                subcomponents :
                     (tidP thread thrPlant . impl),
                connections :
     (iptCmd --> tidP . ipthrCmd ) ;
     (tidP . opthrPhysPhenom --> optPhysPhenom) > .

 --- --------------------------------------------------------
 --- The Plant Thread
 --- --------------------------------------------------------
   eq thread(thrPlant) =
       ports
           (ipthrCmd in event data thread port)
           (opthrPhysPhenom out event data thread port)
dispatch aperiodic-dispatch .

    eq TI thread thrPlant . impl =
      < TI : Thread | features : ports(thread(thrPlant)),
        subcomponents : none,
        connections : none,
        properties : dispatch(thread(thrPlant)),
        status : completed ,
        behavior :
```

```
            (states
                initial: sReceive complete: sReceive
             state variables
               (inCmd |-> 0)
               (realDotPos |-> 0)
               (pRecpt |-> 0)
             transitions
--- Receive an incoming command,
--- update the position of the dot, and
--- "send" the dot position to the sensor.
             (sReceive -[ ipthrCmd ? (inCmd) ]-> sReceive
              { (pRecpt := pRecpt + 1) ;
                (realDotPos := realDotPos + inCmd) ;
                (opthrPhysPhenom ! (realDotPos)) } ) ) > .

 --- --------------------------------------------------------
 --- The Whole Thing
 --- --------------------------------------------------------

   op init : -> GlobalSystem .
   eq init = {MAIN system sysSimplex . impl} .

endtom)
```

J Model-checking the simple dot example in A2M

```
(tomod MODEL-CHECK-DOT-SIMPLEX is including DOT-SIMPLEX .

op crashing : Configuration ~> Bool [frozen (1)] .

eq crashing ( < MAIN : System |
   subcomponents :
      (C1:Configuration
      < idSRW : System |
         subcomponents :
         (C2:Configuration
         < idP : Process |
            subcomponents :
            (C3:Configuration
            < tidP : Thread |
               behavior :
                  states
                     current: L:Location
                     complete: LS:LocationSet
                     others: LS1:LocationSet
                  state variables
                     VAL:Valuation
                  transitions TS:TransitionSet
      >)
           >)
        >)
     >) = L:Location == crash .

endtom)

(tsearch [1] init =>* {C:Configuration}
   such that crashing(C:Configuration) with no time limit .)
```

References

[1] IT Convergence Lab. In *http://decision.csl.uiuc.edu/t̃estbed*.

[2] Robert Allen, Steve Vestal, Dennis Cornhill, and Bruce Lewis. Using an architecture description language for quantitative analysis of real-time systems. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 203–210, New York, NY, USA, 2002. ACM.

[3] Neal Altman, Chuck Weinstock, Lui Sha, and Danbing Seto. Simplex in a hostile communications environment: The coordinated prototype. *Software Engineering Institute Technical Report CMU/SEI-99-TR-016*, 1999.

[4] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. In *FOCS '97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, page 100, Washington, DC, USA, 1997. IEEE Computer Society.

[5] George S. Avrunin, Lori A. Clarke, Elizabeth A. Henneman, and Leon J. Osterweil. Complex medical processes as context for embedded systems. *SIGBED Rev.*, 3(4):9–14, 2006.

[6] G. Baliga, S. Graham, L. Sha, and P. R. Kumar. Etherware: Domainware for wireless control networks. In *IEEE Seventh International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 155–162, 2004.

[7] G. B. Baliga. *A Middleware Framework for Networked Control Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

[8] Y. Bar-Shalom. *Tracking and data association*. Academic Press Professional, Inc., San Diego, CA, USA, 1987.

[9] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice–2nd Ed.* Addison-Wesley Publishing Company, Boston, Massachusetts, 2003.

[10] W. L. Brogan. *Modern Control Theory*. Prentice Hall, 3 edition, 1991.

[11] D. Brugali and E. Fayad. Distributed computing in robotics and automation. *IEEE Transactions on Robotics and Automation*, 18(4):409–415, August 2002.

[12] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An Open-Source Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[13] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[14] Tanya L. Crenshaw, Elsa Gunter, Craig L. Robinson, Lui Sha, and P. R. Kumar. The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical systems. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, 2007.

[15] Tanya L. Crenshaw, C. L. Robinson, Hui Ding, P. R. Kumar, and Lui Sha. A pattern for adaptive behavior in safety-critical, real-time middleware. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, Rio de Janeiro, Brazil, 2006.

[16] Joseph K. Cross and Douglas C. Schmidt. Quality connector: An architectural pattern to enhance qos and alleviate dependencies in distributed real-time and embedded middleware. *Design Patterns for Distributed and Real-Time Systems, Christopher Gill and Lisa DiPippo (Eds)*, 2006.

[17] J. Czyz, B. Ristic, and B. Macq. A particle filter for joint detection and tracking of multiple objects in color video sequences. In *2005 8th International Conference on Information Fusion*, volume 1, July 2005.

[18] H. Ding and L. Sha. Dependency algebra: A tool for designing robust real-time systems. In *26th IEEE Real-Time Systems Symposium*. IEEE Computer Society, 2005.

[19] Bruce Douglass. *Real-Time Design Patterns*. Addison-Wesley Publishing Company, Boston, Massachusetts, 2003.

[20] Matthew Dwyer. Software model checking: the bandera approach. In *FMOODS '02: Proceedings of the IFIP TC6/WG6.1 Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems V*, pages 3–4, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[21] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *FMSP '98: Proceedings of the second workshop on Formal methods in software practice*, pages 7–15, New York, NY, USA, 1998. ACM.

[22] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[23] Stuart Faulk, John Brackett, Paul Ward, and James Kirby, Jr. The core method for real-time requirements. *IEEE Software*, 9(5), September 1992.

[24] P. Feiler, B. Lewis, and Steve Vestal. The SAE architecture analysis and design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *Workshop on Model-Driven Embedded Systems co-located with the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003.

[25] Peter Feiler. Modeling of system families. *Software Engineering Institute Technical Report CMU/SEI-2007-TN-047*, July 2007.

[26] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL): An introduction. *Software Engineering Institute Technical Report CMU/SEI-2006-TN-011*, 2006.

[27] R. Fielding and R. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, 2002.

[28] Mike Gagliardi, Theodore Marz, Neal Altman, and John Walker. Simplex architecture performance and cost. *Software Engineering Institute Technical Report CMU/SEI-2000-TR-006*, 2000.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[31] S. Graham and P. R. Kumar. The convergence of control, communication, and computation. *Proceedings of PWC 2003: Personal Wireless Communications. Lecture Notes in Computer Science*, 2775, 2003.

[32] Mark Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*. Wiley, 1998.

[33] Ellis F. Hitt and Dennis Mulcare. Fault-tolerant avionics. *The Avionics Handbook*, 2001.

[34] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[35] M. Isard and J. Maccormick. Bramble: a bayesian multiple-blob tracker. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, volume 2, pages 34–41 vol.2, 2001.

[36] ITU-TS. Message sequence charts (MSC). In *ITU-TS Recommendation Z.120*, 1996.

[37] Sheena S. Iyengar and Mark R. Lepper. When choice is demotivating: Can one desire too much of a good thing? *Journal of Personality and Social Psychology*, 79(6), December 2000.

[38] Michael Jackson. *Problem Frames: Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[39] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems*, 5(2), May 1987.

[40] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 372–381, New York, NY, USA, 2005. ACM.

[41] Marta Kwiatkowska. Quantitative verification: models, techniques and tools. In *ESEC-FSE companion '07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 449–458, New York, NY, USA, 2007. ACM.

[42] Kihwal Lee and Lui Sha. A dependable online testing and upgrade architecture for real-time embedded systems. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2005.

[43] David L. Lempia and Steven P. Miller. Requirements engineering management handbook. *U.S. Department of Transportation. Federal Aviation Administration.*, December 2006.

[44] N. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[45] J. Li and P. Feiler. Impact analysis in real-time control systems. In *15th IEEE International Conference on Software Maintenance*, 1999.

[46] Peihua Li and Hajing Wang. Object tracking with particle filter using color information. In *Third International Conference, MIRAGE 2007*, pages 534–541, 2007.

[47] D. Liberzon. *Switching in Systems and Control.* Systems and Control: Foundations and Applications. Birkhäuser, 2003.

[48] J. W. S. Liu. *Real-Time Systems.* Prentice Hall, Inc., 2000.

[49] Joseph P. Loyall, Paul Rubel, Richard Schantz, Michael Atighetchi, and John Zinky. Emerging patterns in adaptive, distributed real-time, embedded middleware. In *Proceedings of the 9th Conference on Pattern Language of Programs*, Monticello, IL, 2002.

[50] A. M. Lyapunov. *Stability of motion.* Academic Press, New-York and London, 1966.

[51] Theodore McCombs. Maude 2.0 primer; version 1.0. *Stanford Research Institute*, August 2003.

[52] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[53] Tommi Mikkonen. Formalizing design patterns. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 115–124, Washington, DC, USA, 1998. IEEE Computer Society.

[54] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* New York, NY, 1990.

[55] Peter Csaba Ölveczky and José Meseguer. Real-Time Maude: a tool for simulating and analyzing real-time and hybrid systems. In K. Futatsugi, editor, *Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications*, volume 36. ENTCS, Elsevier, 2000.

[56] David L. Parnas and Paul C. Clements. A rational design process: how and why to fake it. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT) on Formal Methods and Software, Vol.2: Colloquium on Software Engineering (CSE)*, pages 80–100, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[57] P.Dissaux, J.P. Bodeveix, M. Filali, P. Gaufillet, and F. Vernadat. AADL behavioral annex. In *Proceedings of the DASIA 2006 Conference*, May 2006.

[58] D. B. Reid. An algorithm for tracking multiple targets. In *IEEE Transactions on Automatic Control*, volume 24, pages 843–854, 1979.

[59] William C. Rounds and Hosung Song. The phi calculus: A language for distributed control of reconfigurable embedded systems. *Lecture Notes in Computer Science*, 2623:433–449, 2003.

[60] Walker Royce. Successful software management style: Steering and balance. *IEEE Software*, September/October 2005.

[61] Abhik Roychoudhury, Ankit Goel, and Bikram Sengupta. Symbolic message sequence charts. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 275–284, New York, NY, USA, 2007. ACM.

[62] B. Rubel. Patterns for generating a layered architecture. *Pattern Languages of Program Design, J. Coplien and D. Schmidt (Eds)*, pages 119–128, 1995.

[63] D. Schmidt. Middleware techniques and optimizations for real-time, embedded systems. In *Proceedings of the* 12th *International Symposium on System Synthesis*, volume 11, pages 12–16, November 1999.

[64] Bikram Sengupta and Rance Cleaveland. Triggered message sequence charts. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 167–176, New York, NY, USA, 2002. ACM.

[65] Danbing Seto, Enrique Ferriera, and Theodore Marz. Case study: Development of a baseline controller for automatic landing of an F-16 aircraft using linear matrix inequalities (LMIs). *Software Engineering Institute Technical Report CMU/SEI-99-TR-020*, 2000.

[66] Lui Sha, José Meseguer, and Jennifer Hou. Nsf grant award csr ehs: Formal model based health and medical system composition, 2008.

[67] David Sharp and Wendy Roll. Pattern usage in an avionics mission processing product line. In *OOPSLA 2001 Workshop on Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems*, 2001.

[68] J. Shin, L. Guibas, and F. Zhao. A distributed algorithm for managing multi-target identities in wireless ad-hoc sensor networks, 2003.

[69] Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. Propel: an approach supporting property elucidation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 11–21, New York, NY, USA, 2002. ACM.

[70] Society of Automotive Engineers. The SAE architecture analysis and design language (AADL) standard. *SAE International Document Number AS5506*, November 2004.

[71] Software Engineering Institute. The Simplex distributed pilot study. *http:// www.sei.cmu.edu/ simplex/ demonstrations/ distributed.html*, 1999.

[72] S. Song, J. Huang, P. Kappler, R. Freimark, and T. Kozlik. Fault-tolerant ethernet middleware for ip-based process control networks. In *Proceedings. 25th Annual IEEE Conference on Local Computer Networks*, volume 11, pages 116–125, November 2000.

[73] Neelam Soundarajan and Jason O. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 666–675, Washington, DC, USA, 2004. IEEE Computer Society.

[74] Joe Sutter. *747: Creating the World's First Jumbo Jet and Other Adventures from a Life in Aviation*. Smithsonian Books, 2006.

[75] The SEI AADL Team. *An Extensible Open Source AADL Tool Environment (OSATE): Release 1.3.0*. Software Engineering Institute, June 2006.

[76] Y. Tipsuwan and M. Chow. Gain scheduler middleware: A methodology to enable existing controllers for networked control and teleoperation part i: Networked control. *IEEE Transactions on Industrial Electronics*, 51(6), 2004.

[77] Y. Tipsuwan and M. Chow. Gain scheduler middleware: A methodology to enable existing controllers for networked control and teleoperation part ii: Teleoperation. *IEEE Transactions on Industrial Electronics*, 51(6), 2004.

[78] Thomas Vergnaud, Laurent Pautet, and Fabrice Kordon. *Reliable Software Technology*, chapter Using the AADL to Describe Distributed Applications from Middleware to Software Components, pages 67–78. Springer, 2005.

[79] H. S. Witsenhausen. A counter example in stochastic optimal control. *Siam J. Control*, 6:131–147, 1968.

[80] H. S. Witsenhausen. Separation of estimation and control for discrete time systems. *Proceedings of the IEEE*, 59(11):1557–1566, November 1971.

[81] Janusz Zalewski. Real-time software design patterns. In *9th Conference on Real-Time Systems*, 2002.

Author's Biography

This fall, Tanya L. Crenshaw returns to her home state and baccalaureate alma mater, University of Portland, as an assistant professor. At the Computer Science department at the University of Illinois at Urbana-Champaign, her PhD research focused on real-time systems integration. Tanya has consistently worked to improve computer science education and directed her energy into such diverse efforts as undergraduate and graduate retention, the MergeSort Dance Troupe, and developing family-leave policies. During her six years in Illinois, Tanya hosted a variety community radio shows: from Japanese pop to country music.