The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

THE CONTROL, COMMUNICATION, AND COMPUTATION LANGUAGE
(C3L): COMPLETING THE DESIGN CYCLE IN COMPLEX DISTRIBUTED
SYSTEM DEVELOPMENT

A Dissertation in
Computer Science and Engineering

by

John Phillip Sustersic

© 2009 John Phillip Sustersic

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2009
The dissertation of John Phillip Sustersic was reviewed and approved* by the following:

Mahmut Kandemir  
Associate Professor of Computer Science and Engineering  
Dissertation Co-advisor  
Chair of Committee

Shashi Phoha  
Professor of Electrical Engineering  
Dissertation Co-advisor

Mary Jane Irwin  
Robert E. Noll Professor of Computer Science and Engineering

Robert Nickel  
Adjunct Professor of Electrical Engineering

Raj Acharya  
Professor of Computer Science and Engineering  
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School
ABSTRACT

Innovations in computing and communications have enabled autonomous electromechanical devices that change their internal states in response to real-time interactions with other devices, software agents, or smart sensors. Ad-hoc wireless networks of such devices may be engineered to perform autonomous missions like battlefield surveillance using sensor networks or multi-robot air campaigns. A critical research challenge is to harness the capabilities of these devices via ad-hoc wireless networks to invoke deliberate processes in response to operational dynamics and perform collaborative missions effectively. The Control, Communication and Computation Language (C3L) is a language designed to capture the semantics of the dynamic architectures of such distributed systems and concisely express complex, time-critical and mission-critical operations within the system. The alphabet of C3L consists of atomic events and actions that may be either controllable or uncontrollable. Atomic events and actions are combined in various ways to represent composite behaviors as event/action sequences. This programming paradigm enables heterogeneous, independently-developed devices plug-n-play compatibility in ad-hoc networked applications with either independent or coordinated strategies. By embedding the operational dynamics of the system in the higher-level semantics of the language, this framework permits multi-level fusion of distributed data in an intuitive manner.

The revised syntax and a complete operational semantic description of C3L are presented and detailed. To more fully explore the value of C3L as a tool for rapid development and testing of scalable distributed programs for autonomous devices, a
compiler has been developed for the language. The compiler exploits the implicit task-level parallelism inherent in C3L programs to automatically generate multi-threaded executable programs. Based on the Microsoft Phoenix framework for compiler development and optimization, the C3L compiler can target any architecture supported by the framework; consequently a C3L program may be compiled for a particular architecture for simulation studies and verifications, and then the same program and compiler may be used to create executable programs for the actual hardware implementation of real devices using a different architecture. The compiler design and implementation are described, and the compiler is shown to generate deadlock-free multithreaded applications for any legal C3L program. Initial benchmarking tests show that the executables generated by the C3L compiler achieve near-linear speedups in standard, multiprocessing desktop systems in ordinary operating environments when sufficient concurrency exists to offset the inter-processor messaging costs of the application.

Simulation studies are generally conducted early in the development cycle for most real systems; distributed system development of large-scale heterogeneous sets of autonomous devices operating in real-world environments imposes stringent demands on such simulations. The C3L compiler was integrated with the Visualization Tool Kit to enable native support for sophisticated presentations of either simulation or real-world behaviors. This compiler support for high-performance visualizations and the language extensions needed to exploit this functionality is presented. Finally, to illustrate the utility of the C3L language, its compiler and visualization support, an application study in the domain of the marine mine countermeasure mission is presented.
TABLE OF CONTENTS

LIST OF FIGURES .......................................................................................................................... ix

LIST OF TABLES ................................................................................................................................. xi

ACKNOWLEDGEMENTS ...................................................................................................................... xii

Chapter 1  Introduction ..................................................................................................................... 1
  Roadmap ........................................................................................................................................... 5

Chapter 2  Background .................................................................................................................... 7
  Languages for Real-time Embedded Control .................................................................................. 7
  Formal Language Descriptions .......................................................................................................... 11
    Grammar ......................................................................................................................................... 11
    Semantics ....................................................................................................................................... 11
    Operational Semantics .................................................................................................................... 13
  Concurrency in Formal Language Descriptions .............................................................................. 14
    Process Calculi ............................................................................................................................... 15
    The Actor Model ............................................................................................................................ 19
    Petri Nets ....................................................................................................................................... 20
  Lightweight Formal Methods ............................................................................................................ 21
    Type Systems .................................................................................................................................. 22
  Semantics of Real Time, Reactive Systems ....................................................................................... 23
    Temporal Logic-based Formal Methods ......................................................................................... 24
    Duration Calculus ........................................................................................................................... 25
  Compiler Design and Optimization .................................................................................................... 26
    Compilers for Parallel, Concurrent Languages .............................................................................. 28

Chapter 3  C3L Description and Syntax .......................................................................................... 30
  Introduction ...................................................................................................................................... 30
  Contributions to C3L Syntax ............................................................................................................. 32
  Chapter Roadmap ............................................................................................................................. 34
  Description of C3L ............................................................................................................................ 35
    The Plant Model .............................................................................................................................. 36
    The Controller Model ....................................................................................................................... 39
    C3L Memory Variables .................................................................................................................... 39
    C3L Sets .......................................................................................................................................... 40
    C3L Control Events .......................................................................................................................... 41
    Raise Statements ............................................................................................................................ 42
    If Statements ................................................................................................................................. 43
    Assignment Statements .................................................................................................................... 44
Chapter 4  C3L Operational Semantics and Formal Language Properties ........... 50

Introduction ........................................................................................................... 50
Contributions to C3L Semantics ........................................................................ 51
Concurrency in C3L Semantics .......................................................................... 52
Chapter Roadmap ............................................................................................ 53
C3L Operational Semantics .................................................................................. 53
Notations ............................................................................................................. 54
The Device Context ............................................................................................ 56
Operational Semantics of C3L ............................................................................. 59
Evaluation of Values and Variables .................................................................... 59
Evaluation of Expressions .................................................................................... 61
Unary Operations ................................................................................................. 61
Non-relational Binary Operations ....................................................................... 61
Relational Operations .......................................................................................... 63
Binary Logical Operations ................................................................................... 66
Equivalence of Expressions ............................................................................... 66
Constant Expressions .......................................................................................... 67
Execution of Statements ....................................................................................... 68
Assignment Statements ....................................................................................... 68
Selection Statements ............................................................................................ 69
Repetition Statements ......................................................................................... 69
Local Variable Declarations ............................................................................... 72
Calling Functions ................................................................................................. 73
Invocation of Controllable Plant Events ............................................................ 74
Semantics of Events ............................................................................................. 75
Occurrence of Control Events ............................................................................. 76
Raise Statements ................................................................................................. 77
Equivalence of Statements in C3L ....................................................................... 79
Occurrence of Uncontrollable Plant Events ....................................................... 80
Semantics of the Event Queue ............................................................................. 81
Summary of Formal Semantics ............................................................................ 83
Formal Properties of C3L ..................................................................................... 84
Prefix-closed Controllability in C3L ................................................................. 84
Event Fairness ...................................................................................................... 86
Summary ............................................................................................................... 93

Chapter 5  A Parallelizing Compiler for C3L ....................................................... 94

Introduction ......................................................................................................... 94
Chapter Roadmap ............................................................................................... 96
The Compiler Architecture ................................................................. 96
The Compiler Front End ...................................................................... 98
  The Scanner ................................................................................... 98
  The Parser .................................................................................... 99
  The Compiler ............................................................................... 99
  The Runtime Library Interface ..................................................... 101
The Compiler Back End ...................................................................... 103
  The C3L Runtime Library .............................................................. 103
Linking C3L Programs ....................................................................... 105
Debugging Support for C3L Programs .............................................. 105
Compiler Implementation Details ..................................................... 105
The C3L Runtime Environment ........................................................ 106
Implementation of C3L Language Constructs .................................. 107
  The Structure of Controllable Event Handlers ............................... 108
  The Structure of Uncontrollable Event Handlers .......................... 109
  The Structure of Control Event Handlers ...................................... 111
  The Structure of the C3L Runtime Environment Event Processor ... 114
  The Structure of C3L Device Executors ........................................ 115
Structure of C3L Executables ............................................................ 116
  Generated ‘main’ Function for C3L Programs ............................... 117
Formal Properties of Executables Generated by the C3L Compiler ... 118
  Deadlock in Multithreaded C3L Executables ............................... 118
  Event Fairness and Starvation, and Livelock in Multithreaded C3L
    Executables ............................................................................... 120
Performance Evaluation of the Multithreaded Executables Generated by the
  C3L Compiler on Multicore Systems ............................................ 122
Summary and Research Directions ................................................... 125

Chapter 6 Visualization and Simulation Support .................................. 128
  Introduction .................................................................................. 128
  Chapter Roadmap ........................................................................ 130
  The Visualization Tool Kit (VTK) .................................................. 130
  Visualization Support in the C3L Compiler .................................... 131
    MCM – Bathymetric Data Processing ......................................... 131
    MCM Visualization Models ...................................................... 138
    C3L Language Extensions ....................................................... 138
  Simulation and Application Development Using C3L .................... 139
  Implementation Status and Future Work ....................................... 141

Chapter 7 Application Study .............................................................. 143
  Introduction and Motivation ......................................................... 143
  Chapter Roadmap ....................................................................... 143
  Application Model Development .................................................. 144
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario Description</td>
<td>144</td>
</tr>
<tr>
<td>Kinematic Modeling</td>
<td>146</td>
</tr>
<tr>
<td>Acceleration Model</td>
<td>146</td>
</tr>
<tr>
<td>Depth Model</td>
<td>148</td>
</tr>
<tr>
<td>Heading Model</td>
<td>150</td>
</tr>
<tr>
<td>The Position Model</td>
<td>151</td>
</tr>
<tr>
<td>C3L Implementation of AUV Kinematic Models</td>
<td>151</td>
</tr>
<tr>
<td>AUV Plant Implementation of Kinematic Models and Control Systems</td>
<td>153</td>
</tr>
<tr>
<td>AUV State Space for Kinematic Models and Control Systems</td>
<td>153</td>
</tr>
<tr>
<td>AUV Controllable Event Handlers for Kinematic Models and Control Systems</td>
<td>154</td>
</tr>
<tr>
<td>Performance of C3L Kinematic Models and Control Systems</td>
<td>155</td>
</tr>
<tr>
<td>High-Level Scanning and Identification Modeling</td>
<td>162</td>
</tr>
<tr>
<td>Scanning AUV Mission Controller</td>
<td>163</td>
</tr>
<tr>
<td>Identification AUV Mission Controller</td>
<td>166</td>
</tr>
<tr>
<td>Performance Evaluation of MCM Scenario</td>
<td>167</td>
</tr>
<tr>
<td>Summary and Future Work</td>
<td>170</td>
</tr>
<tr>
<td>Chapter 8 Conclusions</td>
<td>174</td>
</tr>
<tr>
<td>Research Directions</td>
<td>175</td>
</tr>
<tr>
<td>Bibliography</td>
<td>179</td>
</tr>
<tr>
<td>Appendix A C3L Application Programs</td>
<td>190</td>
</tr>
<tr>
<td>TestProg4.c3l – ‘Computation-bound’ Benchmarking Compiler Test</td>
<td>190</td>
</tr>
<tr>
<td>TestProg5.c3l – ‘Communication-bound, higher messaging complexity’</td>
<td>192</td>
</tr>
<tr>
<td>Benchmarking Compiler Test</td>
<td>192</td>
</tr>
<tr>
<td>TestProg6.c3l – ‘Communication-bound, lower messaging complexity’</td>
<td>195</td>
</tr>
<tr>
<td>Benchmarking Compiler Test</td>
<td>195</td>
</tr>
<tr>
<td>app1.c3l – The MCM Scenario Simulation</td>
<td>197</td>
</tr>
<tr>
<td>Appendix B C3L Compiler Code Documentation</td>
<td>214</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 3-1: C3L Conceptual Model ............................................................................. 36
Figure 3-2: C3L Device Specification ......................................................................... 37
Figure 3-3: Example C3L Plant and Controller Models ............................................. 39
Figure 5-1: Compiler Architecture Overview .............................................................. 97
Figure 5-2: Conceptual Model of the C3L Runtime Environment ......................... 106
Figure 5-3: A template C3L controllable event handler definition with specified precondition and the corresponding control flow graph generated by the compiler. ............................................................................................................... 109
Figure 5-4: A template C3L uncontrollable event handler definition with specified precondition and the corresponding control flow graph generated by the compiler. ......................................................................................................... 111
Figure 5-5: Control flow graph of the algorithm handling events raised to a device, describing both the admission of events to the event queue of the device as well as the specific processing of uncontrollable events. ..................... 112
Figure 5-6: A template C3L control event handler definition with specified authorization and the corresponding control flow graph generated by the compiler. ............................................................................................................... 113
Figure 5-7: The C3L runtime environment event processing algorithm .................. 114
Figure 5-8: The Control Flow Graph of the C3L Device Executor, which takes input events from the queue and executes the appropriate handler. ................. 115
Figure 5-9: The structure of the ‘main’ function generated for all C3L programs ...... 116
Figure 5-10: Speedup of Performance Benchmarks ................................................... 126
Figure 6-1: Screenshot of a visualization created representing two collaborating AUVs in an MCM mission. The bathymetry visualization was generated directly from NOAA depth sounding data of an area in the Chesapeake Bay off the coast of Virginia ....................................................................................... 129
Figure 6-2: A small sample of ungridded data points in plan view from NOAA’s NGDC Digital Sounding Data. The data are soundings from the Chesapeake Bay off the coast of Virginia and displayed here in NOAA’s GEODAS Hydro-Plot application ........................................................................................................ 132
Figure 6-3: Color coding of bathymetric and hypsometric depth ranges defined for use specifically in visualizations for MCM applications. ............................... 134

Figure 6-4: A visualization of an area in the Adriatic sea using the MCM color coding. This data is from the GEBCO one minute gridded data set and shows depth to a MSL of 7 meters. ................................................................................. 136

Figure 6-5: The affects of quadric decimation.......................................................... 137

Figure 7-1: MCM Region, Segmentation and AUV Assignments. ......................... 145

Figure 7-2: Acceleration and Deceleration vs. Speed for the AUVs. ................. 149

Figure 7-3: C3L Simulation Time Keeper Device Definition – a device that raises events to all devices in prescribed intervals. ......................................................... 152

Figure 7-4: C3L Implementation of State Variables used in the AUV’s Kinematic Models and Control Systems................................................................. 154

Figure 7-5: C3L Implementation of Controllable Event Handlers used in the AUV’s Kinematic Models and Control Systems.................................................. 158

Figure 7-6: Acceleration and Deceleration Model and Control System Test Results – Forward speed in knots versus time in seconds. ................................. 159

Figure 7-7: Acceleration and Turn Performance Test Results - Position of the AUV in the X-Y plane. Data markers show the position of the AUV each second. Units are nm................................................................. 160

Figure 7-8: The C3L Implementation of the Controller to Execute the Multiple Turn Test........................................................................................................... 161

Figure 7-9: Plot of AUV Position in the Multiple Turn Test. Units are nm ............ 162

Figure 7-10: The First Portion of the C3L Implementation of AUV1’s High-level Mission Controller (for the complete implementation, see Appendix A) .... 164

Figure 7-11: The C3L Implementation of the scan Control Event – the event handler that simulates detection of MLCs.............................................................. 166

Figure 7-12: State Machine for the Identification AUV’s Mission Controller........ 168

Figure 7-13: C3L Implementation of the State Machine for the Identification AUV’s Mission Controller. ............................................................................. 172

Figure 7-14: AUV Trajectories and Locations of MLCs in MCM Mission Simulation................................................................................................................. 173
LIST OF TABLES

Table 2.1: Classes of Formal Semantics of Programming Languages ......................... 12
Table 3-1: C3L Syntax – part 1...................................................................................... 46
Table 3-2: C3L Syntax – part 2...................................................................................... 47
Table 4-1: Symbols Used in C3L Semantics .................................................................. 56
Table 4-2: Metavariables Used in C3L Semantics ......................................................... 56
Table 4-3: Function Definitions Used in C3L Semantics ............................................... 57
Table 5-1: C3L Compiler’s Phase List in Order of Application................................. 100
Table 5-2: Standard, Implicit C3L Runtime Library Functions and Definitions .......... 102
Table 5-3: Standard, Implicit C3L Runtime Library Functions and Definitions for Random Value Generation .......................................................... 103
Table 6-1: An example control event designed to communicate tracking information to collaborating devices ......................................................... 140
Table 7-1: Performance Characteristics of AUVs ....................................................... 144
Table 7-2: AUV Performance Characteristics Determined Experimentally............. 156
ACKNOWLEDGEMENTS

Special thanks to Mahmut Kandemir, Shashi Phoha, Mary Jane Irwin, Robert Nickel, Mendel Schmiedekamp, Matthew Davis and Doina Bein for their efforts reviewing this work and for their advice.

I would also like to thank my wife, Heather, and my daughter, Charlotte, for their long-suffering patience and understanding, without which this work would not be possible.
Chapter 1

Introduction

Computer-controlled autonomous devices are deployed in critical roles in military and civilian operations with increasing frequency, collaboratively accomplishing their mission in human time frames and often working with humans. One may view this system of autonomous, interacting mobile devices and the environment in which it operates as a complex dynamic system in which changes to state are the direct result of discrete events. In this model, higher-level abstractions like situational awareness become one or more state-space parameters, and changes to a device’s situation may happen as the result of either the occurrence of events internal to that device or events external from it. This approach enables these higher-level abstractions like situational awareness to be managed as easily as lower-level abstractions like a device’s battery state.

Reactive control of devices involves specifying controlled actions when certain events or conditions are detected. In effect, the communication of the knowledge that an event occurs is necessary and sufficient to enable the appropriate control response. Simply put, control and communication share the same abstraction. This abstraction permits an intuitive description of required specific device behavior under prescribed circumstances in complex distributed systems. The Control, Communication, and Computation Language (C3L) [Sustersic2009a, Sustersic2009b, Sustersic2008, Sustersic2009c, Schmiedekamp2006, Skarbez2004, Phoha1999] creates a language-level abstraction of this control methodology for distributed system environments, freeing the
programmer from details regarding how to communicate events between devices and allowing him or her to focus exclusively on the behavioral description of the device. The occurrence of events is treated exactly the same regardless if the event is an internal device event or a message from another device, further abstracting complexity of the problem into the language domain. This programming paradigm permits intuitive yet powerful and computationally expressive distributed applications targeted for ad-hoc networked environments to be developed quickly and effectively by small groups of programmers without special training – in effect enabling truly mainstream distributed programming.

Technological advances have enabled large systems of networked, heterogeneous, autonomous devices. In the context of autonomous vehicles there is a clear need for a programming language that captures the dynamics of the system. As an example, consider unmanned underwater vehicles (UUVs). There has been a growing interest in incorporating UUVs into various military and civilian applications. The main motivation for using UUVs in such applications may be found in their clandestine, low-risk nature. Specifically, UUVs could be easily deployed in an area of interest such as offshore fleet operating areas, or littoral penetration areas to perform Intelligence, Surveillance and Reconnaissance (ISR), tactical hydrographic and oceanographic data collection. Mission profiles in these areas of operations might include search and survey for Mine Counter-Measures (MCM), wrecks, lost objects, cables or pipelines (e.g. oil industrial applications). Similar applications in the domain of Unmanned Aerial Vehicles (UAVs), Unmanned Surface Vehicles (USVs), and collaborations between heterogeneous devices
including UUVs, UAVs, USVs as well as human operatives and other assets are intriguing as well.

In the domain of UUVs, consider the problem of mine detection [Eberbach1992, Phoha2000, Phoha2002]. A set of UUVs is tasked with scanning a region with various types of sensors, each with a particular sensitivity and accuracy in detecting mines. In the absence of detections or other events, the objective of this distributed system is to scan the greatest area possible in the shortest time; if a mine-like detection is made by one of the UUVs, the system should respond according to the quality of the detection, its proximity to shipping lanes, the availability of additional UUV resources to investigate the detection, etc. In this example, the situation is expressed in terms of the collective state-space of detections and their locations, the state of the UUVs, and other external factors such as an approaching convoy through the region. A weak detection far from established shipping lanes might not warrant an immediate response, while strong detections by multiple UUVs directly in the path of an approaching convoy would warrant a strong response, not only by the UUVs scanning for mines but also potentially by the human operators commanding the convoy.

The path between the high-level abstractions created in modeling complex distributed systems using language-based or other methods and the decidedly low-level machine-coded implementations of those behavioral models is long and convoluted. The primary advantage of using C3L in these applications is that, by design, the C3L language was created to allow a direct implementation of those high-level abstractions. The formal properties of the language ensure that all events defined in the semantic description of the distributed system are treated fairly by guaranteeing that all are handled
in finite time [Sustersic2009, Schmiedekamp2006, Skarbez2004]. Moreover, these formal properties guarantee that any device in C3L is prefix-closed controllable. The programmatic description of the distributed system may then be analyzed on a regular desktop environment as a compiled executable using the simulation and visualization support built into the compiler. The next step in the development cycle involves implementing the system on real hardware, then eventually in-situ in real applications. Since its compiler can target any architecture supported by the Phoenix framework, this last step is easily implemented using C3L. The overall objective is to provide a reliable and efficient toolset through which complex distributed system development can be accomplished effectively.

The operational model upon which C3L is based have been described in the literature [Phoha1999], as has a previous version of C3L [Schmiedekamp2006, Skarbez2004]. This version did present both a syntax and formal semantics. However, several significant shortcomings of those formalities were identified. The previous version of C3L syntax was cumbersome and had known shortcomings for which there were readily available solutions. The previous version of C3L semantics were reasonable with respect to defining simple operations that didn’t affect other devices, but much of the complexity and the formality of inter-device event distribution were defined in something of a ‘black box’ manner. Additionally, the original semantics were large-step semantics, and objective, peer reviewers were uncomfortable with them, in one case going so far as to speculate that the results derived from the semantics might have been artifacts of the large-step notation in themselves. Clearly, the larger programming
community was unwilling to accept the previous versions of C3L as a formal language with useful properties.

This thesis begins by formally describing the C3L language in a newly revised and improved syntax and in a complete operational semantic description. A modular and highly extensible, parallelizing compiler for the language is presented. Built on the Microsoft Phoenix framework for compiler design and optimization [Microsoft2008], the compiler allows C3L programs to be generated for any platform supported by the framework and simple integration with both existing compiler optimizations as well as any custom optimizations developed specifically for C3L. The multithreaded executables generated by the compiler are shown to be deadlock-free for any legal C3L program, and benchmarking applications demonstrate that these executables show near-linear speedups\(^1\) in off-the-shelf multiprocessing desktop environments when sufficient computational concurrency exists to offset communication overhead. Extensions to the compiler are described that allow high-performance 4D visualizations in simulation studies that can be generalized to visualizations in any C3L program. Finally, an application study demonstrates the practical utility of the C3L language and its compiler.

**Roadmap**

A background of related work is presented in chapter two. A description of C3L and the revised formal syntax are presented in chapter three, and then a complete

---

\(^1\) Speedup is defined as the ratio of time a task takes to execute on a single processor to the time of the same task on multiple processors. The linear speedup \(S\) for \(N\) processors is defined as \(S = N\).
operational semantic for the C3L language are presented in chapter four. A parallelizing compiler is described in chapter five, then extensions to the compiler to support high-performance visualizations of simulations is discussed in chapter six. An application study is discussed in chapter seven. Finally, this thesis is concluded in chapter eight.
Chapter 2

Background

The C3L programming paradigm enables intuitive yet powerful and computationally expressive distributed system development in ad-hoc networked environments. This language may be classified as an asynchronous, reactive programming language and the intention for the language is to establish a semantic foundation from which formal verification of program behaviors is possible. To provide a reasonable background for this work in the literature, related work in the real-time embedded control languages is discussed. The discussion continues with related work in the field of formal semantics of programming languages. A brief background in compiler design is presented, and finally related work in simulation and scientific visualizations is discussed.

Languages for Real-time Embedded Control

In the domain of real-time embedded control, the synchronous family of reactive languages has been firmly established through successful commercialization and mission-critical industrial application as the technology of choice for modeling, specifying, implementing and validating control applications [Benveniste2003]. Synchronous languages like Esterel [Boussinot1991], Lustre [Halbwachs1991], and Signal [LeGernic1991] have been adopted by aerospace manufacturers and the nuclear power
industry in safety-critical applications. Built on a common mathematical framework, these languages provide deterministic concurrency and synchrony. The success of these languages may be attributed to the sound mathematical models on which they are constructed, enabling formal verification of the properties of the language and real-time embedded control systems implemented with the language – a highly desirable property in safety-critical control systems.

While widely accepted in real-time embedded control, synchronous languages exhibit significant, fundamental shortcomings in scaling towards large-scale heterogeneous control of autonomous devices. The root of these shortcomings is embedded in the very synchrony assumptions that make these languages effective in embedded control applications. Consider for example the Esterel language and the two assumptions upon which the language is built – actions are atomic and take negligible time, and signals propagate infinitely fast (i.e., also take no time) and may occur simultaneously. Clearly the later assumption falters if signaling is required over a wireless ad-hoc network link, and the former assumption fails if an atomic action requires signaling over a similar channel.

In contrast to synchronous control, asynchronous reactive languages such as Electre [Cassez1995] are based on a different set of assumptions. First, actions in the system take finite time, therefore may consequently be preempted and resumed. Second, signals may not occur simultaneously since raising a signal is a discrete event in a continuous (real) time domain. While these assumptions are more realistic for control of a scalable, distributed system, there is a distinct and significant trade-off: formal verification of behaviors is difficult under asynchronous language assumptions.
Researchers in [Richard1996] attempt to blend the properties of synchronous and asynchronous languages to preserve desirable characteristics of both languages in a unified, ambisynchronous reactive language. This approach involved the synchronous composition of transition systems using Esterel and Electre. This effort was an early step towards formal verification of hybrid systems – systems with both discrete and continuous state changes.

The Esterel-C language is a hybrid language blending the ANSI-C general-purpose programming language with Esterel components for specifying reactivity [Lavagno1999]. Formal analysis of hybrid systems has been shown for certain classes of hybrid systems through discrete abstraction of the continuous state space [Alur2000]. Temporal logic-based automatic verification of embedded systems of hybrid automata-communicating machines with finite control and real-valued variables has been implemented [Alur1996]. With the formalism of a mathematical foundation, hybrid systems have provided a foundation to a wide range of significant applications from air-traffic management systems to robotics.

Hierarchical discrete event control models interacting devices as finite automata with defined states, input events and output events [Peluso1996, Zhong1999]. Automata communicate by sending events to subordinate or supervisory automata. Large systems can be defined in this way, but hierarchical discrete event control is often difficult to validate. Discrete event dynamics of interacting mobile devices has been modeled as hybrid hierarchical interacting probabilistic automata [Phoha2002, Phoha1999].

Interacting hybrid systems modular design was presented in a modeling language called CHARON, allowing architectural specifications, discrete and continuous
components and a behavioral hierarchy [Alur2003]. The Hybrid Concurrent Constraint (HybridCC) language permits modeling the temporal dynamics of hybrid systems [Gupta1998]. In HybridCC, the constraints are evaluated in a regimented step-by-step fashion and may either be algebraic or differential equation constraints.

Agent-oriented design paradigms have also been incorporated into networked system development. Dependability in these systems is founded not upon mathematical formalism but in the rigor of strict protocols governing agent interaction and includes FIPA [FIPA2002] and KQML [Chalupsky1992] protocols. Structured agent languages include a class of event based languages such as Structured Circuit Semantics [Lee1997] and $\Psi$–calculus [Kinney2003]. These languages specify explicit procedures to be executed when a device recognizes a situation, often through detecting events.

In the field of autonomous underwater vehicles (AUVs), Chappell et. al. [Chappell2005, Duarte2005] express AUV missions as statement sequences in a mission file and make up a Common Control Language (CCL) that is compiled and executed by the AUV’s control system. Stokey et. al. [Stokey2005] propose using a CCL containing commands for an AUV and data messages for sensors aboard the AUV to control one or more vehicles operating in an area. Davis and Brutzman [Davis2005] propose using a CCL that not only contains task level commands for the vehicles, but also includes meta-
commands used for vehicle specific information

C3L was developed in response to the operational needs of networking cooperating, autonomous UUVs. The original language was conceived as a generic behavior message-passing language [SJ1998] as part of the Autonomous Ocean Sampling Network (AOSN) project [Phoha2002]. Over time, this language was generalized into a
class of common control languages. One of these was based on process algebras [Eberbach2003] and is representative of the script-based structure of these languages.

**Formal Language Descriptions**

**Grammar**

One of the earliest and most widely used formal expressions through which a programming language may be described is the Backus-Naur Form (BNF) [Naur1963, Chomsky1956, Chomsky1957, Knuth1964], a metasyntax used to express context-free grammars. BNF notation defined the syntax of a programming language using lexical rules and syntactic rules to specify a set of derivation rules that precisely describe the formal language – the set of strings (symbols) over the alphabet of the language [Harrison1978]. While this formal grammar describes all possible sequences of symbols in a language, it does not describe their semantics – i.e. what the particular generation of symbols means.

**Semantics**

The meaning of statements in a programming language is defined through its formal semantics [Winskel1996, Gunter1992, Krishnamurthy2003, Reynolds1998, Hennessy1990]; though there are many approaches to formally describing the meaning of languages, all known approaches may be classified into one of the three classifications or some hybrid thereof outlined in Table 2.1. Instead of competing with each other,
these three classes of semantic systems are often used together symbiotically, and variations of these semantic systems are found in the choice of supporting mathematical formalism underlying the semantic system.

<table>
<thead>
<tr>
<th>Table 2.1: Classes of Formal Semantics of Programming Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>Operational Semantics</strong> – In operational semantics, the execution of the language is described directly by giving meaning to phrases of the language through the transitions they induce on the states of the machine. Operational semantics may include the description of an abstract machine and show operation on that machine, thereby define the meaning of language phrases on any machine independent of architecture.</td>
</tr>
<tr>
<td>• <strong>Denotational Semantics</strong> – In denotational semantics, each phrase in the language is translated into a phrase in another language (a denotation). Usually, the denotations are a mathematical formalism instead of another computer language and use more abstract mathematical concepts of complete partial orders, continuous functions, and least fixed points.</td>
</tr>
<tr>
<td>• <strong>Axiomatic Semantics</strong> – In axiomatic semantics, each phrase in the language is given meaning by describing the logical axioms that apply to them. There is no distinction between the phrase’s meaning and the axioms that describe it. Since axiomatic semantics try to fix the meaning of a program phrase by giving proof rules for it within a program context, these semantic systems emphasize proof of correctness from the beginning. The Hoare logic [Hoare1985] is the canonical example of axiomatic semantics.</td>
</tr>
</tbody>
</table>
Operational Semantics

While a common goal of semantic descriptions of programming languages is to provide an aesthetic mathematical foundation for the language, it is still important to maintain contact with the realities of the eventual implementations of the languages:

It is all very well to aim for a more ‘abstract’ and a ‘cleaner’ approach to semantics, but if the plan is to be any good the operational aspects cannot be completely ignored. [Scott1970]

Consequently, the operational semantics of programming languages are frequently the starting point for the development of formal semantics, particularly since denotational semantics and axiomatic semantics often work in collaboration with those operational semantics. It should not be surprising, therefore, that there is much discussion of operational semantics in the literature – a very brief overview of which is provided here.

The operational semantic class of formal semantics provides meaning to the phrases of a programming language by interpreting those phrases as sequences of computational steps; these sequences are then the meaning of the phrase. One of the first implementations of operational semantics was the λ-calculus, developed as part of an investigation into the foundations of mathematics centering on function definition, function application, and function recursion [Hankin1994, Krivene1993, Revesz1988, Barendregt1984, Church1941]. The λ-calculus is conceptually an idealized programming language capable of describing any algorithm and is the mathematical model underlying the functional programming languages. In functional programming, functions are stateless and exclusively return data given unalterable input data. Consequently,
functional programs contain no ‘side effects’ – i.e. changes in state that may affect the behavior of other functions. Building on the \( \lambda \)-calculus, modern functional programming languages include Erlang [Armstrong2007], F# [Petricek2008, Microsoft2008a], Haskel [Haskell2008], Lisp [Winston1997], ML [Milner1997], Scheme [Dybvig2003], and Scala [Odersky2006].

Structural Operational Semantics are an important subclass of operational semantics developed as a means to describe the behavior of a program in terms of the behavior of its parts [Winskel1996, Plotkin2004, Plotkin2004a]. These semantics are called structural because the semantics define meaning according to the structure of the program as directed by the syntax of the language and provide an inductive view of the operational semantics. Structural operational semantic definitions include one or more transition relations and a set of inference rules defining valid transitions as the composition of the transitions of its pieces. These definitions permit formal analysis of relations between programs, some of which are particularly useful in the context of concurrency theory. Structural operational semantics have become a de facto standard for defining operational semantics.

**Concurrency in Formal Language Descriptions**

While the formal semantic descriptions of programming languages discussed thus far have properties that may be useful in analyses of concurrent programs, in themselves they generally describe sequential operations as modeled by the single instruction, single data (SISD) classification of Flynn’s Taxonomy [Flynn1972]. In the \( \lambda \)-calculus,
evaluation through β-reduction can be carried out in any order or even in parallel, but there is no explicit support for parallelism.

Concurrency theory mathematically formalizes the study of systems consisting of independent, interacting components, each of which evolves simultaneously with the others [Bowman2006]. A number of classes of formal systems with varying degrees of mathematical rigor for reasoning about general concurrent systems have been described in the literature; of these the process calculi, actor model, and Petri net models will be discussed in more detail.

**Process Calculi**

Process calculi have been used to formally model concurrent systems through the description of interactions, communications and synchronizations between independent processes to permit formal reasoning about parallel and distributed systems [Hennessy1988, Hoare1985a, Milner1980, Pierce1997]. Process calculi use explicit channels for communications, a key feature distinguishing these calculi from other models of concurrency. Though the variation in process calculi are large and include some which specify stochastic behaviors, there are three features common to all [Pierce1997]:

- Intercommunication between independent processes is represented by message-passing through explicit channels. There is no concept of shared memory between independent processes.
• Processes and systems are represented by a small set of primitives and a set of operators with which to combine those primitives.

• Process expressions are manipulated using equational reasoning using a defined set of algebraic laws for the process operators.

Though differing in form, the basic operators in process calculi allow for the following [Baeten2005]:

• Parallel composition of processes. This is the key primitive distinguishing the process calculi from sequential computations.

• Specification of channels to use when sending and receiving data.

• Sequentialization of interactions between processes.

• Hiding of interaction points.

• Process replication or recursion – i.e. operations that allow for finite descriptions of infinite behavior.

One of the first process calculi proposed for the study of interactive systems was CCS – a Calculus for Communicating Systems [Milner1989] and consisted of the following two components:

• A simple formal language describing systems as individual, interacting components.

• A semantic theory to help understand the behavioral interactions of systems described in this language.
While successful, CCS is limited to only systems with static topologies. Since modern Internet-scale systems are highly dynamic, mobile networks, more expressivity in the process calculus was needed.

Interaction between processes (i.e. communication) may be directed (oriented) across a channel, and calculi that include this will define an input and output primitive. Data flows from the outputting process to the inputting process, and the kinds of data that may be exchanged when interacting is a distinguishing feature of the various process calculi. In the π-calculus [Milner1992, Milner1999, Sangiorgi2001], for example, channels themselves may be in messages to other processes, allowing the process interaction topology to change over time. Moreover, processes in the π-calculus may create and maintain private communication channels between processes.

The π-calculus developed from CCS in an effort to address some dynamic issues in communicating, mobile processes, and within its scope provides a powerful tool with which to reason about the behavior of distributed systems. Though again successful, real distributed systems are far more diverse than can be modeled through even the π-calculus.

One of the most notable shortcomings is the absence of a suitable representation in the π-calculus for concept of domain that permeates today’s Internet [Hennessy2007]. Developed specifically to address this shortcoming, the Asynchronous, Distributed π-calculus was defined to reason about distributed systems in which dynamically created domains are hosts to resources that may be used by agents (i.e. processes)
In this calculus, agents reside in domains and may migrate between domains to enable the use of locally defined resources.

A closely related though fundamentally different formal approach in process calculi is the Ambient Calculus [Cardelli1998], a semantic system used to describe and reason about the behavior of concurrent systems that include mobility. While other process calculi can express certain subclasses of mobility, in ambient calculi mobility refers to both to the mobility of the devices (i.e. dynamic topology), and the mobility of computation – that is, code that is able to move around the network, such as software agents. The basic primitive of ambient calculus is the ambient – loosely defined as a bounded place where computation can occur. Some examples of ambients include the following:

- A web page.
- A virtual address space.
- A file system.
- A data object.
- A laptop.

Ambients have names that are used to control access to them, can be nested in other ambient, and can themselves be mobile. There are three basic primitives in ambient calculus – in, out, and open. Communication within an ambient (i.e. local communication) is anonymous and asynchronous. Communication across ambient is represented in various fashions involving composing such communication from the basic primitives. For example, one might model the channel-based communications from other process calculi in terms of ambients and operations on those ambients.
**The Actor Model**

Originating in 1973, the Actor Model [Hewitt1973] is a mathematical model of concurrent computation in which the primitive is an actor – a process that can receive messages, make local decisions, send messages, create more actors, and determine how to respond to other messages received. Its development was motivated thirty-five years ago by factors which today seem eerily familiar:

> “[The Actor Model’s development was] motivated by the prospect of highly parallel computing machines consisting of dozens, hundreds or even thousands of independent microprocessors, each with its own local memory and communications processor, communicating via a high-performance communications network. [Clinger1981]”

In the Actor model, there is no assumed sequencing to the actions an actor takes in response to the receipt of a message; these actions could be performed in parallel. Communication between actors occurs asynchronously. Actors that send messages do not wait for the message to be received before proceeding. Messages are addressed, and actors may send to other actors whose addresses they have, either through messages it receives or as the address of actors it has created. There is no restriction on message arrival order and computation within and among actors is inherently concurrent.

Several different formal systems have been described in the literature for the Actor model of concurrency, including operational semantics [Greif1975, Agha1993], denotational semantics [Clinger1981, Hewitt2006], and transition semantics [Agha1985]. Formal semantic descriptions of the functional programming language Erlang which uses the actor model as the basis for its concurrency have been developed [Svensson2007, Claessen2005, Roy2006].
Since both were developed to formally define and reason about concurrent distributed systems, it is not surprising that there are many similarities between the Actor model with the various process calculi and a few significant differences. These differences may be summarized as follows:

- There is only one Actor model, but many variations and incarnations of process calculi.
- The Actor model was inspired by the natural laws of physics and depends on those laws in the definition of the axioms of the model, while the roots of process calculi may be found in the mathematical formalisms of algebra.
- In the Actor model, processes (i.e. actors) have identities and direct messages to identified actors. In the process calculi, the processes are anonymous and instead communication over named (i.e. identified) channels.

Additionally, the Actor model explicitly states that communication is asynchronous, while there are variations of the process calculi in both synchronous and asynchronous communication paradigms.

**Petri Nets**

Also known as a Place/Transition Net, a Petri Net is a mathematical modeling language used to formally describe and reason about discrete distributed systems [Peterson1981, Peterson1977]. Petri nets are directed bipartite graphs where the nodes are
places and transitions. The places in a Petri net are conditions, and the transitions in the Petri net represent discrete events that may occur in the system. The directed arcs describe which places are preconditions or postconditions for which transitions. Places from which an arc connects to a transition are input places, and places to which arcs run from a transition are called output places. Places may contain a non-negative number of tokens, and the distribution of tokens over the net is called a marking. Transitions occur in Petri nets whenever there are tokens at each of the input places of that transition. When the transition occurs, a token at each input place is consumed (atomically), and one token is placed at each output place connected to the transition. The behavior of Petri nets is non-deterministic since if more than one transition is enabled at any point in time, the order in which they occur is undefined. Because Petri nets handle non-determinism and concurrency, they have been employed to model many software (and hardware) systems [Ahga2001, Cortadella2002].

**Lightweight Formal Methods**

The broad range of reasoning systems developed for the analysis of programs and programming languages provide many methods to ensure that those programs or languages behave correctly with respect to some specification of the desired behavior. Powerful, expressive calculi permit the establishment of formal properties in a very general manner yet can be prohibitively unwieldy and inaccessible to all but highly sophisticated programmers. At the other end of the complexity spectrum, lightweight formal methods are more modest systems useful for practical applications that bring
realizable verification techniques to all programmers. Such lightweight formal methods may be incorporated into real software tools such as compilers, linkers, or program analysis tools. One very successful lightweight formal method is the model checker [Baier2008, Clarke1999], a tool that searches for errors in finite state systems, often by verifying whether a given formula in the propositional logic is satisfied by the system. Another class of lightweight formal methods that is of increasing interest are the run-time monitors [Vishwanathan2004, Sherif2004], systems that detect dynamically when a component's behavior deviates from its specification.

**Type Systems**

The most successful and widely used lightweight formal method is the type system [Pierce2002, Cardelli1997], for which the following practical definition is presented [Pierce2002]:

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

Thus, type systems are formal methods used to reason about programs, though more abstract applications for type systems form a distinct and significant branch of the field in which the type system is used to reason about programming languages as in the typed \( \lambda \)-calculi [Barendregt1992].

Type checkers are typically built into compilers, linkers, and run-time systems to prevent certain kinds of program misbehaviors, and are necessarily found in both static and dynamic variants. Type checkers are useful in helping guarantee ‘language safety’ –
preservation of the language’s abstractions. Languages such as Java, ML, and Haskell rely on static type checking to ensure safety, while safe languages such as Lisp, Scheme, and Perl dynamically check types. However, type checking does not always guarantee language safety – C and C++ are unsafe languages that use static type checking.

**Semantics of Real Time, Reactive Systems**

The formal methods for reasoning about the behavior of systems and programs discussed thus far only hint at the depths to which formal methods have been applied in the study of such systems. However, there is an important subclass of programs and systems for which highly specialized reasoning systems are needed. This subclass consists of real-time, reactive systems. Real-time computing is the study of hardware or software systems subject to bounds from event to response time – i.e. a ‘real time’ constraint [Liu1973, Halang1994]. Real-time systems are correct not only if their behaviors are correct but also occur within prescribed time limits. Hard real-time systems are the subclass of real time system in which the results are useless if the time limit is not met, while soft real time systems may tolerate some degree of lateness – possibly with a degraded service level. Mission-critical real-time systems are those upon which the success or failure of the entire system depends; life-critical real-time systems are those whose failure would likely result in the loss of human life. It should not be surprising, therefore, that there is a much research described in the literature towards developing reasoning systems for the study and verification of real time systems [Heitmeyer1996, VanTilborg1991].
Reactive systems are those that maintain an ongoing interaction with their environment [Manna1992]. Traditional programs and systems, regardless of their degree of concurrency, are transformational – given an input (and sometimes a state), determine a result. Consequently, reasoning systems that characterize the relationship between such a system’s inputs and outputs cannot evaluate a reactive system that may by design never terminate.

Reactive systems are typically real time systems. An example of soft real time reactive system is a graphical operating system – the system is meant to respond to user inputs and allow interaction between various processes, the operating system, and the user(s), however there is no hard limit set for those responses. Examples of hard real time reactive systems include fly-by-wire flight controls, power generation or chemical processing plant control, and other mission-critical and life-critical processes [Benveniste2003] (see the previous section on real-time embedded control, specifically the discussion on synchronous languages).

*Temporal Logic-based Formal Methods*

The formal systems used to reason about the behaviors of systems discussed thus far often differed in the logic system upon which they were founded. By definition, any logic system that represents and is used to reason about propositions quantified in terms of time is a temporal logic [Rescher1971, Ohrstrom1995, Gabbay1994, and Gabbay2000]. Temporal logic has two kinds of operators – logical operators and modal operators. The logical operators are typically the truth-functional operators (¬, ∧, ∨, →),
while the modal operators might include Until, Release, Next, Future, Globally, All, and Exists.

Interval temporal logic is a temporal logic in which both propositional and first order logical reasoning is supported over finite time intervals [Cau2008]. The Hennessy-Milner logic is a temporal logic used to specify properties of a labeled transition system similar to an automaton [Hennessy1980]. A temporal logic-based framework for modeling, specifying, and verifying systems composed of real-time discrete event processes was described in [Ostroff1989]. A temporal logic for reactive and concurrent systems was described in [Manna1992]. Verification of timed Petri nets and timed automata were discussed using a temporal logic-based approach [Penczek2006]. An executable form of temporal logic (i.e. one in which logic statements are executed as statements in a computer program) has also been described in the literature [Barringer1996].

**Duration Calculus**

Duration calculus (DC) represents a formal method for reasoning about and the formal design of real-time systems [Chaochen2004]. Duration calculus extends interval temporal logic to a calculus in which real numbers are used to represent time and Boolean-valued functions over time represent the states of real-time systems. The formalization of DC includes state transitions, events, and superdense transitions – transitions which are assumed to be timeless (i.e. negligible with respect to the time interval). These formalizations permit the handling of safety, liveness and fairness.
properties. Theoretical results on completeness and decidability have been described, and a model-checking algorithm and other support tools have been published.

**Compiler Design and Optimization**

A compiler is a computer program or set of programs that transform a computer language program into another computer language [Aho2006, Cooper2004, Wirth1996, Scott2005]. Generally, the source code input to a compiler is a high-level language and the output from the compiler is a lower-level language (machine or assembly language) formatted for processing by a linker. Compiler output typically targets a particular platform.

Modular compiler design developed from the Production Quality Compiler-Compiler (PQCC) Project at Carnegie Mellon University [Leverett1980]. This compiler design paradigm decomposes compiler architecture into a sequence of relatively independent phases that are easier to work with in large development groups and easier to replace with upgrades. Moreover, inserting additional phases later in the compiler life cycle (for example, adding a new optimization) is easily supported. In the PQCC project, there were three named phases – the front end, the middle end, and the back end.

Though its scope is open to some debate, the front end typically encompasses the syntactic and semantic processing of the compiler, ending with the transition to a lower level of code representation (an Intermediate Representation (IR)). The middle end typically consists of architecturally-neutral optimization phases that operate on the IR. Since the IR is both architecturally neutral and source language independent, the middle
end is intended to be used in a generic manner in compilers for different programming languages and targeting differing architectures. The back end consumes the output from the middle end, and after performing any additional analyses, transformations, and optimizations specific to the target architecture, the back end generates code for the target architecture.

A compiler front end typically performs some or all of the following functions:

- **Lexical Analysis**: A lexical analyzer or scanner breaks the text of the source code down into tokens – classifiable, atomic units of the language.
- **Preprocessing**: A compilation phase that can be required before further compilation supporting macro definitions and conditional compilation.
- **Syntax Analysis**: The rules of the formal grammar of the language are used to identify the syntactic structure of the program, typically by parsing the sequential token stream into a parse tree.
- **Semantic Analysis**: A compilation phase in which syntactic structure of the program is augmented with semantic information. Semantics checks such as type checking typically occur at this time.

Compiler back ends typically include the following:

- **Analysis**: IR is analyzed to provide a basis for the optimizations to follow. These analyses might include data flow analysis [Hecht1977], use-define chains, dependence analysis, and others. Additionally, the call graph [Grove1997, Callahan1990] and control flow graph [Aho2006] are generally built during these analyses.
- **Optimization**:
• Code Generation:

Compilers may be classified as single-pass or multi-pass compilers, depending on the number of times the compiler reads the source code.

Microsoft is crafting a framework for software optimization and analysis to serve as the foundation of its compiler technologies [Microsoft2008]. This framework, code named Phoenix, is an extensible system for building compilers as well as a wide range of tools for program analysis, optimization, and testing. Central to this framework is the multi-level IR in which varying degrees of architectural independence are maintained. The framework includes a powerful set of IR components, including flow graphs, region graphs, and multi-model exception handling to yield high-quality, robust and correct compilers, analyses, and optimizations.

Compilers for Parallel, Concurrent Languages

Computer languages for expressing parallel, concurrent languages can be broadly categorized according to how communication occurs between concurrent processes [Lin2008]. In shared memory programming, communication is accomplished through manipulation of memory variables shared by the communicating processes. In distributed memory programming, explicit messages are used to communicate between processes. POSIX Threads [Butenhof1997] and OpenMP [Chapman2008, Quinn2004] are two widely used shared memory APIs, while the Message Passing Interface (MPI) [Quinn2004] is a widely used distributed memory API. C++ [Hughes2004] and Java [Wellings2004] have also been widely used to implement concurrent programs in either
category. Additionally, there is support for concurrent threads in the Boost library [Karlsson2006, Boost2008]. These mechanisms for generating concurrent applications rely on standard programming languages like C++ and Java, and as such they may use the ‘regular’ compilers for those languages. Most significantly, since no special compiler is needed for such approaches to concurrency, it is not surprising that the work of defining how these programs should be parallelized must be done by the programmer through explicit program statements or directives.


Despite decades of work by compiler researchers, the development of a compiler that would automatically parallelize a sequential program – the ‘holy grail’ of parallel computing – has been only partially successful [Shen2005]. Though limited in scope, there have been a few successful implicitly parallel languages. One of these is SISAL, a functional, dataflow programming language featuring strict semantics and Pascal-like syntax for which partitioning, optimizing compilers have been developed [Sarkar1990]. O’Haskell is an object-oriented, concurrent, functional programming language extending Haskell [Nordlander1999].
Chapter 3

C3L Description and Syntax

Introduction

The C3L language was designed to allow a direct implementation of high-level abstractions created in modeling complex distributed systems. Presented in this chapter is the grammar of C3L as a formal syntax – a rule system governing how programs written in this language are structured.

As stated earlier, the previous version of C3L had significant syntactic shortcomings. For example, \(1 + 2 + 3\) was not a valid C3L expression! This grammar had no operator precedence defined and no operator associativity defined. Consequently, evaluation of expressions required the explicit use of parentheses to prevent ambiguity. Furthermore, redundant use of parentheses was prohibited. So \(1 + 2\) was a valid C3L expression, but \((1 + 2)\) was not. One might argue that the evaluation of C3L expressions was unsound with respect to the basic properties of algebra of associativity and commutativity based on this because such properties cannot be expressed in the language. Also, the usage of parentheses in Boolean expressions was different, but still required to prevent ambiguity. For example, \(\text{true and false}\) was not a valid Boolean expression, but \((\text{true})\ \text{and} \ (\text{false})\) is. One might argue that C3L was unsound with respect to the laws of Boolean algebra since the language cannot generate the expressions of Boolean algebraic theorems. More significantly, the formal grammar
of the language had unconstrained rules, included terminal characters that are not part of
the terminal character set (inequality, less than and greater than), and only loosely
defined what integers, real, characters, and labels. Fundamental questions that the
grammar should answer unequivocally were left unanswered, such as: Is it legal to
express a real number as 1e-5? Is it legal to express an integer as +2? Are punctuation
characters and spaces legal in labels? Since the formal grammar is the foundation upon
which everything about the language is built, any imperfections in its rules can weaken
the set of formal systems built upon it.

There were also other issues in the original syntax that affected the language’s
acceptability as a formal programming language. For selective execution of statements,
the original syntax did not have if-else statements, but rather had choice-‘cor’ statements.
A choice is like an if statements except there was no way to specify an else. Instead, the
choice syntax permitted zero or more ‘cor’ clauses, each consisting of the keyword ‘cor’
followed by one or more statements. However, the ‘cor’ clause brings nothing unique
into the language as one may equivalently use nested if statements in their place. And if
the functionality provided by the choice statement was identical to that of the ubiquitous
if statement, why call it a choice?

At a higher level, the original C3L syntax featured a group as a primitive data
type. Group variables were used to identify groups of devices for the purposes of sending
and receiving events, and C3L provided some operations on this group type. Groups were
implemented as lists of strings, and to statements were included in the language to
transform a group – gadd and grem – which added or removed, respectively, a label or
another group to or from a group. A relational operator, in, was implemented to test for
inclusion of a string or a label in a set. It was observed, however, that these groups and
the operations provided for them were simply a subset of standard sets and set operations.
Though more problematic from a semantics point of view, the syntactic implementation
of groups was cumbersome, and since the implemented functionality was a subset of set
theory, it seemed as though a better approach was available to express group dynamics in
C3L.

**Contributions to C3L Syntax**

With such fundamental problems in the lower-level grammar rules in C3L,
significant changes were necessary. The syntax was heavily revised to define operator
precedence and associativity in a mathematically sound manner (for example, \(a + b * c\)
evaluates \(b * c\) first, then adds that product to \(a\).) The syntax was structured to
permit arbitrarily deep, redundant use of parentheses. These changes alone were
sufficient to legalize those expressions that were mathematically sound but unnecessarily
rejected by the syntax as illegal. While most operators were implemented with left-
associativity, the assignment operator was implemented with right-associativity to permit
chained assignments such as \(a = b = c = 0;\).

The formal grammar was extended to unambiguously define the exact characters
that may appear in identifiers as well as defining what manner in which integer, real,
string, set and Boolean literals may be represented in a valid C3L program. Additionally,
the standard if-else statements replaced the choice-cor statement for selective execution.
The original C3L syntax had no syntactic support for repetition. To implement repetitive processes, a device had to recursively raise events to itself. The reason for omitting loops from the language was that there was no way to statically verify that the loops would terminate, and as will become clear this would undermine some formal properties desired in the language. In the revised syntax, both ‘for’ and ‘while’ loops are included in a form that static analysis can guarantee that the loops will eventually terminate, preserving the formal properties of the language. Additionally, the syntax was extended to allow for the definition of local variables and to permit the extension of the modifiers that may be specified in variable declarations.

To avoid the idiosyncrasies with the original syntax of groups and to found groups and their operations on the widely known and formalized set theory, the concept of a group was removed from the language and replaced by an implementation of the set as a primitive data type. An extension of the standard operators of the language was made to fully support the set operations of union, difference, complement, intersections, subset of, proper subset of, inclusion, superset of, and proper superset of. Additionally, C3L concepts of all, me, you and the empty sets were reformulated according to this new type.

The syntax for C3L presented here implements the improvements and extensions described while maintaining the higher-level operational model which is presented in [Phoha1999].
Chapter Roadmap

This chapter presents the major elements of C3L syntax alongside a detailed description of C3L, including a discussion of the plant and controller model abstractions upon which C3L is built as well as the data types implicit in the language. Since most of the programming action occurs in the control event handlers, most of the statement-level syntax is present in the context of that C3L construct. The complete syntax is then presented and the chapter concluded.

Description of C3L

A C3L program is defined as a sequence of device definitions. Each device consists of a plant, a controller, and an event queue. These specifications include a description of the variables that quantitatively define the current state-space of the device and the specific set of transitions that may occur in the state-space; collectively these specifications are referred to the plant. Transitions in the state-space are defined by a set of plant event handlers that are further classified as either controllable plant events or uncontrollable plant events. A controllable plant event is one whose occurrence is completely under the control of the device, while an uncontrollable plant event is one whose occurrence is completely out of the direct control of the device. For example, the event of a robot device hitting an obstacle is an uncontrollable event, while actuating a robot’s ultrasonic sensor to search for obstructions is a controllable event.

The controller of a C3L event consists primarily of a set of control event handlers that constitute the high-level behavioral description of the device. Control event handlers
are executed when the raised, either by the device itself or external devices as authorized. When raised, control events are added to the input event queue for the device. Execution of the event handler occurs when the event reaches the head of the input queue, and while executing control event handlers may exercise both control and communication by raising other control events. Communication occurs when control events are raised to external devices, while control occurs in two ways – when a device raises control events to itself (high-level device control), or when a device executes a controllable plant event (low-level device control).

Finally, a C3L device requires an input event queue to properly store received events until they can be processed. This queue is implemented as a simple first-in, first-out (FIFO) buffer yet plays an important role in enabling the formal controllability and behavioral properties of the C3L.

The complete conceptual model of a C3L device is illustrated in Figure 3-1. The basic structure of a C3L device specification is shown in Figure 3-2. Every C3L device has an input event queue so no declaration is needed for this component of the device.

Before embarking on the discussion of higher-level program features of C3L, a very brief mention of some relevant low-level features is necessary. C3L is a strongly typed language and supports five basic data types – real, integer, boolean, string, and set. The C3L set is a Pascal-style general-purpose bit vector implementation used for particular applications within C3L. Finally, C3L is case-sensitive and supports C++-like associativity and order-of-operations.
The Plant Model

The plant model definition of a C3L device consists of the set of state-space variables (plant parameters) and the set of plant event handlers that define the transitions that may occur in the device’s state-space. Resembling simple variable declarations, parameter definitions implement the state-space of the device. The device’s state-space may include internal components as well as components that quantify a view of the device’s environment (e.g. situational awareness). Though cosmetically similar to memory definitions described in a following section, parameter definitions are semantically quite distinct and may only change through the occurrence of plant events.

Figure 3-1: C3L Conceptual Model
Plant events are classified as either controllable or uncontrollable. Uncontrollable events are those whose occurrence is outside of the control of the device, and handlers for these events define how the plant state changes through a list of assignment statements. Syntactically, these event handlers resemble standard function definitions and may include formal parameter definitions. Additionally, a precondition for the event may be specified as a Boolean expression in the plant parameters and/or formal parameters of the event handler; these preconditions define the conditions under which the event is permitted (i.e. the plant state may change). If the precondition of an uncontrollable event is met, the assignment statements of the uncontrollable event are executed and the event is placed in the input queue. In this way, a control event may be implemented to describe the changes in plant state required when a valid uncontrollable event is received.

Uncontrollable plant events are atomic – whenever an uncontrollable event is received with a precondition evaluating to true, that event’s uncontrollable event handler
is immediately executed, pre-empting whatever control event is currently executing. This allows the greatest possible fidelity in the plant model’s representation of the physical device state. Consequently, plant parameters may change while a control event is executing. Therefore, plant parameters must be treated as volatile variables in the context of the control events.

Controllable event definitions are syntactically identical to uncontrollable event definitions. Semantically, controllable events differ in that they occur only when the device controller triggers them and are not added to the input event queue at any time. Additionally, since controllable plant events are only invoked by the controller’s control events, controllable plant events do not pre-empt control events in the same way that uncontrollable events can. Like uncontrollable events, controllable events are themselves atomic and cannot be pre-empted even though the control events can themselves be.

Consider the example C3L plant model in Figure 3-3 (a). This device’s plant model consists of two parameters – Temp and criticalBatt. The Temp parameter may change only in the event of Heat or Cool. As an uncontrollable event, Heat occurs outside of the device’s control – at least as far as the C3L program is concerned. Since there is a precondition to the Heat event, the precondition must evaluate to true before the event is processed. The Cool event operates in much the same way as Heat, differing only in its controllability. As an uncontrollable, the LowBatt event is triggered by the device hardware when appropriate, and when it occurs the plant state is updated in the event handler, then the event is added into the input event queue. This permits the controller, if it has an event handler for this event, to take further control action as necessary – perhaps by initiating low-power operations and navigating to a recharge station.
The Controller Model

The C3L Controller Model is implemented primarily through the definition of control event handlers, supported by non-plant device-global variables.

C3L Memory Variables

C3L memory variables are most like traditional global variables in general programming languages but with a scope limited to the device (remember that C3L
programs are lists of device specifications). Device memory variables may not encode plant state information, and their values must change independent of plant events. Note that plant parameters are visible in the device controller context as read-only variables, but device memory variables are not in scope from the context of plant event handlers. Figure 3-3 (b) illustrates an example C3L controller definition.

C3L Sets

A language for distributed control of autonomous systems requires a means of expressing group semantics. C3L provides the set as a native data type, and C3L programs may specify any number of set variables as needed to express groups of devices in set notation; these sets are then commonly used in directing and authorizing controller events.

There are four pre-defined sets in C3L defined as follows:

(a) The **all** set, containing all known devices.

(b) The **empty** set, containing no devices.

(c) The **me** set, containing only the device of the context in which the **me** set appears. The **me** set is analogous to the ‘this’ concept in object-oriented programming languages. For convenience, the device name is an alias for the **me** set; consequently, the device name may be used in set expressions, direct and authorize arguments, etc. Syntactic support for forward declaration of devices is also included in C3L to further support the use of device names as set variables.
(d) The **you** set, containing only the sender device of an event processed by a device context.

C3L set operations consist of **intersect** (*), **union** (+), **difference** (-), **subsetOf** (<=), **supersetOf** (>=), **properSubsetOf** (<), **properSupersetOf** (>), set inclusion (in), and set complement (prefix -). Words bolded in this description are C3L keywords defined as synonyms for the set operator listed in parentheses.

Since C3L implements a complete set of set operators, dynamic modification of sets of devices within the context of complex, dynamic distributed systems of collaborating devices, is intuitive and straightforward.

### C3L Control Events

Control events constitute the behavioral specification of the C3L device controller. The syntax of C3L control events in modified Backus-Naur Form (BNF) [Naur1963] is shown in Eq. 3.1.

\[
\text{event identifier}(\text{paramDefList}) \ [\text{authorize} \ \text{logicOrExpr}] \ [\text{executeOnStart}] \ \text{blockStmt} \quad 3.1
\]

Identified by name, control events are structurally analogous to traditional function definitions yet differ considerably in *when* (and if) control events are executed. In a traditional function, execution of the function occurs when the function is called and execution within the calling context is suspended until the function returns. A C3L event is invoked by raising the event; however raising the event neither immediately nor automatically results in the event’s execution. Raising an event in C3L simply adds that event to one or more device input queues. Each control event may have an explicit
authorize argument of set type, denoted by the optional clause “authorize logicOrExpr” in Eq. 3.1. This authorization expression must evaluate to a value of type set; this set specifies those devices from which this control event will be recognized. If a control event does not have an explicit authorize argument, the default authorize set of me is used, thus permitting execution of the control event only by the device raising it. When an event is removed from the input queue, its authorize specification is checked by evaluating “you in logicOrExpr”. If true, the event is executed; if false the event is discarded.

The optional attribute executeOnStart may be defined for any control event with no message parameters. When applied, this indicates that the event should be raised to the device when execution of the program begins.

Low-level plant control of the C3L device is accomplished by invoking controllable events from within control event handlers. Controllable events are called by name much like traditional function calls and include arguments when applicable. The C3L statements “MoveEast;” and “Klaxon(3)” invoke controllable plant events defined in Figure 3-3 (a).

Raise Statements

The primary mechanism that permits both high-level control of the C3L device and communication between C3L devices is raising control events. The syntax of C3L raise statements is shown in Eq. 3.2 in modified BNF.

\[ \text{raise identifier[(paramList)] [direct logicOrExpr] ;} \quad 3.2 \]
The raise statement identifies the control event to be raised by name, optionally packaging parameters along with the event (if any). The optional direct clause indicates the device or devices to which this event should be raised, and the \(<\text{logicOrExpr}>\) argument of the direct clause must evaluate to type set. When no direct clause is present, the default target of all C3L devices is implied. A C3L device exercises high-level control over itself by directing control events at itself; C3L devices perform communication by directing control event at other devices. For example, the C3L statement “\texttt{raise ViolentFailure direct broadcast;}” exercises both control and communication because the direct list (as defined in Figure 3-3 (b)) consists of a proper superset of me, while the example C3L statement ”\texttt{raise SearchPattern(1) direct team1 - me;}” exercises communication since ‘me’ is obviously not part from the direct list. It is this fusion of command and control that permits C3L’s natural and intuitive expression of complex distributed system control.

If Statements

In addition to the raise statement, C3L provides several other basic statements. The language provides for select execution of program statements through the ubiquitous if-else statement. The syntax of the if-else statement is illustrated in Eq. 3.3. C3L semantics require the logicOrExpr control expression to evaluate to a value of Boolean type.

\[
\text{If ( logicOrExpr ) statement [ else statement] ;}
\]
Assignment Statements

C3L provides assignment statements in a syntax similar to today’s modern general-purpose languages. Assignment operators may be ‘daisy-chained’ (e.g., \(a = b = c = 0;\)); however C3L syntax prohibits assignment statements from appearing as expressions. For example, the statement “\(\text{if } (a = b) \ <\text{statement}>\)” is not permitted by C3L syntax.

Repetition Statements

C3L provides repetition structures that are semantically limited to a statically verifiable number of iterations. This guarantees that no event handler can continue ad infinitum and supports the controllability of devices using the language. The syntax of C3L for loops (Eq. 3.4) and while loops (Eq. 3.5) is presented in modified BNF.

\[
\text{for ( identifier = constExpr to constExpr [ by constExpr ] ) statement}
\]

\[
\text{while ( logicOrExpr , constExpr ) statement}
\]

Since constExpr are required to be known at compile-time, the for loop syntax ensures that static analysis will always determine the exact number of iterations required in executing the for loop. More importantly, C3L for loops can never be infinite. The syntax for C3L while loops ensures a similar property by including, in addition to the specification of a Boolean loop control expression, a statically-verifiable iteration limit for the loop. While the Boolean loop control expression is true and the iteration count is
less than the statically-defined iteration count limit, the loop repeats. Both repetition
structures provided by C3L enable non-recursive repetition in a manner which preserves
the formal properties of the language to be discusses in subsequent sections.

C3L events are structurally similar to functions in many regards; however the
language does permit device-specific functions. Syntactic support for these implicit
functions is found in the postfixExpr rule.

For direct documentation of C3L programs, the \textasciitilde{} character is designated as an
end-of-line comment symbol. To maintain consistency with other modern languages, the
// comment sequence is also recognized as an end-of-line comment symbol.

\textbf{C3L Syntax}

The complete formal syntax of C3L is shown in Tables \textbf{3-1} and \textbf{3-2}. 
### High-level C3L Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>deviceList</td>
</tr>
<tr>
<td>deviceList</td>
<td>device</td>
</tr>
<tr>
<td>device</td>
<td>device identifier</td>
</tr>
<tr>
<td>identifierList</td>
<td>identifier</td>
</tr>
<tr>
<td>par</td>
<td>parameter varList</td>
</tr>
<tr>
<td>unc</td>
<td>uncontrollable peventList</td>
</tr>
<tr>
<td>con</td>
<td>controllable peventList</td>
</tr>
<tr>
<td>mem</td>
<td>memory varList</td>
</tr>
<tr>
<td>control</td>
<td>control ceventList</td>
</tr>
<tr>
<td>peventList</td>
<td>pevent</td>
</tr>
<tr>
<td>ceventList</td>
<td>cevent</td>
</tr>
<tr>
<td>pevent</td>
<td>identifier</td>
</tr>
<tr>
<td>blockExprStmt</td>
<td></td>
</tr>
<tr>
<td>cevent</td>
<td>event</td>
</tr>
<tr>
<td>executeOnStart</td>
<td>blockStmt</td>
</tr>
</tbody>
</table>

### Statements

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>exprStmt</td>
<td>[expr] ;</td>
</tr>
<tr>
<td>blockExprStmtList</td>
<td>exprStmt</td>
</tr>
<tr>
<td>blockExprStmt</td>
<td>{}</td>
</tr>
<tr>
<td>statement</td>
<td>localVar</td>
</tr>
<tr>
<td>blockStmt</td>
<td>{}</td>
</tr>
<tr>
<td>stmtList</td>
<td>statement</td>
</tr>
<tr>
<td>selectStmt</td>
<td>if (logicOrExpr) statement</td>
</tr>
<tr>
<td>raiseStmt</td>
<td>raise identifier</td>
</tr>
<tr>
<td>repStmt</td>
<td>for (identifier = constExpr to constExpr [by constExpr])</td>
</tr>
</tbody>
</table>

### Expressions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>primaryExpr</td>
<td>literal</td>
</tr>
<tr>
<td>postfixExpr</td>
<td>primaryExpr</td>
</tr>
<tr>
<td>unaryExpr</td>
<td>postfixExpr</td>
</tr>
<tr>
<td>multiExpr</td>
<td>unaryExpr</td>
</tr>
<tr>
<td>addExpr</td>
<td>multiExpr</td>
</tr>
<tr>
<td>relExpr</td>
<td>addExpr</td>
</tr>
<tr>
<td>logicAndExpr</td>
<td>relExpr</td>
</tr>
<tr>
<td>logicOrExpr</td>
<td>logicAndExpr</td>
</tr>
<tr>
<td>assignExpr</td>
<td>logicOrExpr</td>
</tr>
<tr>
<td>expr</td>
<td>assignExpr</td>
</tr>
<tr>
<td>constExpr</td>
<td>logicOrExpr</td>
</tr>
<tr>
<td>paramList</td>
<td>logicOrExpr</td>
</tr>
<tr>
<td>constExprList</td>
<td>constExpr</td>
</tr>
</tbody>
</table>
### Variable Declarations

<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>typeSpecifier</td>
<td>::= [const] type</td>
</tr>
<tr>
<td>typeSpecSeq</td>
<td>::= typeSpecifier</td>
</tr>
<tr>
<td>initClause</td>
<td>::= logicOrExpr</td>
</tr>
<tr>
<td>initList</td>
<td>::= initClause</td>
</tr>
<tr>
<td>initializer</td>
<td>::= = initClause</td>
</tr>
<tr>
<td>initDeclarator</td>
<td>::= declarator [initializer]</td>
</tr>
<tr>
<td>declaratory</td>
<td>::= identifier</td>
</tr>
<tr>
<td>decList</td>
<td>::= initDeclarator</td>
</tr>
<tr>
<td>var</td>
<td>::= typeSpecSeq decList</td>
</tr>
<tr>
<td>localVar</td>
<td>::= typeSpecSeq decList</td>
</tr>
<tr>
<td>varList</td>
<td>::= var</td>
</tr>
<tr>
<td>paramDef</td>
<td>::= type declarator</td>
</tr>
<tr>
<td>paramDefList</td>
<td>::= paramDef</td>
</tr>
</tbody>
</table>

### Operators

<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>relEqOp</td>
<td>::= ==</td>
</tr>
<tr>
<td>relOp</td>
<td>::= relEqOp</td>
</tr>
<tr>
<td>multiOp</td>
<td>::= *</td>
</tr>
<tr>
<td>addOp</td>
<td>::= +</td>
</tr>
<tr>
<td>unaryOp</td>
<td>::= not</td>
</tr>
<tr>
<td>assignOp</td>
<td>::= =</td>
</tr>
</tbody>
</table>

### Literals and Identifiers

<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>digit</td>
<td>::= 0</td>
</tr>
<tr>
<td>hexDigit</td>
<td>::= digit</td>
</tr>
<tr>
<td>charNondigit</td>
<td>::= a</td>
</tr>
<tr>
<td>character</td>
<td>::= digit</td>
</tr>
<tr>
<td>identifier</td>
<td>::= charNondigit</td>
</tr>
<tr>
<td>hexNumber</td>
<td>::= 0x hexNumber</td>
</tr>
<tr>
<td>number</td>
<td>::= digit</td>
</tr>
<tr>
<td>sign</td>
<td>::= +</td>
</tr>
<tr>
<td>exponent</td>
<td>::= e [sign] number</td>
</tr>
<tr>
<td>intLiteral</td>
<td>::= number</td>
</tr>
<tr>
<td>fixedLiteral</td>
<td>::= number .</td>
</tr>
<tr>
<td>realLiteral</td>
<td>::= fixedLiteral [exponent]</td>
</tr>
<tr>
<td>charString</td>
<td>::= character</td>
</tr>
<tr>
<td>stringLiteral</td>
<td>::= &quot;[charString]&quot;</td>
</tr>
<tr>
<td>setLiteral</td>
<td>::= all</td>
</tr>
<tr>
<td>literal</td>
<td>::= intLiteral</td>
</tr>
<tr>
<td>boolLiteral</td>
<td>::= true</td>
</tr>
<tr>
<td>type</td>
<td>::= integer</td>
</tr>
</tbody>
</table>
Summary

C3L was designed for model-based design and development of complex distributed systems. The language was described and a complete syntax was presented. The syntax presented has been completely revised from that of previous versions of the language which, frankly, was rather ad-hoc in design and had myriad problems for which there were readily available solutions. Some of these problems resulted from the attempt to use the formal grammar to enforce typing, some resulted from an attempt to avoid ambiguity in the language, and others resulted from the lack of a defined precedence and associativity for the operators. Most significantly, the revised syntax supports mathematically sound operator precedence and associativies consistent with established high-level languages like C++. The original C3L syntax did not support the set data type; instead a rather cumbersome specification of a group was employed using a subset of set operations mixed with string arguments to provide a mechanism through which the semantics of dynamic organization of collaborating devices could be specified. In the revised syntax, this mechanism is completely provided by the set abstraction as presented, allowing for a stronger theoretical foundation for the expression of group dynamics through a general-purpose data type and a set of operators on that type that may also be used for other purposes. Additionally, syntactic support for local variables and for repetitions consistent with the design goals of C3L extend the language.

Built on the strong foundation of a well-defined and expressive lower-level syntax, the C3L operational model provides a natural and intuitive plant/controller abstraction suitable for complex distributed-system development. In C3L, communication
between devices shares the same abstraction as high-level control of a single device, enabling the application developer to focus on the high-level behavioral description of the distributed system instead of becoming consumed with the low-level details of how inter-device communication is achieved. Low-level control of a C3L device involves the specification of a plant through sets of state-space parameters and plant event handlers that describe the set of all possible state transitions. The C3L device controller exercises control over its plant through invocation of controllable plant events – the subset of plant events within the direct control of the controller. The remaining plant events – the uncontrollable plant events – are those plant events outside of the direct control of the device controller; these events are triggered directly by the device hardware to be controlled and may include such events as GUI user interface events, sensor detections, hardware fault detections, etc., as appropriate for a particular application. The complete plant and controller model constitute the complete behavioral description of the device by defining the specific actions required as the dynamics of the distributed system evolve over the lifetime of the C3L application.
Chapter 4

C3L Operational Semantics and Formal Language Properties

Introduction

Formal properties of a language provide a mathematical foundation for the specification of the language, enabling verification of program behaviors that have been fundamental requirements for safety-critical applications. This foundation is established through the formal semantics of the language.

There are several widely accepted ways to describe the behaviors of programming languages and computing devices, including operational semantics, axiomatic semantics, denotational semantics, lambda-calculus and pi-calculus to name a few [Harrison1978, Winskel1996, Gunter1992, Reynolds1998, Hennessy1990, Scott1970, Hankin1994, Krivene1993, Milner1999]. Operational semantics and formal results for C3L have been presented in the literature previously [Schmiedekamp2006, Skarbez2004], however there were significant problems identified in their derivations. In addition to being incomplete, the previous semantics had unconstrained variables in several inference rules. Many inference rules rely heavily on specially defined functions in their definitions to express complex functionality. Consequently, the validity of the rule depends on the assumption that the function operates correctly. This ‘black box’ approach to giving meaning to a programming language tends to defeat the formality of the reasoning system by introducing non-trivial assumptions. In the previous semantics, there does not appear to
be any mechanism which an event is transported from the output queue of one device into the input queue of another device; since this is a fundamental unit of work in the C3L operational model, a lack of formal (or any) description of this process is a serious problem. Most significantly, the evaluation operator used in the semantics is overloaded with respect to type. Since these operational semantics define meaning through inductively-generated sets, typing is critically important to avoid the logical paradoxes that threatened the foundation of mathematics in the early twentieth century [Russell1903, Cardelli1997]. Consequently, having weakly typed evaluation operators employed in the semantics immediately introduces an unnecessary weakness in the calculus. Moreover, since the syntax for the language was heavily revised as described in the previous chapter, and since structural operational semantics are driven by the syntax, simply changing the syntax is sufficient justification to revisiting the semantics.

**Contributions to C3L Semantics**

For all practical purposes, there are only shadows of the original semantics remaining in the operational semantics presented here. This calculus uses a set of strongly typed, well-defined set of evaluation operators in expressing its inference rules. The few functions used in its formal description are well-known and/or well-defined. The device context (i.e. state) was simplified to reflect the replacement of groups by set variables, and the output buffer was rendered superfluous in the semantics by the novel ways in which events and their affect on the device context are expressed. Semantics for the syntactic extensions to C3L are provided; most notably, the semantics of repetitions
structures are presented that preserve the property of C3L event handlers that they always terminate in finite time – a result necessary for the event fairness property of the language.

Concurrency in C3L Semantics

The operational semantics of C3L are presented here to literally give the ‘meaning of operations’ in C3L programs. In itself, C3L is not a language through which multithreaded programs may be expressed, but rather a high-level language to describe the behaviors of distributed systems of interacting devices. C3L neither requires multithreading implementations nor explicitly precludes it. There is also no explicit mechanism in C3L that allows the programmer to dictate when or where to create threads or to define how threads interact. However, the C3L programming paradigm enables multithreading by exposing clear task-level parallelism throughout its device model. Since actual instantiations of C3L programs may be either single threaded or multithreaded, it was decided to formalize the behavior of the language through an operational semantics that describes what operations are required by C3L language constructs both in the context of a single device and also in the larger context of the set of device contexts that is a C3L program. Any threading model that satisfies these operational semantics would therefore be a valid instantiation of the C3L programming language. This is accomplished formally by defining inference rules governing how the device contexts of two devices may interact, completely avoiding any discussion of processes and the requisite elevation of the semantics to a process calculi.
Chapter Roadmap

Following are several formal results which enhance the utility of specifications in C3L beginning with a structure operational semantic [Winskel1996, Plotkin2004, Plotkin2004a] description of the language’s behavior on an abstract device. First, the foundations for the semantics are provided through the definition of syntactic sets, metavariables, notations, symbols, and functions used in the operational semantics. The device context is then defined, and then the operational semantics begin with rules governing the evaluation of values and expressions. The operational semantics continue into the execution of increasingly complex statements and include calling functions, occurrence of events, raising events, etc. The operational semantics conclude with a presentation of the rules governing the behavior of the event queue. Finally, the chapter concludes by showing that the previously demonstrated formal properties of C3L - that all C3L programs are prefix-closed controllable and treat events fairly – are still provable properties of the language using the new semantics.

C3L Operational Semantics

C3L is an event-driven language, requiring formal semantics that include a behavioral description of the input event queue of a device. Since the input queue, plant parameters, memory and local variables could all be modified at any point, a complete description of the device context is necessary to formalize the semantics of event response. Additionally, several static look-up tables are needed to identify and classify events as control events, controllable plant events or uncontrollable plant events as the
semantics for these events are different. The formal semantics begin with descriptions of these structures, and then build on this foundation in defining the higher-level semantics of C3L. In the following discussion, the empty object is designated as $\varepsilon$.

**Notations**

The following syntactic sets are defined to support the formal definition of C3L:

- Reals $\mathbb{R}$
- Integers $\mathbb{Z} \subseteq \mathbb{R}$
- Boolean values $B \in \{true, false\}$
- Sets $Se$
- Strings $St$
- Arithmetic Expressions $Aexp \supseteq \mathbb{R}$
- Boolean Expressions $Bexp \supseteq B$
- Set Expressions $Seexp \supseteq Se$
- String Expressions $Stexp \supseteq St$
- Non-string Expressions $H = Aexp \cup Bexp \cup Seexp$
- Expressions $E = Aexp \cup Bexp \cup Seexp \cup Stexp$
- Constant Expressions $O$
- Plant Parameters $P$
- Memory Variables $M$
• Statements \( S \)

• Values \( V \equiv S \cup R \cup B \cup Se \cup St \)

• Variables (named memory locations) \( L \equiv P \cup M \)

• Control Events \( J \)

• Controllable Plant Events \( C \)

• Uncontrollable Plant Events \( U \)

• Input Events \( I \equiv J \cup U \)

• Types \( Y = \{ \text{integer, real, boolean, string, set} \} \)

• Identifiers \( D \)

• Implicit Functions \( IF \)

• Callable Functions \( F = IF \cup C \)

• Devices \( G \subseteq Se \)

Metavariables are defined for use in the formal semantics of the language in Table 4-2 and may appear primed or subscripted. The standard symbols used in C3L semantics are defined in Table 4-1, and the function definitions used in the formal semantics are listed in Table 4-3.
The following are definitions required to construct the device context, permitting a direct and unambiguous description of state changes during C3L program execution (i.e. in response to the occurrence of events).

**Definition 1 (Q).** A queue, Q, is a string of objects of the form \( q_1q_2 \ldots q_n \). \( Q = qQ' \) implies that \( q \) is the first (head) element of the queue and \( Q' \) contains the remaining elements of the queue \( q_2q_3 \ldots q_n \). Similarly, \( Qq = Q' \) implies that \( q \) is the last element of the queue \( Q' = q_1q_2 \ldots q_nq \).

### Table 4-1: Symbols Used in C3L Semantics

- \( \rightarrow \) \( \equiv \) Evaluate statements in a context into a new context.
- \( \rightarrow d \) \( _{va} \) \( \equiv \) Evaluates a function definition in a context into a value and a new context.
- \( \downarrow \) \( \equiv \) Evaluates expressions in a context into a value. Indicates that the evaluation of an expression in a context diverges. The result is explicitly undefined and is represented by the ‘undefined’ value.
- \( \uparrow \) \( \equiv \) Raises an instance of an input event to a device context.
- \( \leftarrow \) \( \equiv \) Evaluates an input event raised in a context into a new context.
Table 4-3: Function Definitions Used in C3L Semantics

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toLower</td>
<td>Takes a value of type string and returns the equivalent string in lower case.</td>
</tr>
<tr>
<td>strcat</td>
<td>Takes two value of type string and returns a string whose value is the concatenation of the two argument strings.</td>
</tr>
<tr>
<td>$\Gamma$</td>
<td>Takes a variable name $x$ and returns the type of the input object.</td>
</tr>
<tr>
<td>precondition</td>
<td>Takes as input the definition of a plant event and value/parameter message pairs, and returns a Boolean expression $b$ corresponding to the event’s precondition as defined in the plant event header. If there is no precondition defined in the plant event header, the precondition function evaluates to $\text{true}$.</td>
</tr>
<tr>
<td>auth</td>
<td>Takes as input the definition of a control event and returns a Set expression $t$ corresponding to the event’s authorization as defined in the control event header. If there is no authorization specified in the control event header, the auth function evaluates to $\text{me}$.</td>
</tr>
</tbody>
</table>

**Definition 2 ($\psi$).** A device context (i.e. state) is a 3-tuple, $\psi = (Q, P, M)$, where:

- $Q$ is the input event queue. Each object in the input queue consists of pairs of input event identifiers and ordered value lists. The list of ordered values are the message parameters for the particular event; C3L control events handlers permit the specification of zero or more message parameters, but the list of values always consists of at least one message parameter – a hidden parameter indicating the device from which the event originated which is used to define the value of the ‘you’ set.

- $P$ is the set of plant parameters.

- $M$ is the union of the set of memory variables and the set of local variables declared in the current event handler context.
By definition, the device plant state is completely described by \( \psi_p \). In the scope of plant event handlers, the device context is limited to only the plant state, i.e. \( \psi = \psi_p \).

The initial device context for all C3L device definitions is fixed as follows:

The initial input queue, \( I_0 \), is empty (\( I_0 \equiv \emptyset \)).

The initial state of the set of plant parameters, \( P_0 \), is defined directly by the
C3L plant declarations. For each parameter declaration, the initial state for that variable is indicated by the initialization specified in the declaration. If no initializer is specified, the default value is 0 for parameters of type integer or real, false for parameters of Boolean type, null for parameters of type string, and empty for parameters of type set. The plant parameter always consists of at least one variable – the \textbf{me} set. This variable is a single-element set uniquely identifying the device context and is treated as a constant variable.

The initial state of the set of plant parameters, \( M_0 \), is defined directly by the
C3L controller declarations. For each memory declaration, the initial state for that variable is indicated by the initialization specified in the declaration. If no initializer is specified, the default value is 0 for parameters of type integer or real, false for parameters of Boolean type, null for parameters of type string, and empty for parameters of type set.

The complete initial device context is defined as \( \psi_0 \equiv (\emptyset, P_0, M_0) \).
The device context is a set of functions, $\psi : L \rightarrow V$ mapping locations to numbers. Therefore, $\psi(x)$ denotes the value of location $x$ in the state $\psi$.

Note that each device in a C3L program has its own device context. Since a C3L program is in itself a set of device definitions, C3L programs will generally consist of more than one device, and therefore, a set of device contexts. The semantics of C3L with respect to the operation of a single device will of course be defined in terms of its device context and is independent of other devices in the program. However, when it is necessary to uniquely identify a particular device context within the set of device contexts, a superscript notation will be used

**Operational Semantics of C3L**

With the foundation for the semantic description complete, the higher-level functionality of the language may now be described. This complete description will begin with lower-level functionality and build up to the higher-level functionality.

**Evaluation of Values and Variables**

Let $\langle n, \psi \rangle$ and $\langle x, \psi \rangle$ denote the value $n$ and the location $x$, respectively, waiting to be evaluated in the device context $\psi$. This notation is called the expression-evaluation configuration. Evaluation of values in a state and the evaluation of variables in a state are then denoted by the following two axioms:
Thus, any value is already evaluated as itself and variables evaluate to their
contents in the state. For convenience, C3L defines the constant variables pi and e:

\[
\langle \text{pi, } \psi \rangle \downarrow n (n = \pi \approx 3.14159265358979323)
\]

\[
\langle \text{e, } \psi \rangle \downarrow n (n \approx 2.71828182845904523536)
\]

Additionally, the C3L set variables empty and all have pre-defined values:

\[
\langle \text{empty, } \psi \rangle \downarrow n (n = \emptyset)
\]

\[
\langle \text{all, } \psi \rangle \downarrow n (\forall t \in S_e \land |t| = 1. n = \bigcup t)
\]

Changes to the device state involve the replacement of the current value in one or
more of the variables by which the device state is defined. Let \[ \psi [v/x] \] denote the state
obtained from \[ \psi \] by replacing the contents of location x by value v. One may then write
the following:

\[
\psi [v/x] (y) = \begin{cases} v & x = y \\ \psi (y) & x \neq y \end{cases}
\]

This notation simply states that the location x in the state has been updated with
the value v while the other locations in the state have been left unchanged.
Evaluation of Expressions

C3L supports two types of arithmetic expression – integer and real. C3L semantics allow for automatic conversion of integer to real types through this semantic rule:

\[
\begin{align*}
\langle a, \psi \rangle \downarrow n_0 \quad & (n_0 = n_i \land \Gamma(a) = \mathbb{Z} \land n_0 \in \mathbb{Z} \land n_i \in \mathbb{R}) \\
\langle a, \psi \rangle \downarrow n_1
\end{align*}
\]

Unary Operations

The following four semantic rules describe the behavior of C3L’s three unary operators:

\[
\begin{align*}
\langle a, \psi \rangle \downarrow n & \quad \langle +a, \psi \rangle \downarrow n \\
\langle a, \psi \rangle \downarrow n_0 & \quad \langle -a, \psi \rangle \downarrow n_1 (n_i = -n_0) \\
\langle b, \psi \rangle \downarrow n_0 & \quad \langle \text{not } b, \psi \rangle \downarrow n_1 (n_i = -n_0) \\
\langle t, \psi \rangle \downarrow n_0 & \quad \langle -t, \psi \rangle \downarrow n_1 (n_i = n_0^c)
\end{align*}
\]

Non-relational Binary Operations

The operations of the six non-relational binary operators defined in C3L are described in twelve semantic rules. The following three rules describe the basic arithmetic semantics of binary +, binary -, and binary *:

\[
\begin{align*}
\langle a, \psi \rangle \downarrow n & \quad \langle +a, \psi \rangle \downarrow n \\
\langle a, \psi \rangle \downarrow n_0 & \quad \langle -a, \psi \rangle \downarrow n_1 (n_i = -n_0) \\
\langle b, \psi \rangle \downarrow n_0 & \quad \langle \text{not } b, \psi \rangle \downarrow n_1 (n_i = -n_0) \\
\langle t, \psi \rangle \downarrow n_0 & \quad \langle -t, \psi \rangle \downarrow n_1 (n_i = n_0^c)
\end{align*}
\]

\[
\begin{align*}
\langle a, \psi \rangle \downarrow n & \quad \langle +a, \psi \rangle \downarrow n \\
\langle a, \psi \rangle \downarrow n_0 & \quad \langle -a, \psi \rangle \downarrow n_1 (n_i = -n_0) \\
\langle b, \psi \rangle \downarrow n_0 & \quad \langle \text{not } b, \psi \rangle \downarrow n_1 (n_i = -n_0) \\
\langle t, \psi \rangle \downarrow n_0 & \quad \langle -t, \psi \rangle \downarrow n_1 (n_i = n_0^c)
\end{align*}
\]
Binary / requires three semantic rules to describe its operation over the set of possible arithmetic expressions, specifically to define the behavior when dividing by zero. Simply put, division by zero of integer arguments is undefined (i.e. run-time errors), while division by zero of real arguments evaluates to an undefined real value. The rules defining these semantic rules:

\[
\frac{a_0 \times a_1 \downarrow n \downarrow \downarrow n_0 \downarrow \downarrow \downarrow n_1 \quad (n = n_0 + n_1)}{a_0 \downarrow \downarrow \downarrow a_1 \downarrow \downarrow \downarrow n \quad (n = n_0 - n_1)}
\]

\[
\frac{a_0 \downarrow \downarrow \downarrow a_1 \downarrow \downarrow \downarrow n \quad (n = n_0 \times n_1)}{a_0 \downarrow \downarrow \downarrow \downarrow a_1 \downarrow \downarrow \downarrow n_0 \downarrow \downarrow \downarrow n_1 \quad (n = n_0 / n_1)}
\]

Binary + has a second distinct semantic meaning for arguments of string type. Addition of string arguments is defined as string concatenation.

\[
\frac{r_0 \downarrow \downarrow \downarrow r_1 \downarrow \downarrow \downarrow v \quad (v = strcat(v_0, v_1))}{r_0 + r_1 \downarrow \downarrow \downarrow v \quad (v = strcat(v_0, v_1))}
\]

Furthermore, binary + has a third distinct semantic meaning, set union, for arguments of set type. For convenience, a conservative extension is defined for the binary operator union. These semantic rules are defined as follows:
Binary – has a second, distinct semantic meaning for set arguments. Subtraction of set arguments is defined as set difference.

\[
\frac{\langle t_0, \psi \rangle \downarrow v_0 \quad \langle t_1, \psi \rangle \downarrow v_1}{\langle t_0 + t_1, \psi \rangle \downarrow v} (v = \{x : x \in v_0 \lor x \in v_1\})
\]

Finally, an additional semantic rule defining the meaning of the binary * operator on arguments of set type is needed to complete the binary operator semantics. As was the case for binary – on set arguments, a conservative extension defining the binary operator intersect is also defined. These final two semantic rules for binary operations are defined as follows:

\[
\frac{\langle t_0, \psi \rangle \downarrow v_0 \quad \langle t_1, \psi \rangle \downarrow v_1}{\langle t_0 - t_1, \psi \rangle \downarrow v} (v = \{x : x \in v_0 \land x \not\in v_1\})
\]

\[
\frac{\langle t_0, \psi \rangle \downarrow v_0 \quad \langle t_1, \psi \rangle \downarrow v_1}{\langle t_0 \times t_1, \psi \rangle \downarrow v} (v = \{x : x \in v_0 \land x \in v_1\})
\]

\[
\frac{\langle t_0, \psi \rangle \downarrow v_0 \quad \langle t_1, \psi \rangle \downarrow v_1}{\langle t_0 \text{ intersect } t_1, \psi \rangle \downarrow v}
\]

*Relational Operations*

The semantics of the relational equality operator, ==, and its complement operator, !=, are defined by the following two semantic rules for all types except string:
\[
\begin{align*}
\langle h_0, \psi \rangle \Downarrow v_0 & \quad \langle h_1, \psi \rangle \Downarrow v_1 \\
\left\{ v = \begin{array}{ll} 
\text{true} & \text{iff } v_0 = v_1 \\
\text{false} & \text{otherwise}
\end{array} \right. \\
\langle h_0 \neq h_1, \psi \rangle \Downarrow v & \\
\left\{ v = \begin{array}{ll} 
\text{false} & \text{iff } v_0 = v_1 \\
\text{true} & \text{otherwise}
\end{array} \right.
\end{align*}
\]

For string types, these two semantic rules apply:

\[
\begin{align*}
\langle r_0, \psi \rangle \Downarrow v_0 & \quad \langle r_1, \psi \rangle \Downarrow v_1 \\
\left\{ v = \begin{array}{ll} 
\text{true} & \text{iff } \text{toLower}(v_0) = \text{toLower}(v_1) \\
\text{false} & \text{otherwise}
\end{array} \right. \\
\langle r_0 \neq r_1, \psi \rangle \Downarrow v & \\
\left\{ v = \begin{array}{ll} 
\text{false} & \text{iff } \text{toLower}(v_0) = \text{toLower}(v_1) \\
\text{true} & \text{otherwise}
\end{array} \right.
\end{align*}
\]

The relational inequality operators are defined over the set of arithmetic expressions by the following semantic rules:

\[
\begin{align*}
\langle a_0, \psi \rangle \Downarrow n_0 & \quad \langle a_1, \psi \rangle \Downarrow n_1 \\
\left\{ n = \begin{array}{ll} 
\text{true} & \text{iff } a_0 < a_1 \\
\text{false} & \text{otherwise}
\end{array} \right. \\
\langle a_0 \leq a_1, \psi \rangle \Downarrow n & \\
\left\{ n = \begin{array}{ll} 
\text{true} & \text{iff } a_0 \leq a_1 \\
\text{false} & \text{otherwise}
\end{array} \right. \\
\langle a_0 > a_1, \psi \rangle \Downarrow n & \\
\left\{ n = \begin{array}{ll} 
\text{true} & \text{iff } a_0 > a_1 \\
\text{false} & \text{otherwise}
\end{array} \right. \\
\langle a_0 \geq a_1, \psi \rangle \Downarrow n & \\
\left\{ n = \begin{array}{ll} 
\text{true} & \text{iff } a_0 \geq a_1 \\
\text{false} & \text{otherwise}
\end{array} \right.
\end{align*}
\]

Over the set of string expressions, the following rules define the semantics of the inequality operators:

\[
\begin{align*}
\langle r_0, \psi \rangle \Downarrow n_0 & \quad \langle r_1, \psi \rangle \Downarrow n_1 \\
\left\{ n = \begin{array}{ll} 
\text{true} & \text{iff } \text{toLower}(n_0) \text{ alphabetically preceeds toLower}(n_1) \\
\text{false} & \text{otherwise}
\end{array} \right.
\end{align*}
\]
\[ \langle r_0, \psi \rangle \downarrow n_0, \langle r_1, \psi \rangle \downarrow n_1 \left( n = \begin{cases} \text{true} & \text{iff toLower}(n_1) \text{ alphabetically precedes toLower}(n_0) \\ \text{false} & \text{otherwise} \end{cases} \right) \]

\[ \langle r_0 > r_1, \psi \rangle \downarrow n \left( n = \begin{cases} \text{false} & \text{iff toLower}(n_1) \text{ alphabetically precedes toLower}(n_0) \\ \text{true} & \text{otherwise} \end{cases} \right) \]

\[ \langle r_0 <= r_1, \psi \rangle \downarrow n \left( n = \begin{cases} \text{false} & \text{iff toLower}(n_0) \text{ alphabetically precedes toLower}(n_1) \\ \text{true} & \text{otherwise} \end{cases} \right) \]

Finally, over the set of set expressions the following semantics are defined. These semantics include the definition of conservative extensions for the binary set operators 

\text{subsetOf} (\leq), \text{properSubsetOf} (<), \text{supersetOf} (\geq), \text{properSupersetOf}(>), \text{and in} (\subseteq).

\[ \langle t_0, \psi \rangle \downarrow n_0, \langle t_1, \psi \rangle \downarrow n_1 \left( n = \begin{cases} \text{true} & \text{iff } n_0 \subseteq n_1 \\ \text{false} & \text{otherwise} \end{cases} \right) \]

\[ \langle t_0 <= t_1, \psi \rangle \downarrow n \left( t_0 \text{ subsetOf } t_1, \psi \right) \downarrow n \]

\[ \langle t_0 \text{ subsetOf } t_1, \psi \rangle \downarrow n \left( t_0 \text{ in } t_1, \psi \right) \downarrow n \]

\[ \langle t_0, \psi \rangle \downarrow n_0, \langle t_1, \psi \rangle \downarrow n_1 \left( n = \begin{cases} \text{true} & \text{iff } n_0 \subset n_1 \\ \text{false} & \text{otherwise} \end{cases} \right) \]

\[ \langle t_0 < t_1, \psi \rangle \downarrow n \left( t_0 \text{ properSubsetOf } t_1, \psi \right) \downarrow n \]

\[ \langle t_0 < t_1, \psi \rangle \downarrow n \left( t_0 \geq t_1, \psi \right) \downarrow n \left( n = \begin{cases} \text{true} & \text{iff } n_0 \supseteq n_1 \\ \text{false} & \text{otherwise} \end{cases} \right) \]
\[
\frac{\langle t_0 \geq t_1, \psi \rangle \downarrow n}{\langle t_0 \supseteq \ t_1, \psi \rangle \downarrow n}
\]

\[
\frac{\langle t_0, \psi \rangle \downarrow n \quad \langle t_1, \psi \rangle \downarrow n}{\langle t_0 > t_1, \psi \rangle \downarrow n\left(n = \begin{array}{ll}
true & \text{iff } n_0 \succ n_1 \\
false & \text{otherwise}
\end{array}\right)}
\]

\[
\frac{\langle t_0 > t_1, \psi \rangle \downarrow n}{\langle t_0 \ 	ext{properSupersetOf} \ t_1, \psi \rangle \downarrow n}
\]

**Binary Logical Operations**

The logical **and** operator and the logical **or** operator are defined in C3L with syntax-defined precedence of **and** over **or** and with left-associativity. C3L semantics specify short-circuit evaluation of these two logical operations as follows:

\[
\frac{\langle b_0, \psi \rangle \downarrow false}{\langle b_0 \ 	ext{and} \ b_1, \psi \rangle \downarrow false}
\]

\[
\frac{\langle b_0, \psi \rangle \downarrow true \quad \langle b_1, \psi \rangle \downarrow true}{\langle b_0 \ 	ext{and} \ b_1, \psi \rangle \downarrow true}
\]

\[
\frac{\langle b_0, \psi \rangle \downarrow true}{\langle b_0 \ 	ext{or} \ b_1, \psi \rangle \downarrow true}
\]

**Equivalence of Expressions**

There is a natural equivalence relation on expressions in this operational semantic. It is useful to treat two expressions as equivalent if for all states, the expressions evaluate to the same value.
Define:
\[ e_0 \sim e_1 \iff \forall v \in V \forall \psi \in \Psi. \langle e_0, \psi \rangle \downarrow v \Leftrightarrow \langle e_1, \psi \rangle \downarrow v \]

**Constant Expressions**

The set of constant expressions, \(O\), is defined as the subset of expressions that, for all possible values in all possible states, the expression evaluates to the same value. Let

\[ O = \{ e : e \in E \land \forall v \in V \forall \psi \in \Psi. \langle e, \psi \rangle \downarrow v \} \]

By previous definition, the constant values \(pi, e, all\) and \(empty\) are defined to always evaluate to the same value. The set variable \(me\) can also be evaluated as a location in a state, however the value of the \(me\) set must be constant in a particular device context. Let:

\[
\begin{aligned}
&\langle me, \psi^s \rangle \downarrow t_g \left( t_g \neq t_{g_i}, \forall g, g_1 \in G \land g \neq g_1 \land \lvert t_g \rvert = 1 \right) \\
&\forall t \in S \forall \psi^s \in \Psi^s
\end{aligned}
\]

This axiom states that the value of \(me\) is both constant in its device context, consists of a set with exactly one member, and is uniquely valued from that of the \(me\) set in any other device context.

The set of constant expressions in C3L always consists of at least five members:

\[ O \supseteq \{ pi, e, all, empty, me \} \]
**Execution of Statements**

The statement (command) configuration denoting a statement $s$ waiting to be executed in state $\psi$ is represented by the pair $\langle s, \psi \rangle$. The complete execution of a statement $s$ in state $\psi$ and terminating in a new state $\psi'$ is denoted $\langle s, \psi \rangle \rightarrow \psi'$.

C3L statements are executed sequentially as described in the following semantic rule:

$$
\langle s_1, \psi \rangle \rightarrow \psi'' \quad \langle s_2, \psi'' \rangle \rightarrow \psi' \\
\langle s_1, s_2, \psi \rangle \rightarrow \psi'
$$

Block statements are defined in the C3L syntax, and since any arbitrary-length sequence of statements may be sequenced by repeated application of the sequence-rule, the semantics need only define that the execution of statements in a block is equivalent to the execution of the same statements outside a block:

$$
\langle s, \psi \rangle \rightarrow \psi' \\
\langle \{s\}, \psi \rangle \rightarrow \psi'
$$

**Assignment Statements**

The syntax of C3L assignment statements defines daisy-chaining through right associativity; the semantics of assignment statements in C3L is defined by the following rules:

$$
\langle e, \psi \rangle \downarrow v \\
\langle x = e, \psi \rangle \rightarrow \psi'[v/x] \quad (v \neq e)
$$

$$
\langle e, \psi \rangle \downarrow v \quad \langle x = e, \psi \rangle \rightarrow \psi' \\
\langle x = e, \psi \rangle \downarrow v
$$
**Selection Statements**

Selection in C3L is provided by the ubiquitous if-else statement. The else clause is optional. The formal semantic of this class of statements is defined as follows:

\[
\langle b, \psi \rangle \downarrow \text{false} \quad \langle b, \psi \rangle \downarrow \text{true}
\]

\[
\langle \text{if } (b) \ s, \psi \rangle \rightarrow \psi
\]

\[
\langle s, \psi \rangle \rightarrow \psi' \quad \langle s, \psi \rangle \rightarrow \psi
\]

**Repetition Statements**

C3L repetition statements include “for” and “while” loops. The somewhat intricate semantics of the “for” loop are as follows:

\[
\langle o_0, \psi \rangle \downarrow n_0 \quad \langle o_1, \psi \rangle \downarrow n_1
\]

\[
\langle \text{for } (x = o_0 \text{ to } o_1) \ s, \psi \rangle \rightarrow \psi \left[ n_0 / x \right] (n_0 > n_1)
\]

\[
\langle o_0, \psi' \rangle \downarrow n_0
\]

\[
\langle o_1, \psi' \rangle \downarrow n_1
\]

\[
\langle s, \psi \left[ n_0 / x \right] \rangle \rightarrow \psi''
\]

\[
\langle \text{metafor } x \ n_1 \ s, \psi^{\ast} \left( (\psi^{\ast}(x) + 1) / x \right) \rangle \rightarrow \psi' \quad (n_0 \leq n_1 \wedge \psi(x) = \psi^{\ast}(x))
\]

\[
\langle x, \psi \rightarrow \psi(x) \quad \langle n_1, \psi \rangle \downarrow n_1
\]

\[
\langle \text{metafor } x \ n_1 \ s, \psi \rangle \rightarrow \psi' \quad (\psi(x) > n_1)
\]
\[ \langle x, \psi \rangle \rightarrow \psi (x) \]
\[ \langle n_1, \psi \rangle \downarrow n_1 \]
\[ \langle s, \psi \rangle \rightarrow \psi' \]
\[ \begin{array}{c}
\langle \text{metafor } x \ n_1 \ s, \psi'\left[\left(\psi''(x) + 1\right)/x\right]\rangle \rightarrow \psi' \\\n\langle \text{metafor } x \ n_1 \ s, \psi \rangle \rightarrow \psi' \end{array} \]
\[ \left(\psi (x) \leq n_1 \land \psi (x) = \psi''(x)\right) \]
\[ \langle o_0, \psi \rangle \downarrow n_0 \]
\[ \langle o_1, \psi \rangle \downarrow n_1 \]
\[ \langle o_2, \psi \rangle \downarrow n_2 \]
\[ \langle s, \psi \left[n_0/x\right]\rangle \rightarrow \psi'' \]
\[ \begin{array}{c}
\langle \text{metafor } x \ n_1 \ n_2 \ s, \psi''\left[\left(\psi''(x) + n_2\right)/x\right]\rangle \rightarrow \psi' \\\n\langle \text{for } (x = o_0 \ to \ o_1 \ by \ o_2) \ s, \psi \rangle \rightarrow \psi' \end{array} \]
\[ \left(\begin{array}{c}
(n_0 \leq n_1 \land n_2 > 0 \lor n_0 \geq n_1 \land n_2 < 0) \\\n\psi (x) = \psi''(x) \end{array}\right) \]
\[ \langle s, \psi \left[n_0/x\right]\rangle \rightarrow \psi'' \]
\[ \begin{array}{c}
\langle \text{metafor } x \ n_1 \ n_2 \ s, \psi''\left[\left(\psi''(x) + n_2\right)/x\right]\rangle \rightarrow \psi' \\\n\langle \text{for } (x = o_0 \ to \ o_1 \ by \ o_2) \ s, \psi \rangle \rightarrow \psi' \end{array} \]
\[ \left(\begin{array}{c}
(n_0 \leq n_1 \land n_2 > 0 \lor n_0 \geq n_1 \land n_2 < 0) \\\n\psi (x) = \psi''(x) \end{array}\right) \]
\[ \langle x, \psi \rangle \rightarrow \psi (x) \]
\[ \langle n_1, \psi \rangle \downarrow n_1 \]
\[ \langle n_2, \psi \rangle \downarrow n_2 \]
\[ \langle s, \psi \rangle \rightarrow \psi'' \]
\[ \begin{array}{c}
\langle \text{metafor } x \ n_1 \ n_2 \ s, \psi''\left[\left(\psi''(x) + n_2\right)/x\right]\rangle \rightarrow \psi' \\\n\langle \text{for } (x = o_0 \ to \ o_1 \ by \ o_2) \ s, \psi \rangle \rightarrow \psi' \end{array} \]
\[ \left(\begin{array}{c}
(n_0 \leq n_1 \land n_2 > 0 \lor n_0 \geq n_1 \land n_2 < 0) \\\n\psi (x) = \psi''(x) \end{array}\right) \]

The first four semantic rules govern the behavior of “for” loops when the default increment of one is implied, and the second four rules govern the behavior of “for” loops.
with program-defined increment. The first two rules in each set of four rules cover the initialization of the “for” loop environment when the loop iterates and when it doesn’t. When the loop does iterate at least once, the ‘metafor’ rules are utilized. These semantic rules cannot be generated directly in C3L syntax; there purpose is solely to govern the repetition of for loops beyond the initialization of the loop environment.

Notice that these semantics require that the “for” loops all terminate in finite time through the requirement of constant expressions and through the requirement that the statements comprising the body of the loop may NOT alter the contents of the loop control variable.

The semantics of the “while” loop are defined as follows:

\[
\begin{align*}
\langle b, \psi \rangle \downarrow \text{false} & \quad \langle \text{while } (b, o) s, \psi \rangle \rightarrow \psi \\
\langle b, \psi \rangle \downarrow \text{true} & \quad \langle a, \psi \rangle \downarrow n & \quad (n < 1) \\
\langle b, \psi \rangle \downarrow \text{true} & \quad \langle o, \psi \rangle \downarrow n & \quad \langle \text{while } (b, o) s, \psi \rangle \rightarrow \psi \\
\langle b, \psi \rangle \downarrow \text{true} & \quad \langle o, \psi \rangle \downarrow n & \quad \langle s, \psi \rangle \rightarrow \Psi^* & \quad \langle \text{while } (b, n-1) s, \psi^* \rangle \rightarrow \Psi' & \quad (n \geq 1)
\end{align*}
\]

Again, to ensure that all loops terminate in finite time, the “while” loop syntax includes a maximum iteration count expressed as a constant expression. The “while” loop is only permitted to execute while the iteration count is strictly positive.
Local Variable Declarations

In addition to the global variables declared as part of the plant (the device state-space) and of the controller (memory variables), C3L permits the declaration of local variables in control event handlers. These local variables extend the device context over the lifetime of the event execution. Let $\psi \{v/d\}$ denote extending the environment with a new location $d \not\in L$ and initializing this location with value $v$. Therefore,

$$\psi \{v/d\}(x) = \begin{cases} 
    v & x = d \\
    \psi(x) & x \neq d
\end{cases}$$

The semantics for local variable declarations is defined as follows:

$$\langle e, \psi \rangle \downarrow v \quad \frac{\langle y \ d = e, \psi \rangle \rightarrow \psi \{v/d\} (\Gamma(d) = \Gamma(y) = v)}{\langle \text{integer} \ d, \psi \rangle \rightarrow \psi \{0/d\} (\Gamma(d) = \text{integer})}$$

$$\langle \text{real} \ d, \psi \rangle \rightarrow \psi \{0.0/d\} (\Gamma(d) = \text{real})$$

$$\langle \text{boolean} \ d, \psi \rangle \rightarrow \psi \{\text{false}/d\} (\Gamma(d) = \text{boolean})$$

$$\langle \text{string} \ d, \psi \rangle \rightarrow \psi \{""/d\} (\Gamma(d) = \text{string})$$

$$\langle \text{set} \ d, \psi \rangle \rightarrow \psi \{\emptyset/d\} (\Gamma(d) = \text{set})$$
Calling Functions

All functions and C3L events (structurally similarly to function declarations) have the property of mapping zero or more input parameters in a device state into zero or one output parameters and a (potentially) new device state. This mapping is defined through the function definition. Assume the declaration of a definition

\[ f(x_1, \ldots, x_n) = \text{definition}_{f} \quad \forall f \in F \]

Since calling a function both evaluates to a value for non-void returning functions and potentially alters the device state, a distinct notation is required to describe this situation. The \( \rightarrow_{va}^{d} \) operator defined in Table 4-1 denotes the evaluation of a function definition in a device state into a value and new device state using pass-by-value semantics. Let \( \langle v, \psi' \rangle \) denote the complete execution of a function definition. Formally:

\[ \langle \text{definition}_{f} \left[ v_i / x_1, \ldots, v_{n_f} / x_{n_f} \right], \psi \rangle \rightarrow_{va}^{d} \langle v, \psi' \rangle \]

One may then define the semantic rule defining the execution of a function as follows:

\[ \langle e_i, \psi \rangle \downarrow v_i \]

\[ \vdots \]

\[ \langle e_{n_f}, \psi \rangle \downarrow v_{n_f} \]

\[ \langle \text{definition}_{f} \left[ v_i / x_1, \ldots, v_{n_f} / x_{n_f} \right], \psi \rangle \rightarrow_{va}^{d} \langle v, \psi' \rangle \]

\[ \langle f \left( e_1, \ldots, e_{n_f} \right), \psi \rangle \rightarrow_{va}^{d} \langle v, \psi' \rangle \]

Note that this rule is limited to non-controllable plant event handler functions; the semantics of these event handlers is very similar but require an additional antecedent to
describe the behavior of the event’s precondition. If the function is a void-returning function, then \( v = \epsilon \), the empty object. This rule describes the dual nature of a function call in that it both evaluates to a value and can change the device state. The dual conclusions one may reach from executing a function may be denoted as

\[
\left\langle f\left(e_1, \ldots, e_{n_f}\right), \psi \right\rangle \rightarrow_{\text{def}} \left\langle v, \psi' \right\rangle
\]

\[
\left\langle f\left(e_1, \ldots, e_{n_f}\right), \psi \right\rangle \downarrow v \left\langle f\left(e_1, \ldots, e_{n_f}\right), \psi \right\rangle \rightarrow \psi'
\]

Note that the semantics of the assignment operation preclude an empty object from appearing on the right hand side of the assignment operator; consequently void-returning functions may not appear on the right-hand side of assignment operators.

**Invocation of Controllable Plant Events**

As described in chapter 3, control event handlers may invoke controllable events by name and may optionally include message parameters. Remembering that the function precondition() as defined in Table 4-3 maps a controllable plant event to a Boolean expression representing the precondition defined for that controllable plant event, the semantic rule describing the invocation of a controllable plant event is as follows:

\[
\left\langle \text{precondition}\left(\text{definition}, \psi \right) \downarrow \text{false} \right\rangle
\]

\[
\left\langle c, \psi \right\rangle \rightarrow \psi
\]

Simply put, invocation of a controllable plant event from a state in which the precondition for that event is false does not affect the device state. Similarly, a controllable event with message parameters (i.e. arguments) also has no effect when the precondition evaluates to false:
When the precondition for a controllable plant event evaluates to true in the device state, the plant state will make a transition. The precise transition that will occur is defined in the plant event handler definition, similarly to the way general functions may alter the device state. The rule for describing this behavior may be written

\[
\langle e_1, \psi \rangle \downarrow v_1 \\
\vdots \\
\langle e_n, \psi \rangle \downarrow v_n \\
\left\langle \text{precondition} \left( \text{definition}_{c \left[ v_i / x_1, \ldots, v_n / x_n \right]} \right), \psi \right\rangle \downarrow \text{true}
\]

\[
\langle \text{definition}_{c \left[ v_1 / x_1, \ldots, v_n / x_n \right]}, \psi \rangle \rightarrow^d \langle e, \psi' \rangle \\
\langle c \left( e_1, \ldots, e_n \right), \psi \rangle \rightarrow \psi' \\
(\forall x \in M. \psi(x) = \psi'(x))
\]

Note that the change in device state is restricted to changes in plant parameters as no location in the set of memory variables is permitted to change values.

**Semantics of Events**

To this point in the operational semantics of C3L, all operations have been completely defined with respect to a single device context. For the first time in the operational semantics of C3L, a notation is required to express the interaction of devices within the C3L environment as raising events to devices will affect those device states.
The \( \uparrow \) operator defined in Table 4-1 denotes the operation of raising an input event to a device context. Let \( i \uparrow \psi \) denote raising an input event \( i \) in the context \( \psi \) of device \( g \). Furthermore, let the \( \mapsto \) defined in Table 4-1 denote the evaluation of an event raised in a device context into a new context.

Since input events may either be control events or uncontrollable plant events, and since the semantics for these two sub classifications of input events are different, the following discussion will focus first on the semantics of control events.

**Occurrence of Control Events**

Recalling that the device context \( \psi = \langle Q, P, M \rangle \), the following semantic rules define how the device context may change when control events occur in C3L.

\[
j \uparrow \langle Q, P, M \rangle \mapsto \langle Q, P, M \rangle
\]

When message parameters are included with the control event, the values of those parameters are appended to the event in the notation as follows:

\[
j(\langle v_1, \ldots, v_n \rangle) \uparrow \langle Q, P, M \rangle \mapsto \langle Q, P, M \cup \langle v_1, \ldots, v_n \rangle \rangle
\]

Let the notation \( \psi \{ i \} \) denote extending the environment of the current state to yield a new state obtained from appending the input event \( i \) to the end of the input event queue in state \( \psi \). Using this notation one may write the previous two semantic rule as the follows equivalents:
These rules indicate that, when a control event is raised in a device context, the device context is altered by adding the input event to the end of the input event queue.

**Raise Statements**

The bulk of the work of communication and high-level control in C3L is accomplished through “raise” statements. Raise statements generate control events by name and may optionally include message parameters and specification of the devices to which the event should be directed. When no direction is provided for the raised event, the default destination of all devices is used.

First, consider the case when a control event is raised with no message parameters and with no direction. The semantic rule for this case is as follows:

\[
\frac{j \uparrow \psi \rightarrow \psi \{ j \}}{j(v_1, \ldots, v_n) \uparrow \psi \rightarrow \psi \{ j(v_1, \ldots, v_n) \}}
\]

Note that the value of the ‘me’ set is automatically attached to the event as a message parameter. Consequently, there is no way a malicious device can ‘masquerade’ as another device, and a device raising an event is always uniquely identified. Since there is no antecedent, this rule is an axiom and yields a fundamental result; for all devices to
which the event is directed, the event is added to all the input event queues of those devices.

When a direct argument is specified, the behavior is similar. Two semantic rules are required, one for the case when the raising device is also a target for the event and one for the case when it is not.

\[
\begin{align*}
&\frac{\langle t, \psi \rangle \downarrow \nu \quad \left(\begin{array}{c}
me \in \nu \\
\forall g \in (\nu - me)
\end{array}\right)}{
\langle \text{raise } j \text{ direct } t, \psi \rangle \rightarrow \psi \left\{ j(\psi(\nu)) \right\} \quad \psi \notin \left\{ j(\psi(\nu)) \right\} \quad \forall g \in (\nu - me)}
\end{align*}
\]

When message parameters are specified in a raise statement, the behaviors are again similar, with requisite changed needed to denote the evaluation of the expressions constituting the message parameters. These rules are as follows:

\[
\begin{align*}
&\frac{\langle e_1, \psi \rangle \downarrow \nu_1 \\
&\quad \vdots \\
&\quad \langle \text{raise } j(e_1, \ldots, e_n, \psi) \rangle \rightarrow \psi \left\{ j(\psi(e_1), \ldots, \psi(e_n)) \right\} \quad \psi \notin \left\{ j(\psi(e_1), \ldots, \psi(e_n)) \right\} \quad (\forall g \in (G - me))
\end{align*}
\]
Equivalence of Statements in C3L

There is a natural equivalence relation on statements in this operational semantic. It is useful to treat two statements as equivalent if for all states, executing the statements in a state evaluate to the same new state.

Define:

\[ s_0 \sim s_1 \text{ iff } \forall s \in S. \forall \psi, \psi' \in \Psi. S_0, \psi \rightarrow \psi' \iff S_1, \psi \rightarrow \psi' \]
Occurrence of Uncontrollable Plant Events

Previously, the occurrence of control events was described as a set of semantic rules. These control events occurred as the direct results of the execution of raise statements in some device in a C3L program, and depending on how the events were directed would affect the device contexts of zero or more devices. However, the only effect of these actions on those device contexts was adding the event to the device’s input event queues. Uncontrollable plant events are more complex for several reasons. First, uncontrollable plant events occur outside the context of the program, so no operational mechanism is needed to describe their invocation. Second, uncontrollable plant events may directly affect the device’s plant state. Third, uncontrollable plant events may have a precondition specified as a Boolean expression. If this expression evaluates to true in the current device’s plant parameter context, the event is processed; if the expression evaluates to false, the event is ignored. Finally, after an uncontrollable plant event is processed and any changes to the plant parameters are made, the uncontrollable plant event is added to the input event queue of the device, but only that device – no other devices can be directly affected by the occurrence of an uncontrollable plant event.

This discussion will begin with the semantic rules describing the behavior of uncontrollable events when its precondition is false (i.e. the event is not permitted).

\[
\begin{align*}
\left( \text{precondition}(\text{definition}_u \left[v_1/x_1, \ldots, v_n/x_n \right], \psi) \downarrow \text{false} \right) \\
\langle \text{definition}_u \rangle \left(t, v_1, v_2, \ldots, v_n \right) \uparrow \psi \mapsto \psi
\end{align*}
\]
When an uncontrollable event occurs when its precondition is true, the definition of the plant event is executed (updating the plant state), then the event is added to the device’s input queue. The rule describing this behavior is as follows:

\[
\begin{align*}
\left( \text{precondition} \left( \text{definition}_u \left[ v_1/x_1, \ldots, v_{n_u}/x_{n_u} \right] \right), \psi \right) & \cup \text{true} \\
\text{definition}_u \left[ v_1/x_1, \ldots, v_{n_u}/x_{n_u} \right], \psi \rightarrow^d \langle \varepsilon, \psi' \rangle \\
\forall x \in M. \psi(x) = \psi'(x) \\
\end{align*}
\]

Note that the semantics enforce that the memory variable component of the state space cannot be altered by the execution of the uncontrollable event handler.

If the uncontrollable event doesn’t have a definition in the device, it is ignored:

\[
\begin{align*}
\left( t, v_1, \ldots, v_{n_u} \right) \uparrow \psi & \mapsto \psi' \left( \neg \exists \text{definition}_u \right) 
\end{align*}
\]

**Semantics of the Event Queue**

To this point in the description of the semantics of C3L, the operations of evaluation of expressions, execution of statements, and occurrence of events has been defined. This leaves only formally describing the removal of input events from the input event queue. Again, recall that the device context \( \psi = \langle Q, P, M \rangle \), where \( Q \) is the input event queue, \( P \) is the set of plant parameters, and \( M \) is the set of memory variables. When there are no statements to execute and when the input event queue is empty, C3L programs do not terminate; instead the device state remains unaltered as defined by this semantic rule:
When the input event queue of a device is not empty, there is at least one event available to be processed. Recall from Definition 1 that \( Q = qQ' \) implies that \( q \) is the first (head) element of the queue and \( Q' \) contains the remaining elements of the queue \( q, q, \ldots, q_n \). For notational convenience, let the device state with a non-empty queue with head of the queue \( q \) be denoted as \( \psi = \langle Q, P, M \rangle = \langle qQ', P, M \rangle = q\psi' \). When there is an input event to dequeue, the required behavior depends on the authorization of the control event. Recalling that the auth function defined in Table 4-3 returns a set expression \( t \) representing the devices authorized to invoke the given control event at the device, one may define the rule governing the removal of an input control event from the head of the queue when it has been determined that the input control event \( i \) was received from an authorized device through the evaluation of the expression “\( \text{you in auth(definition(i))} \)”, where \( \text{you} \) is defined as the first (hidden) message value:

\[
\left\{ t \text{ in auth(definition}_i),\left\{ i(t,v_1,\ldots,v_{n_i})\psi' \right\} \right\} \Downarrow \text{false} \quad \left\{ \varepsilon,\left\{ i(t,v_1,\ldots,v_{n_i})\psi' \right\} \right\} \rightarrow \psi'
\]

In other words, when the input event at the head of the event queue is not from an authorized device as evaluated in the current context, the input event is removed from the queue and the device context is otherwise unchanged.

When the input event is properly authorized, the definition of the input event is executed, thereby altering the device context as denoted in this semantic rule:
As with uncontrollable events, if there is no definition for a particular event, it is discarded:

\[
\langle \epsilon, \{i(t, v_1, \ldots, v_n)\psi^*\} \rangle \rightarrow \psi' (\neg \exists \text{definition})
\]

**Summary of Formal Semantics**

The operational semantics of C3L have been provided in detail such that all rules may be shown to be derivable from the axioms of the semantic system. The semantics describe a C3L program as sequential operations on a set of device contexts where each device context consists of three distinct components – an input event queue, a set of plant parameters and a set of device-global memory variables. The evaluate of expressions and execution of statements is detailed and with the exception of the raise statements affect only a single device context – the context in which the expression is evaluated or the statement is executed.

The semantics of events presented describes the three classes of events in C3L and define both how the occurrence of these events affect the device states as well as how C3L programs may trigger events. The control event semantics provide a mechanism for inter-process communication and are described in a operational semantic in which a distributed system of collaborating devices is treated as a set of device contexts;
consequently communication may be described by defining how execution of raise statements in one context affect other device contexts. This state-centric approach was chosen over a process-centric approach as it better fits the state-transition methodology of the C3L programming paradigm and frees implementations of the language to create processes in practically any manner consistent with the semantics of the language. In other words, the language is oblivious to the threading model used in implementing C3L applications.

**Formal Properties of C3L**

The formal semantics of a programming language provide a mathematically sound foundation upon which the formal properties of the language may be described. Two properties which are important for a language intended for the control of devices were previously shown for this language – prefix-closed controllability and event fairness; these formal results are proven to still hold in the heavily revised version of C3L.

**Prefix-closed Controllability in C3L**

The operational semantics of C3L enable one to prove that any device controlled through C3L is prefix-closed controllable as characterized by two necessary and sufficient conditions:
1. The language specification cannot cause a state transition if the device’s state prohibits it.

2. The language specification cannot prevent an event from occurring if that event is outside of the control of the device.

In C3L, state is defined in the plant parameters, and state transitions are defined in the uncontrollable and controllable event handlers. Preconditions under which state transitions may occur are defined through a Boolean expression on values and variables in the device state space, and the rules under which these event handlers are executed (and thereby state transitions occur) require that the event precondition evaluates to false before the handler is executed. Preconditions may be specified for either controllable or uncontrollable event handlers. Consequently, the operational semantics of C3L guarantee that no state transition can occur if the device’s state prohibits it. This satisfies the first necessary condition for prefix-closed controllability.

C3L semantics define what is required when an uncontrollable event occurs – that is, events outside of the control of the device occur. If the device state permits the event, then the semantics require that the event handler for the uncontrollable event be executed, implementing the corresponding state transition. Consequently, the semantics cannot prevent an event outside its control from occurring, satisfying the second necessary condition for prefix-closed controllability. Therefore, all C3L programs are prefix-closed controllable.
Event Fairness

The formal semantics of C3L provides event fairness by guaranteeing that any device can never be starved of events (leaving a device waiting indefinitely for an event) and that all events raised to a device are handled in finite time. This property is called event fairness and is proved as follows:

**Proposition:** An event i raised to a device is always handled in finite time.

**Proof:** An event raised to a device may be handled in one of two primary ways – either by immediately handling the event by determining that an uncontrollable plant event raised has a false precondition and discarding it or by adding the event to the input queue, then eventually dequeueing the event and executing the control event handler. For an control event raised to a device to be handled, it must first be added to the input queue, then dequeued and handled. Therefore, one must first show that all events raised to a device are added to the input queue unless the event is an uncontrollable plant event with a false precondition, in which case the event is handled by discarding it. There are two classifications of events that can be raised to a device – control events and uncontrollable plant events. Beginning with the former, the axiom governing the behavior when control events are raised is $\mathcal{D} \models \psi \leftrightarrow \psi \{j(me)\}$, which shows that all control events are added to the tail of the input event queue. In the later case, there are two rules governing the required behavior when uncontrollable plant events occur. The first rule

$\langle \text{precondition}(\text{definition}_{u}[v_{1}/x_{1}, \ldots, v_{n_{u}}/x_{n_{u}}]), \psi \rangle \downarrow \text{true}$

$\frac{\langle \text{definition}_{u}[v_{1}/x_{1}, \ldots, v_{n_{u}}/x_{n_{u}}], \psi \rangle \rightarrow^{q_{\omega}} \langle \varepsilon, \psi' \rangle}{u(t, v_{1}, \ldots, v_{n_{u}}) \models \psi \leftrightarrow \psi' \{j(t, v_{1}, \ldots, v_{n_{u}})\}}$ (exists definition $_{u}$)

$\forall x \in M. \psi(x) = \psi'(x)$
shows that uncontrollable plant events whose precondition evaluates to true, in addition to altering the device plant state, are always added to the input event queue. These two rules show that all input events that should be added to the input queue are added to the input queue.

When an uncontrollable plant event is raised to a device when its precondition is false or for which there is no definition, two additional semantic rules applies:

\[
\begin{align*}
\text{precondition}(\text{definition}_u[v_1/x_1, \ldots, v_n/x_n], \psi) \Downarrow \text{false} \\
u(v_1, v_2, \ldots, v_n) \uparrow \psi \mapsto \psi
\end{align*}
\]

\[
\begin{align*}
\text{u}(t, v_1, \ldots, v_n) \uparrow \psi \mapsto \psi
\end{align*}
\]

Therefore, these plant events are handled by discarding them. These four rules govern the behavior for all control events and uncontrollable plant events – for control events the axiom applies to all, and for uncontrollable plant events the precondition must be either true or false, consequently all uncontrollable events are covered by the last three rules. Since input events that may be raised to a device may only be either control events or uncontrollable plant events, and those input events that are not handled immediately are added to the input event queue, one must now show that all events in the queue are dequeued and handled by executing the appropriate control event handler in finite time.

When added to an input event queue, events are always added to the end of the queue. There are three semantic rules that govern the removal of events from the input queue:
\[
\frac{\langle \text{t in auth(definition)}, \{i(t,v_1,\ldots,v_n)\psi'\} \rangle \downarrow \text{false}}{\langle \varepsilon, \{i(t,v_1,\ldots,v_n)\psi'\} \rangle \rightarrow \psi' \quad (\exists\text{definition})}
\]

\[
\frac{\langle \text{t in auth(definition)}, \{i(t,v_1,\ldots,v_n)\psi^*\} \rangle \downarrow \text{true}}{\langle \text{definition}_i[t/\text{you}, v_1/x_1,\ldots,v_n/x_n], \psi^* \rangle \rightarrow^d \langle \varepsilon, \psi' \rangle \quad (\exists\text{definition})}
\]

\[
\frac{\langle \varepsilon, \{i(t,v_1,\ldots,v_n)\psi'\} \rangle \rightarrow \psi' \quad (-\exists\text{definition})}{\}
\]

These three rules govern the dequeueing of all events as the authorization expression of a control event must evaluate to either true or false if a definition exists, and if a definition doesn’t exist the third rule applies. If the authorization expression evaluates to false or when no handling definition for the event exists, the event is removed from the input queue but no further changes to the device state occur. When a handling definition exists and the authorization expression evaluates to true, the input event is removed from the front of the input queue and definition of the control event handler is executed. Note, however, that these three rules apply when there are no statements to execute – in other words, events are dequeued after the previously executing event handlers are completely executed. Input events are only added to the end of the input queue and only removed from the front of the queue. To prove that every event added to the queue is dequeued (and then handled) in finite time, one must first show that all event handlers are completely executed in finite time.

Let \( \varepsilon \) the empty list of statements, and let the partial order relation be defined as the “is a sublist of” relation. The five distinct atomic statements in C3L are exprStmt
(assignment and control event invocation), raiseStmt (inter- and intra-device messaging), selectStmt (if [-else]), repStmt (“for” and “while” loops) and postfix () for function and controllable plant event invocation. None of the first three classes of these statements add to the statement list, nor can any of those classes of statements execute infinitely in normal operations. The fifth of these – function invocation through postfix () – invokes either controllable plant events or implicit functions. Implicit functions are part of the language and have been verified to always execute in finite time. Controllable plant event handlers are finite lists of expression statements that may only be assignment statements over the set of plant parameters and must therefore also execute in finite time. This leaves the repetition statements – statements that are generally well-known to be prone to infinite execution condition. However, the semantics of C3L preclude this. First, consider the “while” loop semantics defined previously:

\[
\langle b, \psi \rangle \downarrow \text{false} \\
\langle \text{while } (b, o) s, \psi \rangle \rightarrow \psi
\]

\[
\langle b, \psi \rangle \downarrow \text{true} \quad \langle o, \psi \rangle \downarrow n \\
\langle \text{while } (b, o) s, \psi \rangle \rightarrow \psi (n < 1)
\]

\[
\langle b, \psi \rangle \downarrow \text{true} \quad \langle o, \psi \rangle \downarrow n \quad \langle s, \psi \rangle \rightarrow \Psi^* \quad \langle \text{while } (b, n-1) s, \psi^* \rangle \rightarrow \psi' (n \geq 1)
\]

When the Boolean expression b evaluates to false, the loop is removed from the statement list as shown in the first rule and, therefore, does not execute indefinitely. The last two rules govern when the Boolean expression b evaluates to true. In these rules, the behavior depends on the evaluation of the maximum loop iteration count, the constant
expression \( o \). By definition, a constant expression is one whose value is constant for all
time and for all states, and, therefore, is statically deterministic. When \( o \) evaluates to a
value less than 1, the loop terminates; when \( o \) evaluates to a value greater than one, the
loop iterates, but repeats with a maximum loop iteration count one less than that from the
preceding iteration. Therefore, regardless of how the Boolean control expression changes
over the execution of the “while” loop, the maximum loop iteration count will always
eventual be less than one; consequently all while loops in C3L terminate in finite time.

Second, consider the “for” loop semantics defined previously:

\[
\begin{align*}
\langle o_0, \psi \rangle &\downarrow n_0 \quad \langle o_1, \psi \rangle \downarrow n_1 \\
\langle s, \psi[n_0/x] \rangle &\rightarrow \psi'' \\
\langle \text{for } (x = o_0 \text{ to } o_1) s, \psi \rangle &\rightarrow \psi' \\
\end{align*}
\]

\[
\begin{align*}
\langle x, \psi \rangle &\rightarrow \psi(x) \\
\langle n_0, \psi \rangle &\downarrow n_1 \\
\langle s, \psi \rangle &\rightarrow \psi'' \\
\langle \text{for } (x = o_0 \text{ to } o_1) s, \psi \rangle &\rightarrow \psi' \\
\end{align*}
\]

\[
\begin{align*}
\langle x, \psi \rangle &\rightarrow \psi(x) \\
\langle n_0, \psi \rangle &\downarrow n_1 \\
\langle s, \psi \rangle &\rightarrow \psi'' \\
\langle \text{for } (x = o_0 \text{ to } o_1) s, \psi \rangle &\rightarrow \psi' \\
\end{align*}
\]
\[
\begin{align*}
\langle o_0, \psi \rangle & \downarrow n_0 \\
\langle o_1, \psi \rangle & \downarrow n_1 \\
\langle o_2, \psi \rangle & \downarrow n_2 \\
\{ \text{for } (x = o_0 \text{ to } o_1 \text{ by } o_2) \ s, \psi \} & \rightarrow \psi [n_0/x] \Big( \neg \left( n_0 \leq n_1 \land n_2 > 0 \lor n_0 \geq n_1 \land n_2 < 0 \right) \Big)
\end{align*}
\]

\[
\begin{align*}
\langle s, \psi [n_0/x] \rangle & \rightarrow \psi'' \\
\{ \text{metafor } x \ n_1 \ n_2 \ s, \psi'' \left[ \left( \psi''(x) + n_2 \right)/x \right] \} & \rightarrow \psi' \left( n_0 \leq n_1 \land n_2 > 0 \lor n_0 \geq n_1 \land n_2 < 0 \right) \land \psi(x) = \psi''(x)
\end{align*}
\]

\[
\begin{align*}
\{ x, \psi \} & \rightarrow \psi(x) \\
\{ n_1, \psi \} & \downarrow n_1 \\
\psi(x) > n_1
\end{align*}
\]

\[
\begin{align*}
\{ x, \psi \} & \rightarrow \psi(x) \\
\{ n_1, \psi \} & \downarrow n_1 \\
\{ n_2, \psi \} & \downarrow n_2 \\
\{ s, \psi \} & \rightarrow \psi'' \\
\{ \text{metafor } x \ n_1 \ n_2 \ s, \psi'' \left[ \left( \psi''(x) + n_2 \right)/x \right] \} & \rightarrow \psi' \left( \psi(x) \leq n_1 \land \psi(x) = \psi''(x) \right)
\end{align*}
\]

Again, the initial, the ending and the loop increment values are constant expressions that are statically deterministic. The first four rules apply to “for” loops whose loop control variable is monotonically incrementing by one over the iterations of the loop, while the second four rules apply to “for” loops with program-defined increments. The first, third, fifth, and seventh rules define the end state of the “for” loops when the iteration limit is reached, while the remaining rules define the iteration of the
loops. When the loops iterate, the loop control variables monotonically advance towards the iteration limit; therefore, all C3L “for” loops will eventually reach the limit and terminate. Note that the semantic rules require that the loop control variable be unchanged through the execution of statements within the body of the “for” loop. Therefore, all “for” loops in C3L terminate in finite time.

Since all C3L statements execute in finite time, statement execution reduces the length of the statement list by one in finite time. By well-founded induction on the sublist relation, any statement list must be reduced to the empty list $\mathcal{E}$ in finite time.

Having proved that all C3L event handlers execute in finite time and having shown that input events not otherwise handled are added to the end of the input queue, then handled when they are dequeued, it only remains to be shown that events that enter the input queue are dequeued in finite time. Let $I_{\text{pre}}$ denote the state of the input queue at time $t$ when an input event $i$ is added to the queue. By semantic rule, any event arriving after time $t$ is added to the end of the queue. Consequently, these events cannot affect $I_{\text{pre}}$. Since no other mechanism can add an event to the input event queue, $I_{\text{pre}}$ can never increase. Therefore, to show $i$ is eventually dequeued, it is necessary and sufficient to show that $I_{\text{pre}}$ always reduces to the empty list in finite time. This is obviously shown by well-founded induction on the sublist relation, exactly as was done for the statement list. Therefore, every input event added to the input queue will always be dequeued in finite time. One may then conclude that every event raised to a C3L device is handled in finite time.
Summary

An operational semantics for C3L was presented to formally define the behaviors required of all C3L language constructs. These semantics not only define mechanisms through which expressions may be evaluated and the execution of statements in a given device context are evaluated into a new device context, but also define an operational semantic notation to describe the occurrence of events. In this model, the set of devices that define a C3L program are represented by a set of device contexts, and the interactions between the multiple device contexts of a C3L program are described in terms of raising events between contexts. These semantics do not explicitly refer to the processes that might underlie a C3L language implementation and are instead designed only to describe the operations. In this way, a C3L implementation may be either single threaded or multithreaded and still satisfy the formal language requirements specified in this operation semantic rule system.

The operational semantics of C3L were used to derive formal properties for all C3L programs – specifically, prefix-closed controllability and event fairness were proven, showing that these properties were maintained from the previous version of the language. These results are well-suited for the real-time, embedded control of distributed systems of collaborating, autonomous devices applications for which C3L was intended.
Chapter 5

A Parallelizing Compiler for C3L

Introduction

The C3L language was developed specifically for the application domain of Unmanned, Underwater Vehicles (UUVs) participating in a Mine Counter-Measures (MCM) mission [Phoha2002, Eberbach1999, Phoha2000, Schmiedekamp2006, Skarbez2004] in which collaborating UUVs act as a distributed system of autonomous agents. Effectively programming distributed systems has proven to be technically challenging, requiring a language with a foundation of strong controllability properties, the computational expressiveness approaching a general-purpose language and the ability to easily express communication between devices. C3L was developed specifically for this purpose as discussed in preceding chapters. A compiler for C3L would be highly desirable to further explore this language’s potential in distributed system development.

To aid compiler development, Microsoft has created the Phoenix [Microsoft2008] framework for compiler design and optimization. This framework provides the ‘back-end’ compiler infrastructure, manages the architecturally-neutral intermediate representations (IR) used to represent any program, applies specified compiler optimizations at the proper place in the compilation process, then ‘lowers’ the IR towards an architecturally-dependent representation that is then used to write an object file in Microsoft’s common object file format (COFF). A compiler developer then has only to
implement at a minimum a compiler ‘front-end’ consisting of a language parser and a
compiler to serve as the intermediary between the parser and the Phoenix framework.

A high-level language like C3L typically has functionality that is difficult and
impractical to implement directly in IR. Communication functions as well as the
evaluation of arithmetic and relational operators on more complex data types like C3L
sets and strings are more accurately and efficiently described in a high-level language
like C++ or C#. In these cases, instead of having the compiler directly generate IR
implementing these functions the compiler may instead generate calls into a run-time
library. This approach has several advantages; most importantly, the high-level language
description of this higher-level functionality may be compiled and optimized into a
library object file, enabling quicker implementation of complex language functions while
yielding better initial performance of programs using the optimized runtime library.

This chapter provides a description of a compiler for the C3L language that
utilizes Microsoft’s Phoenix framework for compiler design and optimization. The
discussion demonstrates how the compiler automatically extracts task-level parallelism
inherent in C3L programs and generates multithreaded applications with no explicit
actions needed by the programmer. The multithreaded applications generated in this
manner are proven to be deadlock free for all C3L programs. Finally, an experimental
investigation into the performance of compiled C3L programs show that the executables
generated by the C3L compiler can exhibit near-linear speedups when running on desktop
multiprocessors.
Chapter Roadmap

This chapter begins with a complete description of the compiler architecture and implementation, and then a proof is presented demonstrating that any executable generated by the compiler is guaranteed to be free of deadlock. Next, a set of benchmarking experimental results are presented that show near-linear speedup for certain classes of distributed applications. Finally, the chapter concludes and suggests research directions.

The Compiler Architecture

The C3L compiler utilizes a foundation built on Microsoft’s Phoenix Framework for compiler implementation and optimization [Microsoft2008]. Using this framework, the initial implementation was developed rapidly, and permits targeting object code for a wide range of platforms directly. Phoenix provides an intermediate representation (IR) of four levels with varying degrees of implementation-independence; High-level IR (HIR) is architecturally neutral and is typically generated by a compiler front-end. The Phoenix framework then ‘lowers’ the IR through two intermediate levels to an architecturally-dependent Low-level IR (LIR). Selected optimizations may be applied at appropriate levels in the lowering process. Consequently, compiler development in the Phoenix framework requires only a ‘front end’ for the compiler to parse the language according to its syntax and generate an appropriate HIR program representation according to the language’s semantics. To provide the advanced functionality required in C3L programs, a runtime library for the language is also need in lieu of creating complex HIR directly.
from the compiler. Figure 5-1 illustrates the C3L compiler architecture. Implemented in C#, the compiler front end consists of the scanner, parser, compiler, C3L runtime library interface and main program. The compiler back end is provided by the Phoenix framework, and interaction between the front and back ends is governed by the front end’s compiler class. The compiler composes a program by implementing the event handlers of a C3L program and making appropriate variable declarations. The compiler then adds automatically generated functions to create a C3L runtime environment suitable for the program, initialize program structures and variables as indicated by the

C3L program or language semantics, create the threads and necessary mutual
exclusion mechanisms, and then create an application entry point. High-level run-time functionality is implemented as a C++ static library which provides an exposed C interface to functions which map library calls into appropriate constructor calls and method invocations on C3L runtime objects. After successfully completing the high-level module definition, the Phoenix framework is used to ‘lower’ the program representation into architecturally-dependent object code in Microsoft’s Common Object File Format, applying specified optimizations in the process. Once the object code for the program is generated, the compiler spawns the C++ linker to generate an executable program. These components will be discussed in more detail in the following sections.

The Compiler Front End

The C3L Compiler front-end is constructed using a conventional scanner/parser/compiler organization and is implement in unsafe C#.

The Scanner

The scanner is responsible for reading the source code, discarding comments, and tokenizing the program for easier processing by the parser. The scanner is an implementation of the IEnumerable interface providing simplified access by the parser.
The Parser

The parser object interprets the sequence of tokens extracted from the program by the scanner according to the syntax of the language. Largely structured to parallel the C3L language’s syntax, the parser is solely responsible for enforcing the language syntax, utilizing the compiler interface to invoke the creation of IR according the legal program constructs and generating error messages in the event of violation of language syntax.

In processing a C3L program, the parser creates the proper contexts and generates required runtime function calls to initialize the runtime environment, register C3L event handlers with the runtime to enable call back processing, and then start the event processing mechanism of the program. Unlike traditional applications, the entry point is a single explicit function; C3L program may have zero or more entry points (zero if the program requires no action until an external control event or uncontrollable plant event occurs). The parser therefore generates a standard entry point function from which the required initializations are performed.

C3L plant and memory parameters are implemented as module-scoped variables with decorated names to guarantee uniqueness in each device’s scope. Local variables in event handler scopes are created as standard stack variables with function scope.

The Compiler

The compiler is the critical link between the language front-end (implemented in the scanner and the parser) and the Phoenix framework. The compiler instantiates required Phoenix objects, maintains symbol tables, exposes an interface to the parser
through which HIR is created according to language rules, then creates and executes a customizable phase list to transform the High-level IR completely through to executable object code. The complete default list of phases utilized by the C3L compiler in lowering the HIR to executable LIR is shown in Table 5-1.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Check Phase</td>
<td>Enforce type check rules on the HIR. Target architecture independent.</td>
</tr>
<tr>
<td>MIR Lower Phase</td>
<td>Lower HIR to MIR. Target architecture independent.</td>
</tr>
<tr>
<td>Runtime Canonicalize Phase</td>
<td>Generate canonical forms from MIR in preparation for subsequent phases. Target architecture dependent.</td>
</tr>
<tr>
<td>Runtime Lower Phase</td>
<td>Lower MIR to LIR. Target architecture dependent.</td>
</tr>
<tr>
<td>Global Register Allocator Phase</td>
<td>Priority-order register allocation phase. Target architecture dependent.</td>
</tr>
<tr>
<td>Stack Allocator Phase</td>
<td>Prepare stack. Target architecture dependent.</td>
</tr>
<tr>
<td>Frame Generation Phase</td>
<td>Prepare frame. Target architecture dependent.</td>
</tr>
<tr>
<td>Switch Lower Phase</td>
<td>Transform LIR to lower-level LIR. Target architecture dependent.</td>
</tr>
<tr>
<td>Block Layout Phase</td>
<td>Perform block-level layout of executable image. Target architecture dependent.</td>
</tr>
<tr>
<td>Flow Optimizer Phase</td>
<td>Perform flow optimizations on image. Target architecture dependent.</td>
</tr>
<tr>
<td>Encode Phase</td>
<td>Generate assembly from LIR. Target architecture dependent.</td>
</tr>
<tr>
<td>Lister phase</td>
<td>List phase. Target-architecture dependent.</td>
</tr>
<tr>
<td>Build Plug-in Phase</td>
<td>Builds and appends to phase list any specified plug-in phases.</td>
</tr>
</tbody>
</table>

The compiler contains a complete description of the functions externalized by the C3L Runtime Library; this enables the parser and compiler to generate calls into the runtime library as dictated by the C3L program and language semantics. Certain runtime
language calls are generated for all C3L programs to initialize the runtime environment, initialize C3L devices by creating call-back links back into the C3L program, and perform other initialization tasks like instantiating string objects, etc. See section 0 for a description of the complete set of these functions. Additionally, the compiler implements a C-style printf function that is not exposed in the C3L language but may be used for compiler development and verification.

**The Runtime Library Interface**

The runtime library interface provides the functionality required to create the external function header definitions within the compiler to enable both exposed runtime language functions as well as hidden functions inaccessible directly from C3L programs but necessary for the operation of the runtime environment. The implicit (i.e. accessible directly by C3L programs) functions and declarations are shown in Table 5-2 and in Table 5-3. The functions described in Table 5-3 are random number functions in support of stochastic algorithms and simulations. These functions differ somewhat from the other math in Table 5-2 as the random functions were implemented for the runtime library, whereas most of the math functions are simply linked to the underlying C runtime library implementations. The random number functions include functions to generate uniform and Gaussian distributed real numbers, functions to generate uniform, and Poisson distributed integer numbers, functions to simulate fair or weighted coin flips, and a function to seed the random number generator.
Table 5-2: Standard, Implicit C3L Runtime Library Functions and Definitions

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>real __cdecl cos(real x)</td>
<td>Returns the cosine of the angle x (x in Radians)</td>
</tr>
<tr>
<td>real __cdecl sin(real x)</td>
<td>Returns the sine of the angle x (x in Radians)</td>
</tr>
<tr>
<td>real __cdecl tan(real x)</td>
<td>Returns the tangent of the angle x (x in Radians)</td>
</tr>
<tr>
<td>real __cdecl acos(real x)</td>
<td>Returns the inverse cosine of x in Radians</td>
</tr>
<tr>
<td>real __cdecl asin(real x)</td>
<td>Returns the inverse sine of x in Radians</td>
</tr>
<tr>
<td>real __cdecl atan(real x)</td>
<td>Returns the inverse tangent of x in Radians</td>
</tr>
<tr>
<td>real __cdecl cosh(real x)</td>
<td>Returns the hyperbolic cosine of the angle x (x in Radians)</td>
</tr>
<tr>
<td>real __cdecl sinh(real x)</td>
<td>Returns the hyperbolic sine of the angle x (x in Radians)</td>
</tr>
<tr>
<td>real __cdecl tanh(real x)</td>
<td>Returns the hyperbolic tangent of the angle x (x in Radians)</td>
</tr>
<tr>
<td>real __cdecl exp(real x)</td>
<td>Returns $e^x$</td>
</tr>
<tr>
<td>real __cdecl log(real x)</td>
<td>Returns $\ln x$</td>
</tr>
<tr>
<td>real __cdecl log(real x)</td>
<td>Returns $\log_{10} x$</td>
</tr>
<tr>
<td>real __cdecl ceil(real x)</td>
<td>Returns $\lceil x \rceil$</td>
</tr>
<tr>
<td>real __cdecl floor(real x)</td>
<td>Returns $\lfloor x \rfloor$</td>
</tr>
<tr>
<td>real __cdecl abs(real x)</td>
<td>Returns $</td>
</tr>
<tr>
<td>real __cdecl sqrt(real x)</td>
<td>Returns $\sqrt{x}$</td>
</tr>
<tr>
<td>real __cdecl pow(real x, real y)</td>
<td>Returns $x^y$</td>
</tr>
<tr>
<td>real __cdecl atan2(real x, real y)</td>
<td>Returns the angle (in radians) in the correct quadrant corresponding to the inverse tangent of x/y.</td>
</tr>
<tr>
<td>real __cdecl fmod(real x, real y)</td>
<td>Returns floating-point modulus.</td>
</tr>
</tbody>
</table>

- $\pi$: 3.14159265358979323
- $e$: 2.7182818284590452353

- set all: The set consisting of all possible elements.
- set empty: The set consisting of no elements.
- set me: A one-element set uniquely identifying the device in whose context the me set is evaluated.
- set you: A one-element set uniquely identifying the device from which a control event was received. When a device raises a control event to itself, you equals me.

- trace(string format[,…]): Printf-styled console output.
- stop(): Disables event processing and outputs the time since the application started to the console.
The Compiler Back End

The Microsoft Phoenix framework provides a complete back-end implementation that is fully utilized by the C3L compiler. This back-end is used to generate object files adhering to Microsoft’s Common Object File Format (COFF) standard.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>real randu()</td>
<td>Returns a real number, distributed uniformly on the interval [0,1)</td>
</tr>
<tr>
<td>real randReal()</td>
<td>Returns a real number, distributed uniformly on the interval [a,b)</td>
</tr>
<tr>
<td>real randReal2(real a, real b)</td>
<td>Returns a real number, distributed Gaussian with mean 0 and standard deviation 1.</td>
</tr>
<tr>
<td>real randn()</td>
<td>Returns an integer, uniformly distributed on the interval [0,1]</td>
</tr>
<tr>
<td>integer randInteger()</td>
<td>Returns an integer, uniformly distributed on the interval [a,b]</td>
</tr>
<tr>
<td>integer randInteger2(integer a, integer b)</td>
<td>Returns an nonnegative integer, distributed Poisson with a mean of a.</td>
</tr>
<tr>
<td>integer PoissonRV(double a)</td>
<td>Returns a zero or a one with equal probability</td>
</tr>
<tr>
<td>integer fairCoinFlip</td>
<td>Returns a zero or a one s.t. P(1) = p, P(0) = 1-p</td>
</tr>
<tr>
<td>integer coinFlip(double p)</td>
<td>Seeds the random number generator with the value s. Returns 0.</td>
</tr>
<tr>
<td>integer seedRNG(integer s)</td>
<td></td>
</tr>
</tbody>
</table>

The C3L Runtime Library

The C3L Runtime Library is a set of C++ classes and functions used to provide the higher-level functionality of C3L. The library implements set and string operations and is used to allocate string variables (simple data types and the set type are allocated by
the compiler in the program object file). More importantly, the library provides OO encapsulation of C3L program structures. Each device defined in a C3L program is encapsulated in a C3LDevice object and instantiated by a call into the library. Each device includes an input event queue which is also implemented as a C++ object. This C3LEventQueue object is a wrapper object for a C++ Standard Template Library (STL) dequeue with specific support provide for language functionality. By default, one thread is instantiated for each C3LDevice to perform control event processing. Each device thread sequentially processes input events from the head of the queue and blocks when the queue is empty.

The C3L Runtime Environment is implemented as an instance of the C3LRuntimeEnvironment class. In addition to maintaining a list of devices defined in the environment, the runtime environment also implements its own input event queue and a thread to process them; in this way, any event raised by a C3L device requires only constant complexity in the device thread independent of the complexity of the recipient list of devices to which the event is directed. This leaves routing the event to the environment. The last component to the runtime environment is a socket-based network interface. This interface, with its own buffer and threads, enables this runtime environment to communicate with devices in other environments across a network.

The C3L Runtime Library uses C++ STL data structures and the Boost [30] library for thread generation, mutual exclusions, locks, and condition variables. The library also depends on the standard C runtime library (crt.lib).
Linking C3L Programs

Object files generated by the compiler are linked with the C3L Runtime Library (c3lrt.lib), the C standard runtime library, and potentially other libraries on a per-application level basis using Microsoft’s C++ linker. The linker is invoked automatically by the C3L compiler upon successful compilation of a program.

Debugging Support for C3L Programs

Microsoft’s Phoenix framework enables full debugging information to be created and maintained for generated executables, including Program Debug Database files (.pdb) and manifest files. Consequently, one may debug a C3L program as easily as a C++ or C# program using Microsoft’s Visual Studio.

Compiler Implementation Details

The compiler developed for C3L can compile any legal program, and the runtime library fully supports all language constructs\(^2\). The compiler uses standard 32 bit signed integer representation of the C3L integer type and 64 bit floating point format for the C3L real type. C3L Sets are implemented using Pascal-styled bit vectors with a capacity of 2047 distinct set elements. Additionally, the C3L compiler supports the semantics-specified short-circuit evaluation of logical operators.

\(^2\) There are currently unresolved integration issues between code generated by the compiler and the C3L Runtime Library – specifically with respect to the way strings are handled. The number and types of event message parameters is also currently restricted by the runtime library.
The C3L Runtime Environment

As a C3L program consists of an arbitrary number of devices, the compiler generates the specified set of event handlers, plant parameters and memory variables for each device. The parameters and variables are implemented as global variables in the program image, and each event is implemented as a function. As C3L semantics require evaluation of an uncontrollable event’s precondition before either executing the event handler or adding the event to the input queue, the precondition is implemented as a separate function in itself, allowing the runtime environment to evaluate the precondition in the device context before taking any action by executing an event handler. Preconditions in controllable events are implemented as a conditional branch within the event handler; consequently no explicit run-time action is required in handling them.
The compiler generates an input event queue operating as a blocking FIFO buffer for each C3L device, then spawns a single process that takes input events from the head of the queue (or blocks until an event is added to the queue) and processes that event, repeating indefinitely. Since each device may raise control events to itself and/or to one or more other devices, the compiler generates a dedicated event queue to receive events raised by the devices. A separate process services this queue (also implemented as a blocking FIFO buffer) and performs routing of control events – in effect dispatching them back to devices. This routing thread ensures that events are raised by C3L devices with constant complexity with respect to the number of devices to which the event is directed with the context of the device process. Using this methodology, a C3L program consisting of N devices will be compiled into a multithreaded executable with N+1 threads. The compiler also generates MUTEX locks on the STL deques used in the implementation of the event queues to guarantee thread safety. Figure 5-2 illustrates the conceptual model of the runtime environment create by the compiler for any C3L program.

Implementation of C3L Language Constructs

The preceding sections described how some C3L language constructs were implemented by the compiler. To summarize those described to this point, C3L plant parameters and controller memory variables are implemented as global (i.e. module-scoped) variables with name decoration to ensure not only uniqueness in this scope but also that a device may access only its plant or memory variables according to C3L
semantics. The implementation of C3L controllable, uncontrollable and control events are analogous to simple void-returning function definitions; however there are significant semantics associated with these constructs that require further discussion.

*The Structure of Controllable Event Handlers*

Since C3L controllable event handlers are invoked directly by control event handlers in a manner identical to simple function calls, it should not be surprising that the structure of controllable event handlers is closest to tradition function calls. The semantics of C3L limits plant event handlers to act on the set of plant parameters only through assignment statements; these semantics are enforced by the parser. The one aspect of controllable event handlers that requires attention is the optional definition of the event precondition. Recall from C3L semantics that any plant event may include the specification of a precondition in the form of a Boolean expression in the set of state variables and any message variables. The event handler may only be executed if its precondition evaluates to true. The C3L compiler implements this through the explicit evaluation of the Boolean expression given and a conditional branch instruction that ‘skips’ the event handler implementation when the Boolean expression evaluates to false.
Figure 5-3 illustrates a template C3L controllable event handler definition with specified precondition and the corresponding control flow graph generated by the compiler. When there is no preconditioned specified, the compiler does not generate any evaluate IR or the conditional branch instruction, resulting in a trivial control flow graph (Entry Point $\rightarrow$ Implementation $\rightarrow$ Exit Point.)

The Structure of Uncontrollable Event Handlers

Though similar to controllable event handlers in what they do, uncontrollable event handlers differ significantly from controllable events in several key aspects.

```
controllable
someEvent (<boolExp>) {
    //implementation
}
```

Figure 5-3 (a): A template C3L controllable event handler with specified precondition.

Figure 5-3 (b): The control flow graph generated by the compiler for the controllable event handler specified in (a).

Figure 5-3: A template C3L controllable event handler definition with specified precondition and the corresponding control flow graph generated by the compiler.
Uncontrollable event handlers are never explicitly invoked by C3L programs – they are only invoked by the underlying hardware. When raised, uncontrollable events require immediate handling, and if the precondition for the event is met the event handler must be executed. After the event handler for the uncontrollable event is executed, the uncontrollable event is then added to the input event queue so that the corresponding control event handler can be executed in turn. Like controllable events, the precondition for uncontrollable events is optional.

Since explicit, conditional action is required by the event processing mechanisms for the runtime environment (i.e. processes or entities outside of the device context proper), it was necessary to implement uncontrollable event handlers differently than controllable event handlers. Specifically, the precondition is not implemented simply as some expression-evaluation IR and a conditional branch, but rather as separate Boolean-valued function in itself. This enables the run time environment to determine dynamically what to do when an uncontrollable event occurs. The specific steps taken by the parser when processing an uncontrollable event handler are as follows:

1. Check for uniqueness of the identifier and declare that identifier as an uncontrollable event handler.

2. If a precondition is specified, generate a separate function according to the C++ template illustrated in Figure 5-4 (b):
3. Generate the uncontrollable event handler with a decorated name to identifier it as an uncontrollable event.

When an uncontrollable event is raised to a device in the C3L runtime environment, the device checks to see if there is a precondition defined for the event. If there is not a precondition or if the precondition evaluates to true, then the uncontrollable event handler is executed and the event is added to the input queue. The control flow graph of this algorithm is illustrated in Figure 5-5. Note that this algorithm also equally applies to control events raised to the device, as control events have no precondition nor are they uncontrollable; consequently this algorithm simply adds control events to the input queue.

**The Structure of Control Event Handlers**
Control events, like the device’s plant event handlers, have specific and significant semantics associated with their operations. Control events must be explicitly raised by any device; consequently these events always must pass through the input event queue before execution. When they are dequeued, the device must check if the event was

Figure 5-5: Control flow graph of the algorithm handling events raised to a device, describing both the admission of events to the event queue of the device as well as the specific processing of uncontrollable events.
sent by a device authorized to raise the event to this device. If the sender is authorized, then the control event handler for this event is executed; if not, the event is discarded.

This functionality is implemented structurally by the compiler in much the same way as the precondition for controllable events was – by protecting the implementation of the control event implementation with some evaluation statements and a conditional branch; if the sender is not authorized, the conditional branch takes the control flow directly to the exit point of the control event handler. This authorization is determined at run-time by evaluating the set relational expression `you in <setExp>`, where `you` is a hidden parameter always associated with events raised in C3L. A template control

```
control event someEvent authorize <setExp> {
   //implementation
}
```

Figure 5-6 (a): A template C3L control event handler with specified authorization.

![Control Flow Graph](image)

Figure 5-6 (b): The control flow graph generated by the compiler for the control event handler specified in Figure 5-6 (a).

Figure 5-6: A template C3L control event handler definition with specified authorization and the corresponding control flow graph generated by the compiler.
event handler definition with a specified authorization expression and the corresponding control flow graph generated by the compiler are illustrated in Figure 5-6. Note that if no explicit authorization is specified, the set expression defaults to the ‘me’ set according to C3L semantics – therefore only the device can successfully raise the event to itself.

The Structure of the C3L Runtime Environment Event Processor

When a device raises an event, that event is placed in the runtime environment’s event queue. This allows the complexity of raising an event to be constant for the device regardless of the number of devices to which the event is directed. The runtime environment takes the events and raises them to the devices to which they are directed. The pseudo-code algorithm for the runtime environment event processor is illustrated in Figure 5-7.

(C++ pseudo-code)
Runtime environment event processing thread entry point:
1. while (true)
   a. Dequeue an event (block if no events in queue)
   b. If the event is raised to all:
      i. For each device in the runtime environment:
         1. Raise the event to this device
   c. else // the event is not raised to all, but rather some subset
      i. For each device in the runtime environment:
         1. If (this device’s me) in the direct set:
            a. Raise the event to this device

Figure 5-7: The C3L runtime environment event processing algorithm
The Structure of C3L Device Executors

Execution of statements in C3L programs occurs exclusively during the execution of event handlers in response to the occurrence of one of the three classes of events. The executor generated by the C3L compiler for each device simply takes a control event from the head of the input event queue (or waits for an event to occur if the queue is empty), then executes the appropriate handler. Executed explicitly from the context of control events, controllable events are also executed in this manner. Uncontrollable events are somewhat different in that they are executed when first raised to a device – see Figure 5-5 for the control flow graph detailing exactly where this occurs. The compiler instantiates an executor for each C3L device that follows a simple algorithm that is repeated indefinitely: get an event from the head of the input queue (block if queue is empty).

Figure 5-8: The Control Flow Graph of the C3L Device Executor, which takes input events from the queue and executes the appropriate handler.
empty), then execute the handler of the event if there is no appropriate handler for the event, the event is simply discarded. The control flow graph for the executor implemented for each C3L device is illustrated in Figure 5-8.

**Structure of C3L Executables**

The C3L compiler generates multithreaded executables for any C3L program, with the number of threads generated proportional to the number of devices in the program. The executable program itself, however, is invoked as a conventional single-threaded application with a main function generated automatically by the compiler to initialized the C3L runtime environment, handle all necessary initializations to the environment and to devices to ensure the correct implementation of C3L semantics, spawn processes according to the methodology implemented by the compiler, then start

---

(C++ pseudo-code)

‘main’ application entry point:

2. Call runtime::init()
3. For each device in the C3L program:
   a. Call runtime::deviceInit(const char*, set*)
   b. For each uncontrollable event handler declared in this device:
      i. If the uncontrollable has a precondition, call
         runtime::addEvent(const char*, const char*, int (*)(void)) for
         the precondition wrapped as a function.
      ii. Call runtime::addEvent(const char*, const char*, int (*)(void))
          for the uncontrollable event handler.
   c. For each control event handler declared in this device:
      i. Call runtime::addEvent(const char*, const char*, int (*)(void))
         for the control event handler.
4. Call runtime::setStartup(void (*)(void))
5. Call runtime::start()
6. Call runtime::idle()

Figure 5-9: The structure of the ‘main’ function generated for all C3L programs.
the concurrent execution of the C3L program. The main program generated by the compiler has a specific structure that is illustrated in Figure 5-9 and will be discussed in the following sections.

**Generated ‘main’ Function for C3L Programs**

The C3L generates an application entry point for every C3L program that initializes the necessary run-time environment structures, spawns processes and starts the event handling mechanisms. As illustrated in Figure 5-9, after initializing the runtime environment the main function initialized each device individually through a single call to deviceInit(). This function creates a device object to encapsulate the data structures needed for the runtime management of the device created, identifying the device by name and initializing the ‘me’ set. The ‘me’ set is a single-element set guaranteed to be unique in the runtime environment and may be thought of as a unique device serial number. Then, every event handler which might be called by the runtime environment (specifically uncontrollable event handlers (and their preconditions wrapped as events) and control even handlers, are declared to the device through a call to the addEvent() function which adds the named event to the specified device and provides the callback address for the function.

After each device is initialized, the startup function generated by the C3L compiler is registered. This startup function is generated to perform any initializations of global variables and also raises any events marked to ‘executeOnStartup’. After registering this startup initialization function, the runtime environment is started. This
involves spawning a process to handle events raised to the runtime environment (executing the algorithm illustrated in Figure 5-7) and spawning a separate process for each device registered in the environment to execute events from the head of the input event queue (executing the algorithm illustrated in Figure 5-8). A barrier is created to synchronize the device executors so that all processes are created and all initializations are completed before any event processing begins. Finally, the main function calls the blocking idle() function to prevent the application from terminating.

Formal Properties of Executables Generated by the C3L Compiler

Deadlock in Multithreaded C3L Executables

As the C3L compiler automatically generates multithreaded executables, it is reasonable to be concerned that the multithreaded executables generated by the compiler might be susceptible to deadlock – a situation in which two or more processes competing for resources are all waiting for the other to finish indefinitely. Fortunately, it is a simple matter to prove that all executables generated by the C3L compiler are deadlock free.

**Proposition**: All executables generated by the C3L compiler are deadlock free.

**Proof**: The proof is by contradiction. Assume that multithreaded C3L executables may deadlock. For deadlock to occur, it is necessary to show the following four conditions – the Coffman conditions [Coffman-1971] -are true:

1. Mutual exclusion condition – there exist resources that can only be used by one process at a time.
2. The hold and wait condition – a process already holding resources may request new resources, potentially waiting for them before being able to continue.

3. No preemption condition – there is no mechanism through which a process may be forced to release a resource it holds. Resources may only be released by the holding process.

4. Circular wait condition – two or more processes for a cyclic dependency in which each process waits for a resource held by the next process in the chain.

Condition 1 is clearly satisfied by all multithreaded C3L applications. Every C3L executable consists of at least two input event queues – one for the single C3L device specified in the program, and one for the runtime environment. At least one thread is generated for each of these queues, and since the device may raise events to the runtime and vice-versa, a mutual exclusion mechanism is required on the event queues to prevent corruption of STL iterators and ensure proper data structure operation. Consequently, condition 1 applies to all C3L multithreaded applications.

Moreover, condition 4 also clearly exists in C3L multithreaded applications. Devices raise control events to the runtime environment. The runtime environment then routes (and replicates when needed) those events back to the directed devices. Even in the simplest case described above, there are at least two links in the circular chain, and the only interaction between any objects in the C3L runtime occurs via the event queues. Therefore, condition 4 holds true for all C3L multithreaded applications.

Condition 3 is also clearly true for all C3L multithreaded applications. Once a thread holds a mutual exclusion lock on an event queue, no mechanism is available to force the thread to release it.
Three of the four necessary conditions are always true for all C3L programs. Assuming that deadlock can occur, condition 3 must also be true. This means that a process already holding a resource may request other resources. However, the design of the event queues do not permit this. The only locks used on shared mutual exclusion mechanisms are on the event queues. The locks on the event queues are internal to the queue’s enqueue and dequeue methods. To acquire the lock, a process need only wait its turn for other dequeue or enqueue method executions to leave the critical section. Once the lock is obtained, no other event queue – and consequently, locking mechanism – is in scope. Moreover, calls to enqueue and dequeue events from C3L event queues are only made through the runtime environment and cannot be made directly by any C3L program statement or function call. Consequently, no process holding the shared resource (i.e. the critical section of an enqueue or dequeue method) may request additional resources. Therefore a contradiction in the assumption is found and the hold and wait condition (condition two) is never true. Since the necessary conditions for deadlock are never met, all C3L programs are guaranteed to be deadlock free. \textit{QED.}

\textbf{Event Fairness and Starvation, and Livelock in Multithreaded C3L Executables}

Having shown that the multithreaded executables generated by the C3L compiler are free from deadlock for all C3L programs, the discussion continues with the properties of event fairness, starvation, and livelock. These three well-known multitasking-related problems are closely related. The starvation problem is one in which a process is continually denied access to needed resources and, consequently, can never complete. A
special case of this problem is the livelock problem in which processes are active but the system is not progressing. Livelock is of particular concern when yielding mechanisms are used to break deadlocks. Event unfairness is an analogous problem in which an event is waiting an indefinite period of time before it is processed by the appropriate handler. All three problems center on the input event queues whose mutual exclusion locks provide forced serializations through the critical sections. Consider the starvation problem with respect to C3L multithreaded executables:

**Proposition:** All executables generated by the C3L compiler cannot starve a process.

**Proof:** The proof is by contradiction. Assume that multithreaded C3L executables may starve a process. For this to happen, a process would have to wait indefinitely for a resource. As previously proven, C3L multithreaded applications are free from deadlock, the most obvious condition under which indefinite waiting might occur. In Chapter 4, the semantics of C3L were used to show that the language guarantees event fairness by showing that all events raised to a device are dequeued in finite time. Since the MUTEX locks used from the BOOST libraries grant locks to threads waiting for the lock in FIFO order, and since the multithreaded C3L executables generate one thread for each event queue, it follows that the process that removes events from the event queue must not therefore be waiting indefinitely – a contradiction. Therefore, all executables generated by the C3L compiler cannot starve a process. *QED.*

**Proposition:** All executables generated by the C3L compiler are livelock free.

**Proof:** The proof is by contradiction. Assume that multithreaded C3L executables may experience livelock. This requires that a process be free to act yet the state of the program may not advance. Unless a process is waiting for a lock, the process is advancing the
system state through the execution of program statements or the processing of events. There is no mechanism through which a process may execute some code without advancing the program state. Consequently, there is a contradiction in the assumption and one may conclude that all C3L executables are free of livelock. QED.

**Proposition:** All executables generated by the C3L compiler exhibit fairness with respect to event handling.

**Proof:** The proof is by contradiction. Assume that multithreaded C3L executables do not exhibit event fairness. This requires that some event raised to a C3L device have to wait indefinitely to be handled (i.e. dequeued). The formal semantics of C3L presented in Chapter 4 were used to show that the language itself is fair, and the compiler satisfies the requirements of the semantics by using a FIFO event queue. Since there is no mechanism through which events arriving at a later time can ever be dequeued before events arriving earlier, and since the processes dequeuing events are guaranteed to progress having been shown to be both deadlock and livelock free, there is no case in which any event may ever wait indefinitely to be dequeued and handled – a contradiction. Therefore, all executables generated by the C3L compiler exhibit fairness with respect to event handling.

---

**Performance Evaluation of the Multithreaded Executables Generated by the C3L Compiler on Multicore Systems**

An investigation into the performance of the multi-threaded executables generated by the C3L compiler was conducted to verify that these programs are indeed able to exploit the task-level parallelism inherent in C3L programs. Three basic benchmarking
applications were developed for this purpose. The first benchmark consists of 7 devices organized as a logical ring topology. When any device receives an event, it simply raises the event to the next device in the logical ring. Upon program startup, the event is raised to all 7 devices; consequently, there are 7 concurrent messages propagating around the ring. The program stops when a certain number of events have been passed a check point in the ring. Obviously, this test is practically all communication; consequently this test is labeled ‘Communication-bound’. The second benchmark consists of a variation of the first benchmark in which the only difference is that only one message is in flight around the ring.

The third benchmark test was created to simulate a computationally-intensive distributed program. The same communication and logical topology used in the communication-bound benchmark is used in this test, but a repeatedly executed, complex computation with loop-carried dependencies was added to the event handler. This requires significant CPU time before the message can be passed along to the next destination in the logical ring. And since there are multiple messages in flight, this test has a high degree a concurrency. The computation includes basic floating-point arithmetic as well as function calls to square root, sine and cosine functions, and the number of repetitions was set so that processor utilization was just 100% when all four cores were used. This test is called the ‘Computation-bound’ test though it must be clear that there is still communication occurring in the program.

Experiments were conducted using a Dell Precision PWS690 with Dual Intel Xeon 5160 dual-core CPUs operating at 3.00 GHz with 3.00 GB Ram. The operating system is 32-bit Windows XP Professional Version 2002 with Service Pack 3. The Xeon
5160 CPUs features a bus speed of 1.333 GHz and a 4MB L2 Advance Transfer Cache operating at 3 GHz that is shared by the cores. The L1 cache configuration for this CPU is 32KB instruction cache and 32KB data cache per core. The benchmarks were also run on a Dell Optiplex 755 with Intel Core 2 Duo E6850 CPU 3.00 GHz. with 3.00 GB Ram. The operating system is 32-bit Windows XP Professional Version 2002 with Service Pack 3. The E6850 CPU features a bus speed of 1.333GHz and a 4 MB Advance Smart Cache operating at 3 GHz. The L1 cache consists of a 32 KB instruction cache and a 32 KB data cache per core. In all cases, the platforms on which the testing occurred are ‘normal’ desktop environments – that is, no processes were stopped and regular applications (i.e. email, mp3 player, browsers) were running. The test applications were run at normal priority. To control the level of multiprocessing, the processor affinity was defined for the executable. The test executables require approximately 1.7 MB of memory when executing (less than the L2 but considerably more than the L1 cache capacities).

Figure Figure 5-10 illustrates the results obtained in this preliminary benchmarking evaluation in terms of speedup (the execution time of the program on one core divided by the execution time on multiple cores). Linear speedup is shown for reference. All tests were conducted on the Dual Xeon platform, and the communication-bound tests were also performed on the Core 2 Duo platform. These results show near-linear speedup for the computation-bound test. However, the communication-bound tests actually performed slower when run on multiple-cores. Observation of the processor utilization while these tests executed indicated that when more than one core was used in the communication-bound tests, the utilization of any of the cores was never saturated.
This implies that the communication costs even between cores of the same CPUs are significantly greater than the communication costs between devices executing on the same core. Likely this is a result of data sharing in the event buffer data structures used in the C3L runtime environment. The performance comparison between the Xeon and Core 2 Duo platforms was very similar, though the Core 2 Duo performed approximately 5% better in the two-core case. Since additional testing revealed that on the Dual Xeon platform, the 2 core case were in fact using two cores from the same CPU\(^3\), one may conclude that the slight performance improvement observed on the Core 2 Duo platform may be attributed to better L2 cache performance.

**Summary and Research Directions**

C3L is a language designed for efficient programming of complex distributed systems. The compiler developed for this language is intended not only to further demonstrate the language’s utility but also to leverage features of the language that exploit the parallelism inherent in C3L device descriptions. The task-oriented parallel architecture that results from this natural implementation of C3L programs yields a program ideally suited for today’s distributed systems of chip multiprocessors (CMPs). Moreover, Phoenix’s strong support in targeting various architectures enables the C3L compiler to easily generate code for today’s diverse set of embedded and general-purpose processors. Initial benchmarking tests show that the multithreaded executables generated

---

\(^3\) The 2 core case was conducted several times with differing pairs of cores; asymmetries in the results indicate clearly which two cores were physically part of the same CPU.
by the C3L compiler demonstrate near-linear when the computational complexity of the program is sufficient to offset the expense of communicating across CPU cores.

With a new language and compiler that can automatically generate parallel applications from this language, there are many research directions available. One direction of interest is in optimization of C3L code. This is a challenging direction as the nature of C3L programs – specifically the lack of loops and numerous, frequently-called, small functional units – has been found to be the most challenging to optimize. Another intriguing research direction is studying in the automatic task-based thread parallelism made possible by the language. Though the initial threading model employed by the C3L
compiler seems natural and effective, no claim is made regarding its optimality; the initial benchmarking performance results indicate that, while parallelism can be exploited by the compiler for certain classes of applications, the performance of message-passing across cores is clearly significantly more expensive than within the same core. Consequently, a more detailed investigation of exactly how the communication mechanisms of C3L programs interact with the hardware might prove useful in improving such performance.

Raising events to a set of devices in C3L is straightforward, with the complexities of making that communication abstracted into the language. The C3L compiler supports this abstraction by generating a run-time environment that supports an arbitrary number of devices in a multi-threaded executable. The next step needed to support truly distributed systems is enabling the raising of C3L events over arbitrary network resources. The C3L compiler architecture has been designed to be easily extended to instantiate the ‘gateway’ router within the C3L runtime environment to handle the underlying network protocols required to send events over the network.
Chapter 6

Visualization and Simulation Support

Introduction

The preceding chapters have described the formal foundations of the C3L language, and a compiler based on the Microsoft Phoenix framework for compiler design and optimization has been created to take C3L programs and generate executable code for either simulation studies or for direct implementation on actual hardware. This compiler automatically extracts task-level parallelism inherent in C3L programs and generates a multi-threaded application with no explicit actions required by the programmer; such applications are ideal for the multi-core and many-core processors of today’s and tomorrow’s computers and embedded controllers.

In complex, distributed systems, visualizations of performance and behaviors can be critically important to the overall utility of the application in either simulation studies or in actual applications. To best position C3L as a powerful language for modeling and implementing such systems, the Visualization Tool Kit (VTK) [VTK2008, Schroeder2006, Schroeder2003] was integrated with the C3L compiler. This permits a C3L program to generate directly powerful, 4-D presentations of data in a fully configurable way. Figure 6-1 illustrates a representative screen shot of such a visualization, depicting two collaborating AUVs in a MCM mission in a section of the Chesapeake Bay. The bathymetry was generated directly from NOAA echo depth
sounding data in a text-based XYZ format; such data is freely downloadable from the Internet. Functions were created to convert the latitude and longitude data from the depth sounding data to meters, and then convert the irregularly-spaced depth sounding data to a surface using a filtered 2D Delaunay triangulation. The complexity of the resulting surface may be reduced using a quadric decimation filter if desired. Consequently, a C3L-based simulation or application may include visualizations not only of the `actors’ in the distributed system but also in a realistic scenario capable of utilizing actual bathymetry.

Figure 6-1: Screenshot of a visualization created representing two collaborating AUVs in an MCM mission. The bathymetry visualization was generated directly from NOAA depth sounding data of an area in the Chesapeake Bay off the coast of Virginia.
Chapter Roadmap

This chapter begins with a brief description of the Visualization Toolkit (VTK), and then continues with a description of how visualization support is integrated into the C3L compiler. The chapter then continues by discussion how simulation and application development is unified in C3L. Finally the current status of this work is discussed as are concluding remarks and future research directions.

The Visualization Tool Kit (VTK)

The Visualization Tool Kit (VTK) is an open-source C++ library providing high-performance visualizations suitable for scientific, medical, and engineering applications [VTK2008, Schroeder2006, Schroeder2003]. This object-oriented library supports creation of multistage graphic pipelines to bring data on screen. Source data in VTK may come from a wide variety of mathematical sources provided in VTK such as planes, cylinders, cones, etc. or may be imported into a suitable VTK data object; VTK supports a wide variety of data classifications, including image data, rectilinear grid data, structured and unstructured grid data, unstructured point data, and polygonal data. The output of these data objects may then be processed by zero or more filter objects that transform the data in some way and may operate on more than one input data set and/or provide more than one output. Once the data is processed according to the application’s requirements, a mapper object provides the link between that processed data and the graphics interface. Encapsulation of this graphics interface is provided by a VTK actor. One or more actors are then added to a scene object and the scene may be rendered.
Interactors are also provided to allow the user to interact with the scene in extensible manners. Additionally, the VTK graphics model provides core objects to model lights and cameras, enabling sophisticated visualizations with little effort by the application designer. Finally, though VTK is written in C++, the library includes interpreted wrappers in Tcl, Python and Java to allow even simpler instantiation of its powerful visualizations through these scripting languages.

**Visualization Support in the C3L Compiler**

Visualization support was incorporated into the C3L compiler in three main forms – data processing functionality specific to the MCM mission, development of visualization models particular to the MCM mission, and general visualization support integrated into the compiler. These three areas involve syntactic and semantic extension to the C3L language as well VTK integration with the C3L runtime library and are discussed in more detail in the following sections.

**MCM – Bathymetric Data Processing**

A tremendous volume of bathymetric data is available from various world-wide sources, much of it freely through organizations like the US National Oceanic and Atmospheric Administration (NOAA), the British Oceanographic Data Centre (BODC), and many others. While plentiful, these data sets are processed to widely varying degrees;
for example, bathymetry data from NOAA consists largely of sounding data collected over the past century, geo-referenced to a standard datum, while to BODC’s General Bathymetric Chart of the Oceans (GEBCO) dataset is published geo-referenced but also gridded to one minute resolution. Though more expensive to create and maintain, gridded data is more compact and easier to process generally at the expense of some measure of accuracy – this may be in the form of decreased resolution where more precise data might be available and potentially extrapolated data in areas where data is sparse. Figure 6-2

Figure 6-2: A small sample of ungridded data points in plan view from NOAA’s NGDC Digital Sounding Data. The data are soundings from the Chesapeake Bay off the coast of Virginia and displayed here in NOAA’s GEODAS Hydro-Plot application.
illustrates a small area of ungridded depth sounding data taken NOAA’s National Geophysical Data Center (NGDC) Digital Sounding Data set and graphically depicts the difficulty in working with unstructured data points. Ideally, working with general bathymetry data should support both classifications of data while providing an error-bounded method of reducing the complexity of the data as needed for practical (i.e. computational) applications.

To support this objective, functionality was created utilizing the VTK graphics library to do just this. Depth sounding data is imported into the application as latitude-longitude-depth 3-tuples in xyz format with no restrictions on the structure of the data set. These points are transformed into 3D Cartesian coordinate space and optionally translated such that the center of the x-y box bounding the data set has coordinates <0,0,z>. A color may also be associated with each data point as a function of the point’s depth. The color coding used for this functionality is shown in Figure 6-3. Note that these color codes are designed not only to provide more visual cues in graphical presentations but also to depict depth contours to illustrate depth classifications according to the MCM mission. Figure 6-4 shows a rendering of a coastal area in the central Adriatic Sea using this color coding. Data for this visualization was drawn from the GEBCO one minute gridded data set, and one may clearly see the pixilation effects in the color coded bathymetric display. Note that the color coding is optional; the illustration in Figure 6-1 depicts a bathymetric data set rendered without this color coding and instead uses coarse coloring through the VTK actor for the bathymetry data.
<table>
<thead>
<tr>
<th>Bathymetric Tints</th>
<th>Hypsometric Tints</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Surf Zone</strong></td>
<td></td>
</tr>
<tr>
<td>0 – 3.048 m</td>
<td>0 – 200 m</td>
</tr>
<tr>
<td><strong>Very Shallow Water</strong></td>
<td></td>
</tr>
<tr>
<td>3.048 – 12.192 m</td>
<td>200 – 500 m</td>
</tr>
<tr>
<td><strong>Shallow Water</strong></td>
<td></td>
</tr>
<tr>
<td>12.192 – 60.96 m</td>
<td>500 – 1000 m</td>
</tr>
<tr>
<td><strong>Deep Water</strong></td>
<td></td>
</tr>
<tr>
<td>60.96 – 100 m</td>
<td>1000 – 2000 m</td>
</tr>
<tr>
<td>100 – 150 m</td>
<td>2000 – 3000 m</td>
</tr>
<tr>
<td>150 – 250 m</td>
<td>3000 – 4000 m</td>
</tr>
<tr>
<td>250 – 400 m</td>
<td>&gt; 4000 m</td>
</tr>
<tr>
<td>400 – 600 m</td>
<td></td>
</tr>
<tr>
<td>600 – 800 m</td>
<td></td>
</tr>
<tr>
<td>800 – 1000 m</td>
<td></td>
</tr>
<tr>
<td>1000 – 1500 m</td>
<td></td>
</tr>
<tr>
<td>1500 – 2000 m</td>
<td></td>
</tr>
<tr>
<td>2000 – 2500 m</td>
<td></td>
</tr>
<tr>
<td>2500 – 3000 m</td>
<td></td>
</tr>
<tr>
<td>3000 – 3500 m</td>
<td></td>
</tr>
<tr>
<td>3500 – 4000 m</td>
<td></td>
</tr>
<tr>
<td>&gt; 4000 m</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6-3: Color coding of bathymetric and hypsometric depth ranges defined for use specifically in visualizations for MCM applications.

The transformed and optionally color-coded data next converted into a VTK polygonal data set format. In its present form, the data consists only of unstructured data points with some colors associated with each point; to create an effective graphical presentation of the data, the points need to be assembled into a surface. Since the data set consists of points the 3D space but we know that the ocean’s bathymetry (or a land mass’ topography for that matter) should be a single-valued function of latitude and longitude, a
2 dimension Delaunay triangulation is performed on the data, triangulating on the latitude and longitude dimensions while passing the depth values of the points through. The Delaunay triangulation converts the unstructured point data into a set of polygons subject to the Delaunay constraint – no point of the dataset may be in the interior of any polygon. After the Delaunay triangulation, a triangle filter is applied to ensure that any higher-order polygons generated in the Delaunay triangulation are reduced to triangles.

At this point, the original unstructured point data has been color coded and assembled into a set of triangles, approximating the surface of the ocean floor and/or land mass. Since the original data was unstructured, there may be many points that add little real detail to the data set but adversely affect the computational time needed to manipulate and render the data in a graphic presentation – particularly in a simulation application when computational power is at a premium. The problem is to identify which points, if any, might be removed from the set and to quantify the error introduced in such removals; if this can be done, then one may specify an error bounds on the required data set fidelity and then remove points such that the error bounds is maintained. While there are many ways of doing just this, the approach selected in this work is to use a quadric decimation filter. This filter attempts to remove a certain percentage of the original data points from a data set while guaranteeing the output data set is within a specific error bound of the original data set. Figure 6-5 (a) shows the same area as Figure 6-2 from an overhead point of view in grayscale. The full data set is displayed in both figures. Figure 6-5 (b) has the same perspective as Figure 6-5 (a) but with a quadric decimation filter set a target reduction of 50%. The file sizes of the resultant data sets show that the filter did successfully reduce the complexity of the data by the 50%.
Next, the data set is saved as a file. This way, the processing necessary to produce a data source for a visualization in a simulator or other frequently-run application need only be done when the data changes or when the fidelity of the data representation changes.

Though this functionality was developed for unstructured point data, if gridded data is used as input the same procedure works very well – just faster, and without the need to reduce the data set complexity through quadric decimation.

Figure 6-4: A visualization of an area in the Adriatic sea using the MCM color coding. This data is from the GEBCO one minute gridded data set and shows depth to a MSL of 7 meters.
Figure 6-5 (a): The area of figure 3 after processing into a surface representation. No quadric decimation is used, so all the points in Figure 6-2 are included in the data.

Figure 6-5 (b): The areas of Figure 6-2 and Figure 6-5 (a), but this time with quadric decimation targeting a 50% reduction in the number of points.

Figure 6-5: The affects of quadric decimation.
MCM Visualization Models

In support of visualizations of MCM missions, a library of VTK graphic objects has been developed. This library includes a visualization of a generic AUV that was constructed simply of cylinders, planes, cones, etc., and assembled into a single unit called a VTK Assembly. A VTK assembly is just as it seems, and provides an encapsulation of arbitrarily complex assemblies that may be manipulated and rendered in the same manner as any simple VTK actor could. A notional model of a mine is also included in the library. This library of off-the-shelf models will grow as driven by the needs of the simulation studies conducted using this environment.

C3L Language Extensions

As a purely event-driven language, C3L is ideally suited for simulation applications. To integrate a visualization for a C3L device, simple extensions to the language were made. These extensions specify which devices should include visualizations. When a C3L device is associated with a visualization, it is directly connected to a VTK actor. Variables are included in the device plant state-space to designate 3D position as well as heading, elevation and roll. A C3L run-time library function may be invoked from C3L plant event handlers to force a re-rendering of the scene when the physical position or attitude of a device changes. Note that these extensions do not control how devices move, only visually expresses the physical state of a visualized set of device; it is up to the C3L program and any run-time library support functionality to determine the rules by which devices can move.
Simulation and Application Development Using C3L

C3L was developed specifically to meet the operational needs of the distributed system development for multiple, collaborating AUVs in the MCM mission. The motivation in this effort is to enable high-level semantic behavioral descriptions of MCM missions to be described more directly as a computer program for testing and evaluation. C3L has strong controllability properties and permits simplified development of interacting, autonomous distributed systems. In this section, an application development methodology is presented outlining C3L-based specification of three distinct behavioral elements – plant control, device control and communication – and how this process may include simulation support with visualizations.

Low-level plant design of a device involves identifying device properties, observed conditions that the plant may report, and the required actions necessary to control the device. The properties map directly to plant parameters, the observed conditions map directly to uncontrollable events, and the actions directly map to controllable events in a C3L device plant description. To utilize the visualization support available in the C3L compiler, one simply designates the device for visualization and uses the six variables provided for position and attitude as part of the plant state-space. Additionally, any plant event that changes these position/attitude variables should call the refresh function to allow changes to achieve the highest fidelity in the visual representation of the device’s position. Alternatively, the refresh function may be called at a fixed interval at the discretion of the programmer depending on the exact needs of the application.
The controller design incorporates two key elements - handling of uncontrollable events and managing the data required for normal operation of the device. A control event handler may be provided for each uncontrollable plant event (thought C3L does not require this – simple uncontrollable plant events may update the plant state with no further direct action required by the controller).

Communication design begins with identifying necessary groupings of devices relevant to the particular application. C3L provides the set as a native data type to support implementing formal descriptions of group behaviors and can use them directly in the program. The design process continues by identifying communication-related data storage requirements. Next, an analysis of the communication requirements is performed to identify messages (events) to be sent and received between devices. Table 6-1 shows

Table 6-1: An example control event designed to communicate tracking information to collaborating devices

```plaintext
control
  event TrackBroadcast(temptrack)
    authorize neighbors {
      tracks = tracks + 1;
      full[tracks] = temptracking;
      if (tracks > 3) {
        //update trackinfo
        raise AssertSup(trackinfo) direct neighbors
        supervisor = empty; // device is its own sup
        tracks = 0;
      }
    }
  }
  event AssertSup(temptrack)
    authorize neighbors {
      supervisor = you;
      //update track info
    }

The controller design incorporates two key elements - handling of uncontrollable events and managing the data required for normal operation of the device. A control event handler may be provided for each uncontrollable plant event (thought C3L does not require this – simple uncontrollable plant events may update the plant state with no further direct action required by the controller).

Communication design begins with identifying necessary groupings of devices relevant to the particular application. C3L provides the set as a native data type to support implementing formal descriptions of group behaviors and can use them directly in the program. The design process continues by identifying communication-related data storage requirements. Next, an analysis of the communication requirements is performed to identify messages (events) to be sent and received between devices. Table 6-1 shows

Table 6-1: An example control event designed to communicate tracking information to collaborating devices

```
an example control event designed to communicate tracking information to collaborating devices.

Finally, controller-level event implementations would be modified to report necessary information to appropriate collaborating devices. The result of this design process is a program that describes a fault-tolerant dynamic clustering and supervisor election, while simultaneously describing each device as a complete implementation in itself.

Since the visualization support is integrated into the plant model of the C3L device as part of its state-space (plant) parameters, no additional actions are needed in the controller level; and since most real-world applications of autonomous devices will include position and attitude as part of their state-space representation anyway, there is little added complexity in visualizing applications in C3L.

**Implementation Status and Future Work**

The C3L compiler can handle any legal code and has been shown to integrate fully with the VTK library. The functionality required to import general bathymetric data has been verified to handle either unstructured or gridded data from both NOAA NGDC digital sounding data and BODC GEBCO gridded data, respectively. Non-trivial simulations of actual MCM scenarios are currently under development.

C3L is a language for distributed control of autonomous, collaborating devices, supporting dynamic organization, inter- and intra-device communication through a uniform event interface, and enabling intuitive and effective implementation of
distributed algorithms. Formal properties of the language have been developed, permitting simple verification of certain classes of proofs regarding program behavior under C3L. C3L has been used to implement multi-sensor networking applications [Biswas2006] and has been adopted by Lockheed Martin for integration with the Service Oriented Architecture for collaborative unmanned undersea vehicles for naval applications. The intent is to integrate the compiler for this language with visualization support capable of depicting real-world scenarios of complex data sets with error-bounded precision; since the Microsoft Phoenix framework upon which the compiler is built is capable of targeting any supported architecture we then hope to demonstrate the ease with which application development may transition from simulation to implementation. More sophisticated scenarios will emphasize the strengths and identify the weaknesses of the language, the compiler as a simulator and eventually the compiler in implementing executable code for real hardware. The overarching aim is to streamline the design cycle from high-level semantic abstraction to actual working implementation in a unified framework.
Chapter 7
Application Study

Introduction and Motivation

C3L was designed from the ground up to support the operational needs for the development of distributed systems of collaborating, autonomous, mobile sensor platforms, specifically for the Mine Countermeasures Mission (MCM). In some instances of this application, several collaborating Autonomous Underwater Vehicles (AUVs) perform scanning, mapping and identification phases in which scanning and mapping is generally performed by side-scan sonar equipped AUVs, and identification is performed by either side-scan sonars in higher-resolution modes or other specialized sensor equipment. This investigation will focus on a simple application where two collaborating, homogeneous, notional AUVs configured for parallel scanning and mapping missions, then a third heterogeneous, notional AUV is dynamically tasked to identify mine-like contacts. While the application itself is not terribly complex, the purpose of this investigation is to demonstrate how the model-based software engineering paradigm provided by C3L may be employed in distributed system development.

Chapter Roadmap

This chapter will begin with a description of the physics and differential-equation based kinematic and environmental models employed in this investigation. Then,
mapping these models into C3L will be described and intermediate verification of the C3L implementation of these models will be presented. Finally, the performance results of the complete C3L implementation of the scenario will be presented and this chapter will be concluded.

**Application Model Development**

**Scenario Description**

The scenario considered in this investigation consists of the scanning, mapping and identification of mine-like contacts in the MCM mission. The area to be scanned is a six nm by six nm square region of an ocean, and the platforms performing the mission

<table>
<thead>
<tr>
<th>Table 7-1: Performance Characteristics of AUVs</th>
<th>Table 7-1 (a) Scanning-Mapping AUVs</th>
<th>Table 7-1 (b) Identification AUV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Speed (knots)</td>
<td>3.8</td>
<td>5.0</td>
</tr>
<tr>
<td>Travel Speed (knots)</td>
<td>3.5</td>
<td>4.5</td>
</tr>
<tr>
<td>Operating Speed (knots)</td>
<td>3.5</td>
<td>2.0</td>
</tr>
<tr>
<td>Endurance (hrs)</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Maximum Acceleration (knots/sec)</td>
<td>2.0</td>
<td>2.65</td>
</tr>
<tr>
<td>Turning Rate (degrees/min)</td>
<td>180/3</td>
<td>180/3</td>
</tr>
<tr>
<td>Operating Depth (nm)</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>Side scan sonar range (nm)</td>
<td>2 x 0.25</td>
<td>0.075</td>
</tr>
</tbody>
</table>
will consist of three AUVs. Two of these AUVs will be configured for side-scan sonar swath imaging. One of the AUVs will be configured for short-range, high-resolution imaging for identification of mine-like contacts. The basic performance characteristics of these AUVs are shown in Table 7-1.

The two scanning/mapping AUVs will be tasked with scanning one half of the 36

Figure 7-1: MCM Region, Segmentation and AUV Assignments.
nm² area, with each AUV responsible for two of four 1.5 nm x 6 nm segments. With a
total side-to-side scanning range of 0.5 nm, each AUV will make three scanning passes
through each segment. The total region, segmentation and AUV assignments are
illustrated in Figure 7-1.

Kinematic Modeling

AUV kinematics belongs to a class of systems known as non-holonomic systems
– that is, systems whose state is path-dependent. Though there are related definitions in
other contexts, in the field of robotics, a non-holonomic system is one in which the total
number of controllable degrees of freedom is equal to or less than the total number of
degrees of freedom in the system [Goldstein1980]. This is so as the AUV is free to move
laterally, but there is no control mechanism through which such movement can be
actuated. Qualitatively, simple AUVs can steer to a heading, move forward, surface and
dive, however formal analytical, quantitative models describing these motions can be
quite complex. A simple, differential equation-based model is presented here to describe
the motion of an AUV.

Acceleration Model

The relationship between acceleration, velocity, and position are well-known in
Newtonian mechanics as a second-order ordinary differential equation [Kleppner1973].
Though straightforward in textbook examples, real-world applications make closed-form
solutions to these equations more complex than suitable in a simple investigation. However, it is possible to model the movement of an AUV using a differential equation approach quite simply, particularly using the numerical methods and properties of a discrete time simulation. In this model, position is computed as the discrete integration of velocity, and velocity is computed as the discrete integration of acceleration. To make the model realistic, however, nonlinear acceleration must be modeled – particularly in implementing a simulated auto-navigation system as one would expect to find on an autonomous vehicle. A nonlinear acceleration suitable for an AUV would qualitatively have highest acceleration when the speed was low, and the acceleration would drop off as speed approached the maximum speed. This implies an exponential model for acceleration. While a similar model would also be suitable for deceleration considering friction losses that increase with velocity, the non-rotating propeller of an AUV and its potential braking action could reasonably interact resulting more effective deceleration. Therefore, a simple linear model for deceleration was selected for this investigation. These models also interact with a control system to govern the AUV’s speed. This control system consists of a set point (the desired speed), and feedback consisting of the actual speed. The simple control system computes an acceleration from the model when the actual speed is less than the desired speed and a deceleration when the actual speed is greater than the desired speed. Since there is a bona fide control system implemented, there are of course real controllability issues – most significantly, the stability of the control system is an issue. For reasons that shall become clear, the bandwidth of the simulation control system is limited to 1 Hz; consequently, linear control (i.e. that used for deceleration) was shown to be unstable in testing when the gain was sufficient to
allow reasonably paced deceleration, so the linear deceleration model includes a term proportional to the error in speed. The acceleration model used in this investigation is shown in Eq. 7.1, and the deceleration model with the proportional control system gain used is shown in Eq. 7.2, where \( \text{maxAccel} \) is the maximum acceleration (in knots/sec), \( \text{maxSpeed} \) is the maximum speed (in knots), and \( \text{targetSpeed} \) is the speed control system set speed (in knots).

\[
\frac{\Delta \text{speed}}{\Delta t} = \text{maxAccel} \cdot e^{\frac{\text{speed} - \text{ln(0.003346)}}{\text{maxSpeed}}} \quad 7.1
\]

\[
\frac{\Delta \text{speed}}{\Delta t} = -0.5 \cdot \text{maxAccel} \cdot \frac{(\text{speed} - \text{targetSpeed})}{\text{maxSpeed}} \quad 7.2
\]

Graphical representations of the models used for the two AUVs in this investigation are illustrated in Figure 7-2.

**Depth Model**

Submersible vehicles typically control their depth either through altering buoyancy, steering using forward propulsion, bow plane and tail fins, or some combination of the two. The notional AUVs used in this investigation are intended to be simple, cheap devices suitable for larger scale deployment; consequently the propulsive mechanism of depth control was selected in which the AUV is assumed to be neutrally buoyant. The kinematic model used to describe the ability of an AUV to control depth is then a first-order differential equation relating the change in depth with the forward speed.
of the AUV. In this model, the AUVs are assumed to surface or dive at a 15 degree angle when changing depth. The equation for this model is shown in Eq. 7.3.

The control system employed by the AUVs to maintain depth is similar to that used to maintain speed. When the depth differs from the desired depth, the depth model is used to compute how the depth should change.

\[
\frac{\Delta \text{depth}}{\Delta t} = \text{speed} \cdot \sin 15^\circ
\]  

**Figure 7-2**: Acceleration and Deceleration vs. Speed for the AUVs.
Through the mechanism for steering an AUV can be similar to that for controlling depth, a different model was selected for describing how AUVs change headings. This was done to simplify mission planning by using a constant turning rate. The AUVs are assumed to be able to turn at a constant rate of 180 degrees in 3 minutes, independent of the forward velocity of the vehicle. Though this is not a typical performance characteristic of AUVs, it is a simplifying assumption made in this work that can be justified by the fact that the missions will have the AUVs traveling at speed most of the time, so the actual difference in the results of this investigation would be minimal if a more complex heading model was used.

The control system employed by the AUV to maintain heading is also simple and similar to the others described thus far; if the desired heading is different from the actual heading, the AUV will turn toward the desired heading at a constant rate based on the heading model. This model is shown in Eq. 7.4. Note that heading has the same connotation as azimuth in navigation applications; a heading of zero degrees is directed along the positive y axis while a heading of 90 degrees is directed along the positive x axis.

\[
\frac{\Delta \text{heading}}{\Delta t} = \frac{180^\circ}{3 \text{ min}} = \frac{1^\circ}{1 \text{ sec}} \tag{7.4}
\]
The Position Model

This physical location of an AUV is defined not only by its position in 3-space but also its attitude – specifically, its heading. The kinematic models described thus far shown how four of these parameters are computed; there is still a need to define how the AUV’s x-y planar position is determined. This is done by completing the second-order differential equations of motion by defining the relationship between the x and y position of an AUV’s with the AUV’s speed and heading. These equations are shown in Eq. 7.5 and in Eq. 7.6.

\[
\frac{\Delta x}{\Delta t} = \text{speed} \cdot \sin(\text{heading}) \tag{7.5}
\]

\[
\frac{\Delta x}{\Delta t} = \text{speed} \cdot \cos(\text{heading}) \tag{7.6}
\]

C3L Implementation of AUV Kinematic Models

The differential equations constituting the kinematic model of the AUVs are well suited for implementation in a discrete time simulation. To implement this, one needs only to periodically evaluate a set of expressions and update the current state of the model – exactly what C3L was designed to facilitate.

The prerequisite for any discrete time simulation is a mechanism to trigger the periodic updates of the model. Since C3L devices are designed to allow events to be raised to specific devices, it is simple to map the conceptual task of triggering updates of
the kinematic models to an event. It is also simple to map the mechanism that triggers these periodic to a C3L device. This is just what was done in this investigation. An independent device that consists of only a single control event was made to send events at fixed time intervals to all devices in the system. Any device that is interested in performing periodic functions may then implement a control event handler for this event.

The control event for this device is marked as `executeOnStart`, so that it is raised on application startup. This control event consists of a simple loop which repeats the number of times needed to complete the entire simulation. In each loop iteration, the `tic` event is raised to all devices, and then the implicit `wait` function is called to pause execution of this event the specified number of seconds. The complete C3L device definition for this ‘timekeeper’ device is shown in Figure 7-3.

```c3l
device SMTTimeKeeper
//a device to generate events at 1 second intervals
control event start executeOnStart {
    // control event handlers listed here
    trace("starting program appl.c3l");
    real simTime;
    // total sim time set for 14 hours = 840 minutes
    // = 50400 seconds
    for (simTime = 0 to 50400) {
        raise tic(simTime) direct all;
        //simulation time interval is 1 second
        wait(1.0);
    }
    stop();
}
end
```

Figure 7-3: C3L Simulation Time Keeper Device Definition – a device that raises events to all devices in prescribed intervals.
A C3L Plant consists of the state space variables, uncontrollable event handlers and controllable event handlers. Since this is a simulation investigation with no underlying hardware, there will be no uncontrollable events; consequently some small modifications will be made to the C3L implementation. For example, an actual physical implementation of an AUV would likely have an internal clock from which uncontrollable events periodically raised to indicate the passage of time. Consequently the plant model used here will have state variables and controllable event handlers.

AUV State Space for Kinematic Models and Control Systems

The state space required to support kinematic models of AUV motion and the control systems using those models consists primarily of the current physical state of the device (location, depth, heading, speed, acceleration, and rate of depth change), and the desired physical state (target speed, target heading, and target depth – i.e. the set points for the control systems.) There are also Boolean indicator variables used in the control system and available to the device controller as well as a set of constant model parameters. The C3L plant state space used in each AUV to implement the kinematic models and control systems is shown in Figure 7-4.
AUV Controllable Event Handlers for Kinematic Models and Control Systems

There are four C3L controllable event handlers defined to support the kinematic models describing the motion of an AUV and their control systems. The primary event

```c
//The AUV State Variables for Kinematics and Control
parameter
const real hr2sec = 2.77777777777778e-4; // /hr to /sec
const real maxSpeed = 3.8; // maximum forward speed, knots
const real maxAccel = 2.0; // maximum acceleration, 

// acceleration model parameter
const real alpha = log(0.003346)/maxSpeed;
// diving/surfacing model parameter
const real alphaD = 15*pi/180;
// +- variance in target speed acceptable
const real speedEps = 0.0075;
// heading change rate in radians/second
const real headingChangeRate = pi/180;
// turning radius in nm @ speed of 3.5 nm/hr
const real Rturn = 0.5413; // determined experimentally

// x-y-z position of the AUV in NM
real positionX = 0, positionY = 0,
    positionZ = 0,
    speed = 0, // forward speed in knots
    accel = 0, // forward acceleration in knots/sec
    heading = 0*pi/180, // heading in radians where
                      // hdg north := 0,
                      // hdg east := pi/2, etc
    targetSpeed = 0, // speed control system set point
    targetHeading = 0*pi/2, // heading set point
    targetDepth = 0, // depth set point

    // rate at which the AUV depth may change
    depthChangeRate = alphaD*speed;

boolean isAccelerating = false,
    isDecelerating = false,
    isTurning = false,
    isDiving = false,
    isSurfacing = false;
```

Figure 7-4: C3L Implementation of State Variables used in the AUV’s Kinematic Models and Control Systems
handler used to periodically update the kinematic model is the `updateNotionModel` event handler. This handler implements the complete control systems and kinematic models described previously. The other three controllable event handlers alter the state of the AUV plant by changing the set point for one of the three controlled degrees of freedom in the AUV – the forward speed, the heading, and the depth. The complete C3L implementations of these event handlers are illustrated in Figure 7-5.

**Performance of C3L Kinematic Models and Control Systems**

Before continuing with the development of the rest of the scenario, three test trials were conducted on the C3L implementation to verify both the conceptual model and the implementation. The purpose of the first test is to verify the performance of the more complicated kinematic model – the acceleration and deceleration model – and its control system. In this test, the AUV simply starts from rest, then accelerates to 3.5 knots. Shortly after reaching 3.5 knots, the AUV is to decelerate to 0.0 knots. The test was repeated for target final velocities of 1.0, 2.0, and 3.0 knots to demonstrate control system stability. A simple controller model was implemented to execute this state machine and to call the controllable event handler `setTargetSpeed` at the programmed times. The results of this experiment are shown in Figure 7-6 as a graph of the AUV’s speed in nm/hr versus time in seconds.

The second experiment was designed to include both acceleration and a change in heading, primarily to validate the heading control system. In this test, the AUV starts from rest, then accelerates to 3.5 knots on a heading of 0 degrees. When the target speed
is reached, the AUV is to execute a 90 degree turn to the right. When the 90 degree turn is completed, the simulation is terminates. In addition to demonstrating that the C3L control system implementation is capable of navigating a simple course, this test is intended to permit the determination of both the time and distance required to accelerate to 3.5 knots from rest for both classes of AUVs (the scanning/mapping AUVs and the Identification AUV.) The test was repeated for the Identification-class AUV for accelerations from 0 to 2.5 knots as well as 0 to 4.5 knots as in the application this AUV will have more variability in its speed. This test is also intended to permit the determination of the turning radius of the AUVs at this speed. Figure 7-7 illustrates the results of this experiment, plotting the position the AUV every second on the XY plane. The performance data of the two classes of AUVs in straight line acceleration and in turning are summarized in Table 7-2.

The third test was designed to exercise the heading control system through the full

<table>
<thead>
<tr>
<th>Table 7-2: AUV Performance Characteristics Determined Experimentally</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scanning-Mapping AUVs</strong></td>
</tr>
<tr>
<td>Time to Accelerate</td>
</tr>
<tr>
<td>0 to 4.5 knots</td>
</tr>
<tr>
<td>0 to 3.5 knots</td>
</tr>
<tr>
<td>0 to 2.5 knots</td>
</tr>
<tr>
<td>Distance to Accelerate</td>
</tr>
<tr>
<td>0 to 4.5 knots</td>
</tr>
<tr>
<td>0 to 3.5 knots</td>
</tr>
<tr>
<td>0 to 3.5 knots</td>
</tr>
<tr>
<td>Time to Turn 90 degrees</td>
</tr>
<tr>
<td>Turning Radius</td>
</tr>
<tr>
<td>@ 4.5 knots</td>
</tr>
<tr>
<td>@ 3.5 knots</td>
</tr>
<tr>
<td>@ 2.5 knots</td>
</tr>
</tbody>
</table>
360 degrees of heading. The heading control system is set by desired heading, so the
direction to turn to the desired heading is computed by the control system. This decision
is complicated by the ‘wrap-around’ nature of polar coordinates in which $90 + 1 = 91 = -
89$, etc… It seemed prudent to ensure that the logic of this decision algorithm was in fact
correct.

In this test, the AUV accelerates to 3.5 nm/hr starting on a heading of 0 degrees.
Upon reaching the target speed, the AUV turns to heading -45 degrees. When the turn is
complete, the AUV is to turn to a heading of 100 degrees, and then turn to 90 degrees.
Then, the AUV is to turn to a heading of -145 degrees. Finally, the AUV is to turn to a
heading of 80 degrees. Between turns, the AUV is to travel straight for 5 seconds.

The controller needed to execute this plan is a fairly simple finite state machine.
The C3L implementation of this machine is illustrated in Figure 7-8, and takes the form
of a C3L control event called tic that is only authorized when received from the
timekeeper device which raises the event once each second. Six states are used in the
implementation – one for each turn and one final state to terminate the simulation. The
five second straight travel segments are implemented as substates within the six states.
Note the use of the Boolean indicator variables from the plant in the conditions of the if
statements which implement the transitions of the state machine. If one recalls the
operation of the C3L plant, the state variables are visible in the controller, but they may
only be changed by plant events; consequently, to change any aspect of the plant state
space directly by the controller, controllable plant events handlers are used. The control
event implementation of Figure 7-8 exercises control of the lower-level plant control
systems via calls to setTargetHeading.
Figure 7-5: C3L Implementation of Controllable Event Handlers used in the AUV’s Kinematic Models and Control Systems
The plot of the AUV’s position over the course of this test is shown in Figure 7-9.

Note that the control system always chooses the turning direction to arrive at the desired heading in the shortest time. For example, when the heading is -145 degrees and the desired heading set to 80 degrees, the control system does not command a right turn; instead the control system turns the AUV to the left. Turning at 1 degree per second, the turn is complete in 115 seconds. Had the AUV turned right, the turn would have required 245 seconds to complete.
Figure 7-7: Acceleration and Turn Performance Test Results - Position of the AUV in the X-Y plane. Data markers show the position of the AUV each second. Units are nm.
//The control model for an AUV in the compound turn test memory
tic(real t) authorize SIMTimekeeper {
    updateMotionModel;
    if (state == 0 and not is Accelerating) {
        setTargetHeading(-45*pi/180); // turn to -45 deg
        state = 1;
    } else if (state == 1 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            setTargetHeading(100*pi/180); // turn to 100 deg
            state = 2; waypoint = 0;
        }
    } else if (state == 2 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            setTargetHeading(90*pi/180); // turn to 90 deg
            state = 3; waypoint = 0;
        }
    } else if (state == 3 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            setTargetHeading(-145*pi/180); // turn to -145 deg
            state = 4; waypoint = 0;
        }
    } else if (state == 4 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            setTargetHeading(80*pi/180); // turn to 80 deg
            state = 5; waypoint = 0;
        }
    } else if (state == 5 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            stop();
        }
    } else if (state == 0 and not is Accelerating) {
        setTargetHeading(-45*pi/180); // turn to -45 deg
        state = 1;
    } else if (state == 1 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            setTargetHeading(100*pi/180); // turn to 100 deg
            state = 2; waypoint = 0;
        }
    } else if (state == 2 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            setTargetHeading(90*pi/180); // turn to 90 deg
            state = 3; waypoint = 0;
        }
    } else if (state == 3 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            setTargetHeading(-145*pi/180); // turn to -145 deg
            state = 4; waypoint = 0;
        }
    } else if (state == 4 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            setTargetHeading(80*pi/180); // turn to 80 deg
            state = 5; waypoint = 0;
        }
    } else if (state == 5 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            stop();
        }
    } else if (state == 0 and not is Accelerating) {
        setTargetHeading(-45*pi/180); // turn to -45 deg
        state = 1;
    } else if (state == 1 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            setTargetHeading(100*pi/180); // turn to 100 deg
            state = 2; waypoint = 0;
        }
    } else if (state == 2 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            setTargetHeading(90*pi/180); // turn to 90 deg
            state = 3; waypoint = 0;
        }
    } else if (state == 3 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            setTargetHeading(-145*pi/180); // turn to -145 deg
            state = 4; waypoint = 0;
        }
    } else if (state == 4 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            setTargetHeading(80*pi/180); // turn to 80 deg
            state = 5; waypoint = 0;
        }
    } else if (state == 5 and not isTurning) {
        waypoint = waypoint + 1;
        if (waypoint == 5) {
            stop();
        }
    }
}

Figure 7-8: The C3L Implementation of the Controller to Execute the Multiple Turn Test.
High-Level Scanning and Identification Modeling

With the low-level functionality of the AUV plants defined, tested, and with performance characteristics of the behavioral models quantified, the next step in this MCM distributed application development is the definition of the higher-level mission command and control for the two distinct MCM missions – the scanning mission for the
two AUVs and the identification mission for the third. As these missions are operationally quite distinct, they will be discussed individually in the following sections.

**Scanning AUV Mission Controller**

The MCM scanning and mapping missions assigned to two AUVs in this scenario are characterized by the need to systematically scan the assigned regions of the search area, reporting any detected Mine-Like Contacts (MLCs). Since the simplifying assumptions for this scenario neglect significant real-world factors such as current, wind, and system failure, the kinematic motion models and control system behaviors derived describing the AUVs movements are sufficient to plan a course via dead reckoning, which assumes that the actual position is what one would compute via course, speed, heading and time computations. These computations easily yield a state machine which will navigate an AUV over its assigned sub-regions. From this state machine, a C3L implementation of the state machine was created in the form of a control event handler for the event ‘tic’ – the event raised every second by the SIMTimekeeper device. This permits the periodic evaluation of state and potential state transitions as dictated by the rules of the state machine. The first portion of the implementation of this control event handler for AUV1 is illustrated in Figure 7-10. The control event begins by calling the controllable plant event handler `updateMotionModel` which updates the kinematic models and motion control systems as described previously. Next, there is some selectively executed code which sends position reports to an external device called LOGGER at varying intervals depending on the turning state of the AUV. If the AUV is turning,
/The control model for AUV1 in the MCM Scenario

memory
integer state = 0, count = 10, totalContacts = 0;
boolean startPosSet = false; real startx, starty;

control
event tic(real t) authorize SIMTimekeeper {
    updateMotionModel;
    if (isTurning) {
        if (count < 10) // when turning, log pos every 10 secs
            count = count + 1;
        else {
            raise log(1, positionX, positionY, positionZ) direct LOGGER;
            count = 0;
        }
    } else {
        if (count<600) // when not turning, log pos every 10 mins
            count = count + 1;
        else {
            raise log(1, positionX, positionY, positionZ) direct LOGGER;
            count = 0;
        }
    }
    real dx, dy, dist;
    if (state == 0) { // init - accelerate to 3.5 due East
        setTargetDepth(-0.002);
        setTargetSpeed(3.5);
        setTargetHeading(90*pi/180);
        state = 1;
    } else if (state == 1 and not isAccelerating) {
        setTargetHeading(180*pi/180); // turn south
        state = 2;
    } else if (state == 2 and not isTurning) { // turn East
        setTargetHeading(90*pi/180);
        state = 3;
    } else if (state == 3 and not isTurning) { // travel 0.041831 nm straight, then turn North
        if (startPosSet) {
            dx = positionX - startx; dy = positionY - starty;
            if ( sqrt( dx*dx + dy*dy ) >= 0.041831 ) {
                startPosSet = false;
                setTargetHeading(0*pi/180); state = 4;
            }
        } else {
            startPosSet = true;
            startx = positionX;
            startY = positionY;
            state = 4;
        }
    } else if (state == 4 and not isTurning) { . . .

Figure 7-10: The First Portion of the C3L Implementation of AUV1’s High-level Mission Controller (for the complete implementation, see Appendix A)
position reports are sent every 10 seconds; otherwise, position reports are sent every 10 minutes. The LOGGER device simply receives position reports and contact reports from the AUVs, maintains the current position for each AUV, and when any report is received, generates structured output of the simulation.

After the AUV control event has updated the motion model and generated logging messages if appropriate, the state machine is implemented. The initial state sets the initial heading, speed, and depth for the mission. For this AUV in this mission, the initial heading is due East and the speed is set for a constant 3.5 Knots. This state always transits to the next state, and when the AUV is no longer accelerating, the AUV executes a turn to the South. The next state transition occurs when the turn south is completed, then a turn to the East is commanded. When the turn Eastward is completed, the AUV is to execute a straight course 0.041831 nm long. Once the distance is reached, a turn Northward is commanded. This positions the AUV for the initial scanning path to run from <0.25, 0> nm to <0.25,6.0> nm. The state machine implementation continues in this way, executing the six scanning passes over the proper assigned areas. After all six scanning passes are completed, the AUV is turned back to the starting point (<0, 0> nm), and when it reaches that point it signals it is signing off. The second scanning AUV is implemented similarly, which small changes made in the state machine to direct the AUV to navigate the other half of the scanning area.

This state machine also includes logic to identify the portions of the programmed path during which the side scan sonar is to be active. To simulate this, the tic event handler raises a scan event when the sonar is active. The scan event handler simulates the detection of MLCs using a Poisson random variable with a mean computed to have on
average 27 MLCs over the 36 nm² region. The random variable is the number of MLCs detected in the scan; for each MLC detected, a uniformly distributed random range from the AUV is generated from which to compute the coordinates of the MLC. Then the MLC is reported to AUV3. The complete implementation of the scan event handler (which is the same for both scanning AUVs) is illustrated in Figure 7-11.

Identification AUV Mission Controller

The implementation of the mission controller for the identification AUV is rather distinct from that of the scanning AUVs as the missions are rather distinct. The scanning

```c
//The control model for AUV1 in the MCM Scenario

event scan {
    // simulate swath scanning of 0.25 nm by distance
    // traveled in 1 second @3.5 knots
    // assuming there are 27 mine-like contacts (MLC) in
    // the field distributed uniformly
    integer numContacts = PoissonRV(3.6467e-4);
    // handle the number of contacts up to 100
    while (numContacts > 0, 100) {
        // a MLC detected
        real range = randReal2(0.0,0.25),x,y;
        if ( coinFlip() ) { // check what side the contact was made... equal prob either side
            x = positionX + range * cos(heading);
            y = positionY - range * sin(heading);
        } else {
            x = positionX - range * cos(heading);
            y = positionY + range * sin(heading);
        }
        raise contact(x,y) direct AUV3;
        numContacts = numContacts - 1;
        totalContacts = totalContacts + 1;
    }
}
```

Figure 7-11: The C3L Implementation of the scan Control Event – the event handler that simulates detection of MLCs.
AUVs are required to navigate a fixed path, while the identification path cannot be known until the mission is in progress. The methodology of the identification mission used in this investigation is as follows: The identification AUV is to wait for reports of MLCs from the scanning AUVs. When contact reports are received, they are to be visited in the order in which they are received. No path optimization is implemented at this stage. When there are no unvisited MLCs to inspect, the AUV is to simply stop and wait for reports. When all the reported MLCs have been inspected and the two scanning AUVs have reported that they are done, then the identification AUV is to return to base. Upon reaching base, the MCM mission is considered complete. The state machine describing this algorithm is illustrated in Figure 7-12. The C3L implementation of this state machine is shown in Figure 7-13.

The identification process is simulated by having the AUV slow to 2.0 knots when the range to the MLC is less than the maximum range of the AUV’s sonar. Once this is accomplished, the AUV will either stop or turn to the next reported MLC location if an unvisited contact is available.

**Performance Evaluation of MCM Scenario**

The MCM mission described here and its implementation were compiled and executed to show not only the capabilities of the C3L compiler, but also the utility of the C3L language to enable a straightforward implementation of a realistic simulation of a
real-world application. The C3L application was run using ‘time compression’ techniques inherent to event-based simulators so that the nearly 13.4 hour-long MCM mission could be simulated by the compiled C3L program in only 118 seconds the evaluation platform – a Dell Optiplex 755 with Intel Core 2 Duo E6850 CPU 3.00 GHz. with 3.00 GB Ram. The operating system is 32-bit Windows XP Professional Version 2002 with Service Pack 3. The E6850 CPU features a bus speed of 1.333GHz and a 4 MB Advance Smart Cache operating at 3 GHz. The L1 cache consists of a 32 KB instruction cache and a 32 KB data cache per core. The result of a run of the simulation is illustrated in Figure 7-14.

Figure 7-12: State Machine for the Identification AUV’s Mission Controller.
The tracks of the three AUVs are shown along with the locations of the MLCs detected. Remember that the identification AUV is visiting the MLCs in the order in which they are detected; consequently its path is somewhat convoluted. However the scanning AUVs, following their pre-programmed paths, track exactly as expected. It is interesting to note that, despite the simple scheduling algorithm implemented by the identification AUV, because of its higher travel speed and the particular location of the mines simulated in this instance, the identification AUV actually returns to base 0.58 hours before AUV2. The scenario was constructed such that the number of MLCs in the field would average 27; in this instance of the scenario there were 26 MLCs detected.

To estimate how ‘busy’ the identification AUV was in the scenario, it was instrumented to count the number of time intervals that it had at least one active target to pursue. This number, divided by the total time of its mission, is a measure of its busy time. In this scenario, the identification AUV had at least one active (i.e. unvisited) target 65.435% of the time. The average number of active targets over the identification AUV’s mission was 1.0493. This implies that, even if a more sophisticated scheduling algorithm used to decide an alternative ordering for visiting the MLCs, there may not be many options that actually improve the overall time spent giving this scenario. However, this does not mean that other objectives – such as minimizing energy – would not be worth pursuing. It is quite obvious by inspection that the path traversed by the identification AUV is not minimal, it is only the temporal ordering of the reporting of MLCs in this particular instance of the scenario that permits AUV3 to finish before AUV2. Also, the performance recorded in this scenario is heavily dependent on many complex, interacting factors; obvious ones include the density of MLCs in the field and the speed of the
relative speeds of the AUVs… It is not difficult to image that if the speed of the scanning AUVs was approximately equal to or greater than that of the identification AUV, or if the density of the MLCs increased, that the workload on the identification AUV could rapidly increase. To illustrate this, the scenario was re-run with the only change being a reduction of the travel speed of the identification AUV from 4.5 knots to 3.5 knots – equal to that of the scanning AUVs. This one change was sufficient to cause the identification AUV to complete 0.6678 hours after AUV2 – a difference of more than one hour. Additionally, this reduction is the travel speed of AUV3 resulted in an increase of the average number of unvisited targets from 1.0439 to 1.62977. The scenario was run an additional time with the identification AUV travelling at 3.5 knots, but this time the expected number of MLCs in the field was increase 25% from 27 to 33. In this trial, 30 MLCs were detected, and though the average number of unvisited targets rose to 1.82158, AUV3 still returned to base before AUV2 because of where the last detections happened to be in this particular instance of the scenario.

Summary and Future Work

The purpose of this chapter was to outline how C3L supports the model-based development of complex distributed system using a sufficiently complex application to be meaningful while not getting lost in excessively complexity. In looking back through the chapter, it is interesting to note that most of the discussion is not about C3L – instead, most of the discussion centers on the specific application under investigation and the models that were developed to enable this investigation. And this is the key point – the
C3L plant and controller model were developed from the natural and intuitive architectures of real-world devices and entities, and in general, interactions between them are asynchronous and event-driven. The primary benefit of using C3L is to enable the development of complex, concurrent, interacting distributed systems while freeing the programmer from the esoteric details that such systems frequently entail. The intention is to bring truly distributed, concurrent programming into the reach of mainstream programmers.

There are many directions one might pursue from this point. The heavily state-transition based nature of the control systems implemented in this rather simple application imply that improved language support for state machines might be a powerful addition to the language. There are certainly examples of state-machine integration into languages from which to draw – particular in the field of hardware definition languages. There are also many ways in which the simulation study itself could be enhanced, including optimizing the trajectory of the identification AUV, increasing the scale of the scenario in terms of number of AUVs, interaction with other MCM platforms and sensors, and more realistic restrictions on the abilities of AUVs to communicate.

From a language point of view, it would be more interesting to implement C3L on real robotic hardware – perhaps not an AUV, but a small robot capable of supporting a hardware and operating system combinations supported by the Phoenix compiler framework. Demonstrating this step would complete the link from high-level conceptual modeling to actual hardware implementation that C3L was designed to enable.
The control model for AUV3 in the MCM Scenario:

```c
// The control model for AUV3 in the MCM Scenario
event tic(real t) authorize SIMTimekeeper {
    // logging functionality not shown here (same as before)
    updateMotionModel;
    if (state == 0) { // initial state
        raise log(3, positionX, positionY, positionZ) direct LOGGER;
        setTargetDepth(0.002); state = 1;
    } else if (state == 1 and numTargets <= currentTarget and numActiveAUVs == 0) { state = 4;
    } else if (state == 1 and numTargets > currentTarget) {
        // waiting for contact state
        setTargetSpeed(travelSpeed); state = 2;
    } else if (state == 2) { // travel to contact
        dx = targetX[currentTarget] - positionX;
        dy = targetY[currentTarget] - positionY;
        setTargetHeading( atan2(dx, dy) );
        if (sqrt(dx*dx + dy*dy) <= IDrange ) {
            setTargetSpeed(opSpeed); state = 3;
        }
    } else if (state == 3) {
        raise contactReport(2, currentTarget+1,
            targetX[currentTarget], targetY[currentTarget])
            direct LOGGER;
        currentTarget = currentTarget + 1; state = 1;
        if (numTargets <= currentTarget)
            setTargetSpeed(0.0); // stop if there is no target avail
    } else if (state == 4) { // RTB
        setTargetSpeed(travelSpeed);
        dx = 0 - positionX; dy = 0 - positionY;
        setTargetHeading( atan2(dx, dy) );
        if (sqrt(dx*dx + dy*dy) <= 0.05 ) {
            setTargetSpeed(0.0); state = 5;
        }
    } else if (state == 5 and not isDecelerating) { // done
        raise log(3, positionX, positionY, positionZ) direct LOGGER;
        state = 6;
    }
}
```

Figure 7-13: C3L Implementation of the State Machine for the Identification AUV’s Mission Controller.
Figure 7-14: AUV Trajectories and Locations of MLCs in MCM Mission Simulation
Chapter 8

Conclusions

The Control, Communication, and Computation Language (C3L) creates a language-level abstraction of a control methodology developed for the reactive control of distributed systems in which control and communication are unified in a single abstraction embedded into the language. This frees the programmer from details about how to communicate events between devices and enabling him or her to focus exclusively on the behavioral description of devices and the larger-scale behaviors of the distributed system in which those devices operate. C3L is presented here as a formal language with well-defined syntax and a structural operational semantics. These formalisms define how the language is to behave under all possible situations and provide a system through which one may reason about all programs expressed in the language. Using these formalisms, it was shown that C3L exhibits fairness with respect to events and devices under C3L control are prefixed-closed controllable. From this analytical foundation, a compiler implementation based on the Microsoft Phoenix framework for compiler design and optimization was developed that exploits the natural task-level parallelism inherent in C3L programs to automatically generate multi-threaded executables for any C3L program. The generation of these multi-threaded executables is accomplished without any direct actions of the programmer, and the formal properties of the language and the architecture of the compiler were used to show that the executables generated by the compiler are guaranteed to be free from several significant pitfalls in
concurrent programming, including deadlock and livelock. Furthermore, executables generated by the compiler are also guaranteed to be exhibit event fairness and not leave a device starving for events. A set of benchmarking performance tests demonstrate that near-linear speeds are realizable when C3L executables are run on Multicore platforms when computation dominates the complexity of a C3L program. Tools for the support of simulations, visualizations, and stochastic algorithms were integrated into the language and the compiler. Finally, a significant application study was presented to demonstrate how the C3L programming paradigm permits the programmer to indeed focus on the problem domain, not the programming domain.

Research Directions

Given the scope of a formal language for such a broad classification of programs like concurrent, distributed system development, there are many research directions available. The application study illustrated the utility of enhanced language support for the expression of state machines which are sure to be a staple of C3L programming. The C3L plant model is itself a state-transition definition; consequently there are known graphical methods of expressing such models that would be well-suited to support the generation of C3L code from a graphical model. Additionally, anyone experienced in Object-Oriented Programming (OOP) will immediate miss the capability of defining classes and other higher-order data types currently lacking in C3L. C3L was designed to support higher-order expression of programs, but its lack of support for higher-order data types is a heavy burden that was felt even in the relatively simple application study of
Chapter 7. C3L devices are like instances of a class object in OOP yet differ in that each C3L device must be constructed individually. In a case where there are an arbitrary number of devices performing exactly the same tasks, the current instantiation of C3L requires that each device be defined individually with only its identifier changed. It would be very useful to transit C3L device definitions to a class-based paradigm using constructors to customize the particular details of an otherwise standard class of devices.

In terms of the C3L compiler, there is a broad spectrum of opportunities to pursue. The compiler currently implements one thread per device; experienced thread programmers know that it is unlikely that this particular instantiation of multithreading of C3L executables is optimal. Does the complexity of a particular device warrant the expense of a dedicated thread? In complex devices, would more than one threads be useful? What are the issues of concurrent execution within a device? There are clearly many questions to be answered as the language moves forward.

Most importantly, an implementation of the language on actual robotic hardware should be demonstrated. Though purely simulation studies are useful and can yield important results, the full value of C3L as a programming language designed for distributed system development can only be evaluated when finally it is implemented on a hardware system. The Microsoft Phoenix framework makes this a realistic goal, yet there are still issues to address. Most significantly of these involve the mapping of hardware signals to uncontrollable events and the binding of state variables in a device’s plant to measure and/or controlled locations in hardware. For example, if the x, y, and z position variables are mapped to an inertial navigation unit’s (INU’s) output of the corresponding data, how would these values be handled by the language? Is it sufficient
to simply treat them as volatile values? Should such data be managed strictly by the
language – that is, should the INU output periodic uncontrollable events to the device and
just let device do what it will with the data? Beyond the conceptual management of such
data, how are the relationships between the physical hardware and the C3L plant model
to be described? Should there be a definition of a configuration file through which to
express these relations? Should a formal synchronous language like Esterel, a hardware
description language like Verilog or VHDL, or some other language be integrated with
C3L to enable a functional description of the actual links between the hardware and the
software domains?

There is a clear opportunity with a language like C3L to pursue a more formal
approach to the trendy hardware-software co-design paradigm in which the line between
the hardware and the software is blurred and systems are developed, not hardware
developed for software or the converse. It would be interesting to integrate Verilog-like
syntax for the description of hardware signals and components with modern compiler
technology in a unified, formalized framework targeted at embedded system
development. In this model, the system would include descriptions not only a piece of
software and the hardware on which it executes, but also the interfaces between the two.
The value of such an approach is clear and the subject of active research by many in the
field. However, the complexities of formalizing any individual component of these
systems (the hardware, the software, the interface, etc.) are non-trivial and still open
research questions. The complexity of formally describing the integrated ‘big picture’ is
arguably exponentially more complex. Nonetheless, hardware-software co-design is an
active area of research. COMET [Case2008] is a hardware-software co-design language
using C and VHDL for the software and hardware descriptions, respectively. POLIS [Berkeley2008] was developed by the embedded systems group at the University of California, Berkeley which pursues a finite-state machine approach to hardware-software co-design in a unified framework. This approach is similar to what could be accomplished via an integration of the asynchronous C3L and a synchronous language like Esterel or Verilog to describe the hardware and its binding to the C3L plant model. Such an organization would be classified as the increasingly popular Globally Asynchronous, Locally Synchronous designed paradigm – a paradigm that is growing in interest because the complexity of systems is growing such that the global synchrony assumption is becoming impossible to maintain even for modern hardware designs like those of today’s Multicore processors. For decades, asynchrony has been avoided as a means for formal verification of systems because of its complexity and limitations; it seems as though asynchrony at some level will no longer be avoidable. The hope is that at some point, a language like C3L that embraces the realities of asynchrony in system development will strike the proper balance between formality, utility and accessibility and enable truly mainstream distributed system programming and development.
Bibliography


Case Western Reserve University, Department of Electrical Engineering and Computer Science Hardware-Software Co-Design Group (2008) [http://bear.ces.case.edu/hard_soft.html]


Cau, A., Moszkowski, B., Zedan, H. Interval Temporal Logic. (online resource) 2008 [http://www.cse.dmu.ac.uk/STRL/ITL/]


Church, Alonzo. The Calculi of Lambda-Conversion. Princeton University Press. 1941


Clarke, E.M., Grumberg, O, Peled, D.A. Model Checking. MIT Press. 1999


Cooper, K.D. and Torczon, L. Engineering a Compiler. Morgan Kaufmann. 2004


Dybvig, R. Kent. The Scheme Programming Language. MIT Press. 2003


Grove, D., DeFouw, G., Dean, J., Chambers, C. “Call Graph Construction in Object-Oriented Languages.” *SIGPLAN* 32.10 (1997): 108-124


Harrison, Michael A. *Introduction to Formal Language Theory*, Addison-Wesley (1978)


Hecht, Matthew S. *Flow Analysis of Computer Programs*. Elsevier North-Holland Inc. 1977


Hennessy, Matthew. *A Distributed Pi-Calculus*. Cambridge University Press. 2007

Hewitt, Carl, Bishop, Peter, Steiger, Richard. “A Universal Modular Actor Formalism for Artificial Intelligence.” *Proceedings of the Third International Joint Conferences on Artificial Intelligence*. 1973


Hughes, Cameron. Parallel and Distributed Programming using C++. Addison-Wesley. 2004

Karlsson, Bjorn. Beyond the C++ Standard Library: An Introduction to Boost. Addison-Wesley. 2006


Krishnamurthi, Shriram. Programming Languages: Application and Interpretation. 2003


Phoha, S., Eberbach, E., Brooks, R.R. "Coordination of Multiple Heterogeneous Autonomous Undersea Vehicles (AUVs)." Autonomous Robots (2000)


Pierce, Benjamin. Types and Programming Languages. MIT Press. 2002


Quinn, Michael J. Parallel Programming in C with MPI and OpenMP. McGraw-Hill. 2004

Rescher, N. and Urquhart, A. Temporal Logic. Springer-Verlag. 1971


Reynolds, John C. Theories of Programming Languages. Cambridge University Press. 1998


Schroeder, W., Martin, K., Lorensen, B. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics (Fourth Edition)*. Kitware Inc. 2006


Scott, M. *Programming Language Pragmatics*. Morgan Kaufmann. 2005


The Visualization Tool Kit. February 2008 [http://www.vtk.org]


Wirth, N. *Compiler Construction*. Addison-Wesley. 1996
Appendix A

C3L Application Programs

TestProg4.c3l – ‘Computation-bound’ Benchmarking Compiler Test

// testProg4.c3l
device AUV1

memory
set dest = {2};
integer count = 0;

control
event delayedStart executeOnStart {
    raise hello direct all;
}

event hello authorize all {
    // trace("One");
    count = count + 1;
    integer i = 0; real r = 0,f;
    for ( i = 0 to 25000 ) {
        f = sqrt(sin(0.79*r*i)*cos(r/1.2+i));
        r = r + pi/1000;
    }
    if ( count < 50000 ) {
        raise hello direct dest;
    } else {
        trace("Done!
");
        stop();
    }
}

end // device AUV1

//******************************************************************************

device AUV2

memory
set dest = {3};

control
event hello authorize all {
    // trace("Two");
    integer i = 0; real r = 0,f;
    for ( i = 0 to 25000 ) {
        f = sqrt(sin(0.79*r*i)*cos(r/1.2+i));
    }
}
r = r + pi/1000;
} else {

raise hello direct dest;
}
}
}
end // device AUV2

//***********************************************
device AUV3
memory
set dest = {4};

control
event hello authorize all {
    // trace("Three");
    integer i = 0; real r = 0, f;
    for (i = 0 to 25000) {
         f = sqrt(sin(0.79*r*i)*cos(r/1.2+i));
         r = r + pi/1000;
    }
    raise hello direct dest;
}
}
end // device AUV3

//***********************************************
device AUV4
memory
set dest = {5};

control
event hello authorize all {
    // trace("Four");
    integer i = 0; real r = 0, f;
    for (i = 0 to 25000) {
         f = sqrt(sin(0.79*r*i)*cos(r/1.2+i));
         r = r + pi/1000;
    }
    raise hello direct dest;
}
}
end // device AUV4

//***********************************************
device AUV5
memory
set dest = {6};

control
event hello authorize all {
    // trace("Five");
    integer i = 0; real r = 0, f;
    for (i = 0 to 25000) {

f = sqrt(sin(0.79*r*i)*cos(r/1.2+i));
    r = r + π/1000;
}
raise hello direct dest;

end // device AUV5

//***********************************************
device AUV6

memory
set dest = {7};

control
    event hello authorize all {
        // trace("Six");
        integer i = 0; real r = 0,f;
        for ( i = 0 to 25000 ) {
            f = sqrt(sin(0.79*r*i)*cos(r/1.2+i));
            r = r + π/1000;
        }
        raise hello direct dest;
    }

end // device AUV6

//***********************************************
device AUV7

memory
set dest = {1};

control
    event hello authorize all {
        // trace("Seven");
        integer i = 0; real r = 0,f;
        for ( i = 0 to 25000 ) {
            f = sqrt(sin(0.79*r*i)*cos(r/1.2+i));
            r = r + π/1000;
        }
        raise hello direct dest;
    }

end // device AUV7

//***********************************************

TestProg5.c3l – ‘Communication-bound, higher messaging complexity’
Benchmarking Compiler Test

// testProg5.c3l
device AUV1

memory
set dest = {2};
integer count = 0;

control
event delayedStart executeOnStart {
   raise hello direct all;
}

event hello authorize all {
   // trace("One");
   count = count + 1;
   if ( count < 50000) {
      raise hello direct dest;
   } else {
      trace("Done!
");
      stop();
   }
}

end // device AUV1

//********************************************************************

device AUV2

memory
set dest = {3};

control
event hello authorize all {
   // trace("Two");
   raise hello direct dest;
}

end // device AUV2

//********************************************************************

device AUV3

memory
set dest = {4};

control
event hello authorize all {
   // trace("Three");
   raise hello direct dest;
}

end // device AUV3

//********************************************************************

device AUV4
memory
set dest = {5};

control
event hello authorize all {
    // trace("Four");
    raise hello direct dest;
}
end // device AUV4

device AUV5

memory
set dest = {6};

control
event hello authorize all {
    // trace("Five");
    raise hello direct dest;
}
end // device AUV5

device AUV6

memory
set dest = {7};

control
event hello authorize all {
    // trace("Six");
    raise hello direct dest;
}
end // device AUV6

device AUV7

memory
set dest = {1};

control
event hello authorize all {
    // trace("Seven");
    raise hello direct dest;
}
end // device AUV7
// testProg6.c3l

device AUV1

memory
set dest = {2};
in
teger count = 0;

c

control
event delayedStart executeOnStart {
   raise hello direct me;
}

event hello authorize all {
   // trace("One");
   count = count + 1;
   if ( count < 50000) {
      raise hello direct dest;
   } else {
      trace("Done!
");
      stop();
   }
}

end // device AUV1

//**********************************************

device AUV2

memory
set dest = {3};

c

control
event hello authorize all {
   // trace("Two");
   raise hello direct dest;
}

end // device AUV2

//**********************************************

device AUV3

memory
set dest = {4};

c

control
event hello authorize all {
   // trace("Three");
   raise hello direct dest;
}
end // device AUV3

device AUV4

memory
set dest = {5};

control
event hello authorize all {
    // trace("Four");
    raise hello direct dest;
}

end // device AUV4

device AUV5

memory
set dest = {6};

control
event hello authorize all {
    // trace("Five");
    raise hello direct dest;
}

end // device AUV5

device AUV6

memory
set dest = {7};

control
event hello authorize all {
    // trace("Six");
    raise hello direct dest;
}

end // device AUV6

device AUV7

memory
set dest = {1};

control
event hello authorize all {
    // trace("Seven");
    raise hello direct dest;
}
app1.c3l – The MCM Scenario Simulation

// app1.c3l
device AUV1,AUV2,AUV3;

//**********************************************
device SIMTimekeeper
memory
real simTime;
control
event start executeOnStart {
    raise tic(simTime) direct all;
    // total sim time set for 14 hours = 840 minutes = 50400 seconds
    if (simTime < 50400) {
        simTime = simTime + 1;
        // wait(1.0); //simulation time interval is 1 second
        raise start;
    }
}
end // device SIMController

//**********************************************
device LOGGER
memory
real auv1_x, auv1_y, auv1_z,
    auv2_x, auv2_y, auv2_z,
    auv3_x, auv3_y, auv3_z,
    simTime;
control
event tic(real t) authorize SIMTimekeeper {
    simTime = t;
}
event log(integer i, real x, real y, real z) authorize AUV1 union AUV2 union AUV3 {
    if (i == 1) {
        auv1_x = x; auv1_y = y; auv1_z = z;
    } else if (i == 2) {
        auv2_x = x; auv2_y = y; auv2_z = z;
    } else if (i == 3) {
        auv3_x = x; auv3_y = y; auv3_z = z;
    } trace("%f : %f %f %f : %f %f %f : %f %f %f", simTime, auv1_x, auv1_y, auv1_z, auv2_x, auv2_y, auv2_z, auv3_x, auv3_y, auv3_z);
}
event contactReport(integer type, integer ID, real x, real y) authorize AUV3 {
    if (type == 1) { // contact reported
        trace("%f : %f %f %f : %f %f %f : %f %f %f : Contact %d Reported @ %f %f", simTime, auv1_x, auv1_y, auv1_z, auv2_x, auv2_y, auv2_z, auv3_x, auv3_y, auv3_z, ID, x, y );
    } else if (type == 2) { // contact visited

trace("%f : %f %f %f : %f %f %f : Contact %d
visited @ %f %f", simTime, auv1_x, auv1_y, auv1_z, auv2_x, auv2_y, auv2_z,
    auv3_x, auv3_y, auv3_z, ID, x, y);
}
}
end // LOGGER

//*********************************************************************
device AUV1

//the plant

parameter
const real hr2sec = 2.77777777777778e-4;
// factor to convert from /hr to /sec
const real maxSpeed = 3.8;     //maximum forward speed in
knots
const real maxAccel = 2.0;     //maximum acceleration in
knots/sec
const real alpha = log(0.003346)/maxSpeed;  //acceleration model
parameter
const real alphaD = 15*pi/180;  //diving/surfacing model
parameter
const real speedEps = 0.0075;  //+- variance in target speed
acceptable
const real headingChangeRate = pi/180;
//heading change rate in radians/second (pi radians/3 minute =
// 180 degrees/3 minutes = pi radians/360 seconds
const real Rturn = 0.5413;
// turning radius in nm @ speed of 3.5 knots-determined experimentally

real positionX = 0, positionY = 0, //x-y-z position of the AUV in NM
    positionZ = 0,
    speed = 0,     //forward speed in
knots
    accel = 0,     //forward acceleration
in knots/sec
    heading = 90*pi/180,
//heading in radians where hdg north := 0, hdg east := pi/2, etc.
    targetSpeed = speed,  //speed which the AUV should
maintain
    targetHeading = heading,  //heading which the AUV
should maintain.
    targetDepth = positionZ,  //depth which the AUV
should maintain.
    depthChangeRate = alphaD*speed;
//rate at which the AUV depth may change

boolean isAccelerating = false,
    isDecelerating = false,
    isTurning = false,
    isDiving = false,
    isSurfacing = false;

controllable
updateMotionModel {
    //update position
    positionX = positionX + hr2sec*speed*sin(heading);
    positionY = positionY + hr2sec*speed*cos(heading);

    //update speed
    speed = speed + accel;
isAccelerating = speed < targetSpeed - speedEps;
isDecelerating = speed > targetSpeed + speedEps;
if (isAccelerating)
    accel = maxAccel*exp(alpha*speed);
else if (isDecelerating)
    accel = -0.5*maxAccel*(speed-targetSpeed)/maxSpeed;
else
    accel = 0;

//update heading
real dh = fabs(heading - targetHeading);
isTurning = dh > headingChangeRate + 0.001;
if (isTurning)
    if (heading < targetHeading and dh <= pi or heading >
targetHeading and dh > pi)
        //turn right
        heading = heading + headingChangeRate;
    else
        //turn left
        heading = heading - headingChangeRate;
else
    heading = targetHeading;
if (heading > pi) heading = heading - 2*pi; if (heading < -pi)
    heading = heading + 2*pi;

//update depth
depthChangeRate = alphaD*speed;
isDiving = positionZ < targetDepth - (depthChangeRate + 0.001);
isSurfacing = positionZ > targetDepth + (depthChangeRate +
0.001);
if (isDiving)
    positionZ = positionZ - depthChangeRate;
else if (isSurfacing)
    positionZ = positionZ + depthChangeRate;
else
    positionZ = targetDepth;

setTargetSpeed(real s) precondition (s >= 0 and s <= maxSpeed) {
    targetSpeed = s;
}
setTargetHeading(real h) {
    targetHeading = h;
}
setTargetDepth(real d) precondition (d <= 0) {
    targetDepth = d;
}

//the controller
memory
integer state = 0, count = 10, totalContacts = 0;
boolean startPosSet = false;
real startX, startY;
control
event tic(real t) authorize SIMTimekeeper {
    updateMotionModel;
    if (isTurning) {
        if (count < 10) // when turning, log position every 10 secs
            count = count + 1;
        else {
            raise log(1,positionX,positionY,positionZ) direct
        }
    }

LOGGER;
count = 0;

} else {
    if (count < 600) // when not turning, log position every 10
        minutes
        count = count + 1;
    else {
        raise log(1,positionX,positionY,positionZ) direct
        LOGGER;
        count = 0;
    }
}

real dx,dy,dist;
if (state == 0) { // initialization - accelerate to 3.5 due East
    setTargetDepth(-0.002);
    setTargetSpeed(3.5);
    setTargetHeading(90*pi/180);
    state = 1;
} else if (state == 1 and not isAccelerating) { // turn South
    setTargetHeading(180*pi/180);
    state = 2;
} else if (state == 2 and not isTurning) { // turn East
    setTargetHeading(90*pi/180);
    state = 3;
} else if (state == 3 and not isTurning) {
    // travel 0.041831 nm straight, then turn North
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        if ( sqrt( dx*dx + dy*dy ) >= 0.041831 ) {
            startPosSet = false;
            setTargetHeading(0*pi/180);
            state = 4;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
} else if (state == 4 and not isTurning) {
    // travel 6.054133 nm straight, then turn east.
    //Start scanning after 0.054133 nm
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        if ( dist >= 0.054133) {
            raise scan;
            if ( dist >= 6.054133 ) {
                startPosSet = false;
                setTargetHeading(90*pi/180);
                state = 5;
            }
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
} else if (state == 5 and not isTurning) {
    // travel 0.391734 nm straight, then turn south.
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
dist = sqrt(dx*dx + dy*dy);
if ( dist >= 0.391734 ) {
    startPosSet = false;
    setTargetHeading(180*pi/180);
    state = 6;
}
} else {
    // mark start of this distance leg
    startX = positionX;
    startY = positionY;
    startPosSet = true;
} }
else if (state == 6 and not isTurning) {
    // travel 6.0 nm straight, then turn east. Scanning
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt(dx*dx + dy*dy);
        raise scan;
        if ( dist >= 6.0 ) {
            startPosSet = false;
            setTargetHeading(90*pi/180);
            state = 7;
        }
    } else {
    // mark start of this distance leg
    startX = positionX;
    startY = positionY;
    startPosSet = true;
    }
} else if (state == 7 and not isTurning) {
    // travel 0.391734 nm straight, then turn north.
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt(dx*dx + dy*dy);
        if ( dist >= 0.391734 ) {
            startPosSet = false;
            setTargetHeading(0*pi/180);
            state = 8;
        }
    } else {
    // mark start of this distance leg
    startX = positionX;
    startY = positionY;
    startPosSet = true;
    }
} else if (state == 8 and not isTurning) {
    // travel 6.0 nm straight, then turn east. Scanning
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt(dx*dx + dy*dy);
        raise scan;
        if ( dist >= 6.0 ) {
            startPosSet = false;
            setTargetHeading(90*pi/180);
            state = 9;
        }
    } else {
    // mark start of this distance leg
    startX = positionX;
    startY = positionY;
    startPosSet = true;
    }
} else if (state == 9 and not isTurning) {
    // travel 1.891734 nm straight, then turn south.
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        if ( dist >= 1.891734 ) {
            startPosSet = false;
            setTargetHeading(180*pi/180);
            state = 10;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
}
}
} else if (state == 10 and not isTurning) {
    // travel 6.0 nm straight, then turn east. Scanning
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        raise scan;
        if ( dist >= 6.0 ) {
            startPosSet = false;
            setTargetHeading(90*pi/180);
            state = 11;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
}
} else if (state == 11 and not isTurning) {
    // travel 0.391734 nm straight, then turn north.
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        if ( dist >= 0.391734 ) {
            startPosSet = false;
            setTargetHeading(0*pi/180);
            state = 12;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
}
} else if (state == 12 and not isTurning) {
    // travel 6.0 nm straight, then turn east. Scanning
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        raise scan;
        if ( dist >= 6.0 ) {
            startPosSet = false;
            setTargetHeading(90*pi/180);
            state = 13;
        }
    }
} else {
    // mark start of this distance leg
startX = positionX;
startY = positionY;
startPosSet = true;
}
} else if (state == 13 and not isTurning) {
    // travel 0.391734 nm straight, then turn south.
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        if ( dist >= 0.391734 ) {
            startPosSet = false;
            setTargetHeading(180*pi/180);
            state = 14;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
} else if (state == 14 and not isTurning) {
    // travel 6.0 nm straight, then turn to -89.2608392 to RTB. Scanning
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        raise scan;
        if ( dist >= 6.0 ) {
            startPosSet = false;
            setTargetHeading(-89.2608392*pi/180);
            state = 15;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
} else if (state == 15 and not isTurning) {
    // travel 4.2052 then stop
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        if ( dist >= 4.2052 ) {
            startPosSet = false;
            state = 16;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
        setTargetDepth(0);
        raise signoff direct all; // done scanning... let others know
    }
} else if (state == 16) {
    raise log(1,positionX,positionY,positionZ) direct LOGGER;
    state = 17;
    setTargetSpeed(0);
    trace("Auv1 found %d MLCS... Signing off.",totalContacts);
    raise signoff direct all;
}
event scan {
    // simulate swath scanning of 0.25 nm by distance traveled in
    // 1 second @3.5 knots
    // assuming there are 27 mine-like contacts (MLC) in the field
    // distributed uniformly
    integer numContacts = PoissonRV(3.6467e-4);
    while (numContacts > 0, 100) {
        // handle the number of contacts up to 100
        real range = randReal2(0.0,0.25),x,y;
        if ( coinFlip() ) {
            // check what side the contact was made... equal prob either side
            x = positionX + range * cos(heading);
            y = positionY - range * sin(heading);
        } else {
            x = positionX - range * cos(heading);
            y = positionY + range * sin(heading);
        }
        raise contact(x,y) direct AUV3;
        numContacts = numContacts - 1;
        totalContacts = totalContacts + 1;
    }
}

end // device AUV1;

 device AUV2
    //the plant
    parameter
        const real hr2sec = 2.77777777777778e-4;
        // factor to convert from /hr to /sec

        const real maxSpeed = 3.8;  // maximum forward speed in knots
        const real maxAccel = 2.0;   // maximum acceleration in knots/sec

        const real alpha = log(0.003346)/maxSpeed;  // acceleration model
        const real alphaD = 15*pi/180;  // diving/surfacing model

        const real speedEps = 0.0075;  // +- variance in target speed acceptable

        const real headingChangeRate = pi/180;  // heading change rate in radians/second (pi radians/3 minute
        // = 180 degrees/3 minutes = pi radians/360 seconds
        const real Rturn = 0.5413;  // turning radius in nm @ speed of 3.5 nm/hr-determined experimentally

        real posX = 0, posY = 0, // x-y-z position of the AUV in NM
        positionZ = 0,  // forward speed in knots
        speed = 0,  // forward acceleration in knots/sec
        heading = 90*pi/180,  // heading in radians where
        hdg north := 0, hdg east := pi/2, etc.

        targetSpeed = speed,  // speed which the AUV should maintain
targetHeading = heading,          //heading which the AUV should maintain.
targetDepth = positionZ,          //depth which the AUV should maintain.
depthChangeRate = alphaD*speed;   //rate at which the AUV depth may change

boolean isAccelerating = false,
    isDecelerating = false,
    isTurning = false,
    isDiving = false,
    isSurfacing = false;

controllable
updateMotionModel { //update position
    positionX = positionX + hr2sec*speed*sin(heading);
    positionY = positionY + hr2sec*speed*cos(heading);

    //update speed
    speed = speed + accel;
    isAccelerating = speed < targetSpeed - speedEps;
    isDecelerating = speed > targetSpeed + speedEps;
    if ( isAccelerating )
        accel = maxAccel*exp(alpha*speed);
    else if ( isDecelerating )
        accel = -0.5*maxAccel*(speed-targetSpeed)/maxSpeed;
    else
        accel = 0;

    //update heading
    real dH = fabs(heading - targetHeading);
    isTurning = dH > headingChangeRate + 0.001;
    if ( isTurning ) {
        if ( heading < targetHeading and dH <= pi or heading >=
            targetHeading and dH > pi )
            //turn right
        head = heading + headingChangeRate;
    } else
        //turn left
        heading = heading - headingChangeRate;

    if (heading > pi) heading = heading - 2*pi;
    if (heading < -pi) heading = heading + 2*pi;

    //update depth
    depthChangeRate = alphaD*speed;
    isDiving = positionZ < targetDepth - (depthChangeRate + 0.001);
    isSurfacing = positionZ > targetDepth + (depthChangeRate +
        0.001);
    if (isDiving)
        positionZ = positionZ - depthChangeRate;
    else if (isSurfacing)
        positionZ = positionZ + depthChangeRate;
    else
        positionZ = targetDepth;
}
setTargetSpeed(real s) precondition (s >= 0 and s <= maxSpeed) {
    targetSpeed = s;
}
setTargetHeading(real h) {
targetHeading = h;
setTargetDepth(real d) precondition (d <= 0) {
    targetDepth = d;
}

// the controller
memory
integer state = 0, count = 10, totalContacts = 0;  // keep track of sub-state
boolean startPosSet = false;
real startX, startY;

control
event tic(real t) authorize SIMTimekeeper {
    updateMotionModel;
    if (isTurning) {
        if (count < 10) // when turning, log position every 10 seconds
            count = count + 1;
        else {
            raise log(2,positionX,positionY,positionZ) direct LOGGER;
            count = 0;
        }
    } else {
        if (count < 600) // when not turning, log position every 10 minutes
            count = count + 1;
        else {
            raise log(2,positionX,positionY,positionZ) direct LOGGER;
            count = 0;
        }
    }
    real dx, dy, dist;
    if (state == 0) { // initialization - accelerate to 3.5 due East
        setTargetDepth(-0.002);
        setTargetSpeed(3.5);
        setTargetHeading(90*pi/180);
        state = 1;
    } else if (state == 1 and not isAccelerating) { // turn South
        setTargetHeading(180*pi/180);
        state = 2;
    } else if (state == 2 and not isTurning) { // turn East
        setTargetHeading(90*pi/180);
        state = 3;
    } else if (state == 3 and not isTurning) {
        // travel 1.541831 nm straight, then turn North
        if (startPosSet) {
            dx = positionX - startX; dy = positionY - startY;
            if (sqrt(dx*dx + dy*dy) >= 1.541831) {
                startPosSet = false;
                setTargetHeading(0*pi/180);
                state = 4;
            }
        } else {
            // mark start of this distance leg
            startX = positionX;
            startY = positionY;
            startPosSet = true;
        }
    }
} else if (state == 4 and not isTurning) {
    // travel 6.054133 nm straight, then turn east.
    // Start scanning after 0.054133 nm
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        if ( dist >= 0.054133 )
            raise scan;
        if ( dist >= 6.054133 ) {
            startPosSet = false;
            setTargetHeading(90*pi/180);
            state = 5;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
} else if (state == 5 and not isTurning) {
    // travel 0.391734 nm straight, then turn south.
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        if ( dist >= 0.391734 ) {
            startPosSet = false;
            setTargetHeading(180*pi/180);
            state = 6;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
} else if (state == 6 and not isTurning) {
    // travel 6.0 nm straight, then turn east. Scanning
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        if ( dist >= 6.0 ) {
            startPosSet = false;
            setTargetHeading(90*pi/180);
            state = 7;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
} else if (state == 7 and not isTurning) {
    // travel 0.391734 nm straight, then turn north.
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        if ( dist >= 0.391734 ) {
            startPosSet = false;
            setTargetHeading(0*pi/180);
            state = 8;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
} else {
    // mark start of this distance leg
    startX = positionX;
    startY = positionY;
    startPosSet = true;
}
} else if (state == 8 and not isTurning) {
    // travel 6.0 nm straight, then turn east. Scanning
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        raise scan;
        if ( dist >= 6.0 ) {
            startPosSet = false;
            setTargetHeading(90*pi/180);
            state = 9;
        } else {
            // mark start of this distance leg
            startX = positionX;
            startY = positionY;
            startPosSet = true;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
} else if (state == 9 and not isTurning) {
    // travel 1.891734 nm straight, then turn south.
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        if ( dist >= 1.891734 ) {
            startPosSet = false;
            setTargetHeading(180*pi/180);
            state = 10;
        } else {
            // mark start of this distance leg
            startX = positionX;
            startY = positionY;
            startPosSet = true;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
} else if (state == 10 and not isTurning) {
    // travel 6.0 nm straight, then turn east. Scanning
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        raise scan;
        if ( dist >= 6.0 ) {
            startPosSet = false;
            setTargetHeading(90*pi/180);
            state = 11;
        } else {
            // mark start of this distance leg
            startX = positionX;
            startY = positionY;
            startPosSet = true;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
} else if (state == 11 and not isTurning) {
    // travel 0.391734 nm straight, then turn north.
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        if ( dist >= 0.391734 ) {
            startPosSet = false;
            setTargetHeading(0*pi/180);
            state = 12;
        } else {
            // mark start of this distance leg
            startX = positionX;
            startY = positionY;
            startPosSet = true;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
}
startPosSet = false;
setTargetHeading(0*pi/180);
state = 12;
}
} else {
    // mark start of this distance leg
    startX = positionX;
    startY = positionY;
    startPosSet = true;
}
} else if (state == 12 and not isTurning) {
    // travel 6.0 nm straight, then turn east. Scanning
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        raise scan;
        if ( dist >= 6.0 ) {
            startPosSet = false;
            setTargetHeading(90*pi/180);
            state = 13;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
} else if (state == 13 and not isTurning) {
    // travel 0.391734 nm straight, then turn south.
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        if ( dist >= 0.391734 ) {
            startPosSet = false;
            setTargetHeading(180*pi/180);
            state = 14;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
} else if (state == 14 and not isTurning) {
    // travel 6.0 nm straight, then turn to -89.455482536 to RTB. Scanning
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        raise scan;
        if ( dist >= 6.0 ) {
            startPosSet = false;
            setTargetHeading(-89.455482536*pi/180);
            state = 15;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
    }
} else if (state == 15 and not isTurning) {
    // done scanning... let
    others know
    raise signoff direct all;
} else if (state == 15 and not isTurning) {
    // travel 5.7089 then stop
    if (startPosSet) {
        dx = positionX - startX; dy = positionY - startY;
        dist = sqrt( dx*dx + dy*dy );
        if ( dist >= 5.7089 ) {
            startPosSet = false;
            state = 16;
        }
    } else {
        // mark start of this distance leg
        startX = positionX;
        startY = positionY;
        startPosSet = true;
        setTargetDepth(0);
    }
} else if (state == 16) {
    raise log(2, positionX, positionY, positionZ) direct LOGGER;
    state = 17;
    setTargetSpeed(0);
    trace("Auv2 found %d MLCs... Signing off.", totalContacts);
}

} event scan {
    // simulate swath scanning of 0.25 nm by distance traveled in
    // 1 second @3.5 knots
    // assuming there are 27 mine-like contacts (MLC) in the field
    // distributed uniformly
    integer numContacts = PoissonRV(3.6467e-4);
    while (numContacts > 0, 100) {
        // a MLC detected
        real range = randReal2(0.0, 0.25), x, y;
        if ( coinFlip() ) {
            // check what side the contact was made... equal prob either side
            x = positionX + range * cos(heading);
            y = positionY - range * sin(heading);
        } else {
            x = positionX - range * cos(heading);
            y = positionY + range * sin(heading);
        }
        raise contact(x, y) direct AUV3;
        numContacts = numContacts - 1;
        totalContacts = totalContacts + 1;
    }
}
}

end // device AUV2;

/*===================================================================
device AUV3 // the Identification AUV
/*===================================================================

//the plant
parameter
const real hr2sec = 2.777777777777778e-4;
const real maxSpeed = 5.0;  // maximum forward speed in knots
const real maxAccel = 2.65; // maximum acceleration in knots/sec
const real alpha = log(0.003346)/maxSpeed;
const real alphaD = 15*pi/180;  //diving/surfacing model parameter
const real speedEps = 0.01;   //+- variance in target speed acceptable
const real headingChangeRate = pi/180;  //heading change rate in radians/second (pi radians/3
minute
  // = 180 degrees/3 minutes = pi radians/360 seconds

real positionX = 0, positionY = 0, //x-y-z position of the AUV in NM
  positionZ = 0, //forward speed in knots
  speed = 0,
  accel = 0, //forward acceleration
  heading = 90*pi/180,
  //heading in radians where hdg north := 0, hdg east := pi/2, etc.
  targetSpeed = speed, //speed which the AUV should maintain
  targetHeading = heading, //heading which the AUV should maintain.
  targetDepth = positionZ, //depth which the AUV should maintain.
  depthChangeRate = alphaD*speed; //rate at which the AUV depth may change

boolean isAccelerating = false,
  isDecelerating = false,
  isTurning = false,
  isDiving = false,
  isSurfacing = false;

ccontrollable
updateMotionModel {
  //update position
  positionX = positionX + hr2sec*speed*sin(heading);
  positionY = positionY + hr2sec*speed*cos(heading);

  //update speed
  speed = speed + accel;
  isAccelerating = speed < targetSpeed - speedEps;
  isDecelerating = speed > targetSpeed + speedEps;
  if ( isAccelerating )
    accel = maxAccel*exp(alpha*speed);
  else if ( isDecelerating )
    accel = -0.5*maxAccel*(speed-targetSpeed)/maxSpeed;
  else
    accel = 0;

  //update heading
  real dH = fabs(heading - targetHeading);
  isTurning = dH > headingChangeRate + 0.001;
  if ( isTurning ){
    if ( heading < targetHeading and dH <= pi or heading >=
      targetHeading and dH > pi )
      //turn right
      heading = heading + headingChangeRate;
    else
      //turn left
      heading = heading - headingChangeRate;
} else
    heading = targetHeading;
if (heading > pi) heading = heading - 2*pi; if (heading < -pi)
    heading = heading + 2*pi;

//update depth
depthChangeRate = alphaD*speed;
isDiving = positionZ < targetDepth - (depthChangeRate + 0.001);
isSurfacing = positionZ > targetDepth + (depthChangeRate +
0.001);
if (isDiving)
    positionZ = positionZ - depthChangeRate;
else if (isSurfacing)
    positionZ = positionZ + depthChangeRate;
else
    positionZ = targetDepth;

setTargetSpeed(real s) precondition (s >= 0 and s <= maxSpeed) {
    targetSpeed = s;
}
setTargetHeading(real h) precondition (h > -pi and h <= pi) {
    targetHeading = h;
}
setTargetDepth(real d) precondition (d <= 0) {
    targetDepth = d;
}

//the controller
memory
integer state = 0,       //keep track of sub-state
    currentTarget = 0, //keep track of which target we're
working on
    numTargets = 0, count = 10, numActiveAUVs = 2;
real targetX[50], targetY[50],  // sufficient storage for 50
    startX, startY;
    targets
const real travelSpeed = 4.5, opSpeed = 2.0, IDrange = 0.075;

control
event tic(real t) authorize SIMTimekeeper {
    real dx,dy;
    if (isTurning) {
        if (count < 10) // when turning, log position every 10
            seconds
            count = count + 1;
        else {
            count = 0;
            raise log(3,positionX,positionY,positionZ) direct
                LOGGER;
        }
    } else {
        if (count < 600) // when not turning, log position every 10
            minutes
            count = count + 1;
        else {
            count = 0;
            raise log(3,positionX,positionY,positionZ) direct
                LOGGER;
        }
    }
}
updateMotionModel;
if (state == 0) { // initialization State
    raise log(3,positionX,positionY,positionZ) direct LOGGER;
    setTargetDepth(0.002);
    state = 1;
} else if (state == 1 and numTargets <= currentTarget and numActiveAUVs == 0) {
    state = 4;
} else if (state == 1 and numTargets > currentTarget) {
    // waiting for contact state
    setTargetSpeed(travelSpeed);
    state = 2;
} else if (state == 2) { // travel to contact
    dx = targetX[currentTarget] - positionX;
    dy = targetY[currentTarget] - positionY;
    setTargetHeading( atan2(dx,dy) );
    if ( sqrt(dx*dx + dy*dy) <= IDrange ) {
        setTargetSpeed(opSpeed);
        state = 3;
    }
} else if (state == 3) {
    raise contactReport(2, currentTarget+1, targetX[currentTarget], targetY[currentTarget]) direct LOGGER;
    currentTarget = currentTarget + 1;
    state = 1;
    if (numTargets <= currentTarget)
        setTargetSpeed(0.0); // stop if there is no current target
} else if (state == 4) { // RTB
    setTargetSpeed(travelSpeed);
    dx = 0 - positionX;
    dy = 0 - positionY;
    setTargetHeading( atan2(dx,dy) );
    if ( sqrt(dx*dx + dy*dy) <= 0.05 ) {
        setTargetSpeed(0.0);
        state = 5;
    }
} else if (state == 5 and not isDecelerating) { // done
    raise log(3,positionX,positionY,positionZ) direct LOGGER;
    trace("Auv3 visited %d MLCs... Signing off.",numTargets);
    state = 6;
}

event contact(real x, real y) authorize AUV1 union AUV2 {
    if (numTargets < 50) {
        targetX[numTargets] = x;
        targetY[numTargets] = y;
        numTargets = numTargets + 1;
        raise contactReport(1, numTargets, x,y) direct LOGGER;
    } else {
        trace("Contact buffer overflow.");
    }
}

event signoff authorize AUV1 union AUV2 {
    numActiveAUVs = numActiveAUVs - 1;
}
end // device AUV3;
Appendix B

C3L Compiler Code Documentation

The complete Doxygen generated C3L compiler code documentation is available via CD or email zip file. Please contact the Author at sustersi@cse.psu.edu for details.
JOHN PHILLIP SUSTERSIC

RESEARCH EXPERIENCE

Research Engineer, Information Science and Technology Division, The Applied Research Laboratory, The Pennsylvania State University
Under Dr. Shashi Phoha, Summer 2008 – Present
Develop planning tools and software support for Mine Countermeasures Mission planning.

Research Assistant, Department of Computer Science and Engineering, The Pennsylvania State University
Under Dr. Mahmut Kandemir, Fall 2007 - Present


Research Assistant, Department of Computer Science and Engineering, The Pennsylvania State University
Investigated internet caching protocols. Developed event-based network simulator.

Lead Research Assistant, The Advanced Research Lab, Department of Electrical and Computer Engineering, Cleveland State University

PUBLICATIONS


