A PRESENTATION OF THE SEMANTICS AND FORMAL PROPERTIES OF C3L, AN EVENT-DRIVEN DISTRIBUTED CONTROL LANGUAGE

Richard Skarbez

December 15, 2004

Abstract

The viability of a multi-device control language is largely reliant on its ability to satisfy certain general requirements, such as being able to directly interface with the underlying hardware, possessing a sufficiently high level of abstraction to allow complex control structures, being platform independent, supporting concurrent operation, and possessing formal, provable characteristics. Under the ONR-funded research program on Ocean SAmpling MObile Networks (SAMON), C3L was designed as a programming language which would satisfy these and other requirements. The current version of C3L was defined under the ESP MURI prgram, as a device-oriented decision and control language that possesses useful formal characteristics. This thesis presents the complete formal operational semantics for the C3L language. The semantics are demonstrated in proofs of characteristics of the C3L language, including event fairness and prefix-closed controllability. Also presented are a discussion of the static analyses used to ensure that the semantics are properly followed in the language, and an application demo using the C3L language. The application demo consists of an analysis of a locality-based cluster head election algorithm implemented in C3L, and a statistical analysis of the results generated by the algorithm.

Key Words: Programming language semantics, operational semantics, multi-device control, control theory

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank, first, Mendel Schmiedekamp, for bringing me onto this project, and for his extensive guidance of and assistance with my research. I also would like to thank Dr. Shashi Phoha for her comments and contributions to this thesis. My thanks also go out to my honors advisors while at Penn State, Drs. Ali Hurson and Chita Das. I apologize for the headaches that I'm sure I've caused all of you, and I am indebted to each of you for your help.

Also, thanks here have to go out to my parents. I wouldn't have made it here without you. Thanks for everything.

The work for this thesis was sponsored by the Defense Advance Research Projects Agency (DARPA), and administered by the Army Research Office under Emergent Surveillance Plexus MURI Award No. DAAD19-01-1-0504. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

Abstract

The viability of a multi-device control language is largely reliant on its ability to satisfy certain general requirements, such as being able to directly interface with the underlying hardware, possessing a sufficiently high level of abstraction to allow complex control structures, being platform independent, supporting concurrent operation, and possessing formal, provable characteristics. Under the ONR-funded research program on Ocean SAmpling MObile Networks (SAMON), C3L was designed as a programming language which would satisfy these and other requirements. The current version of C3L was defined under the ESP MURI prgram, as a device-oriented decision and control language that possesses useful formal characteristics. This thesis presents the complete formal operational semantics for the C3L language. The semantics are demonstrated in proofs of characteristics of the C3L language, including event fairness and prefix-closed controllability. Also presented are a discussion of the static analyses used to ensure that the semantics are properly followed in the language, and an application demo using the C3L language. The application demo consists of an analysis of a locality-based cluster head election algorithm implemented in C3L, and a statistical analysis of the results generated by the algorithm.

Key Words: Programming language semantics, operational semantics, multi-device control, control theory

Chapter 1

Introduction

The problem of dynamic command and control in multiple device systems is an extremely complex one, incorporating issues of inter-device communication, group formation, and modeling of environments and devices, among other issues. C3L (Command Control Communications Language) represents an attempt to develop a programming language particularly suited to this problem domain.

The original goal of C3L development was to create a language for command and control of multiple unmanned undersea vehicles (UUVs). This line of research was initiated at Penn State ARL in 1997 [1]. Under the ESP MURI, the language was updated and extended for use in sensor network applications, resulting in what is now C3L. This extension of the language was largely completed in late 2003, leading to [2]. The specific contributions of this thesis to this line of research are the development and presentation of the formal operational semantics for the language, the static analysis and proofs derived from these semantics that are presented in this thesis, and the simulation results and validation presented in Section 6.

Despite its origins as a robot control language, the variety of applications for which C3L is suited, make it more appropriate to consider C3L as a device control language, although it certainly still has very clear applications in the robot control domain. Some applications of such a device control language are maintenance of dynamic *ad hoc* networks, agent-based network intrusion detection systems, and traffic management systems. Each of these applications, as well as many other applications for which C3L is well-suited, can be considered mission critical. In order to have some assurance that devices controlled by C3L programs will function correctly in actual use, it is useful to build some formal properties of execution into the language itself.

In order to discuss the formal properties of a program written in a given programming language, one must use the semantics of the language. The semantics of a programming language, loosely, are a

set of formal mathematical rules that describe the meaning of a program written in the language. The semantics of a language is generally developed with consideration to the syntax of the language, which is the set of rules that determines whether a given sequence represents a well-formed program in the context of the language. If a language has a correct and complete semantics, then the semantic rules can be used to prove characteristics of the program's behavior.

Chapter 2, presents some of the motivating characteristics for the creation of the C3L language, as well as a comparison with other languages used for multiple device control. It also presents a brief comparison of the different classes of programming language semantics, and our reasons for choosing to specify the operational semantics for C3L, instead of another type of semantics. Chapter 3, goes into further detail regarding the nature of the C3L language, its goals, and its characteristics. Chapter 4 presents the semantics for the language, and discuss the static analyses used to check program correctness. Chapter 5 presents proofs showing that the properties of event fairness and prefix-closed controllability hold for all programs written in C3L. In Chapter 6, an application developed in C3L is presented, along with an analysis of simulation results, and Chapter 7 presents future research directions for the language.

Chapter 2

Comparisons and Motivations

The motivations discussed in this chapter are twofold. First, general motivating characteristics for a device control language are specified, and C3L is compared with other existing control languages. This serves to motivate the creation of the C3L Language. Second, the motivating characteristics for programming language semantics, and a comparison of several types of semantics, are presented. This section serves to motivate the development of the semantics for the C3L language, and the choice of operational semantics for this project.

2.1 Robot Control Languages

C3L is designed to be suitable for use in a wide variety of multiple device command and control applications. That said, C3L has its roots in the more specific domain of Robot Control Languages (RCLs), and as such, it is useful to consider how C3L stands in relation to other RCLs, and to gauge the ways in which C3L is more or less capable than other languages in the problem space of robot control. To this end, the remainder of this section is dedicated to a survey of other types of multiple device control models, including agent control languages, multi-layered hybrid control, and concurrent constraint programming.

C3L expresses a generic model of decentralized collaboration and behavior based dynamic control of networked devices. The motivating characteristics of such a language are described below and it is compared with other decentralized computational paradigms. The following list of motivating characteristics was taken from [3]. This list of desired characteristics was used in the development of [4]:

- Minimum abstraction level low enough to support real-time machine programming and high enough for platform independence, task planning, and human supervision: If the mimimum abstraction level of a language is too high, the programmer does not have sufficient access to the low-level

structures to control what the machine would actually execute at runtime, and if it is too low, the language does not carry enough semantic information to allow intuitive description and understanding of programs written in the language.

- Construction of group behaviors via semantic aggregation for generating and revising plans: Most multiple device control applications use groups of devices working together to achieve some specified goal. One example of such an application is distributed tracking in a sensor network. A language that would be appropriate for such an application must have the appropriate functionality to allow group formation and coalition.

- Communication constructs suitable for distributed coordination: In order for distributed devices to cooperate on a task, there must be a means for devices to communicate with one another and coordinate their actions. Ideally, the communication constructs would be inherent in the language, for both ease of use and reliability purposes.

- Ability to control a wide range of robots (implies platform independence as well as code mobility and dynamic migration of responsibility): A programming language that is limited to a specific hardware platform has little usefulness in many real world applications, such as a network of sensor nodes, wherein different nodes have different sensor packages. Dynamic migration of responsibility is useful to avoid having a single point-of-failure in the network.

- Accommodation of physical constraints: This requirement regards the fault tolerance of a network of devices. A device should be able to function intelligently even if its abilities are for some reason constrained, and similarly, a network devices should be able to adapt if a device is constrained or disabled.

- Modular system design to allow dynamic reconfiguration of device clusters: This is not a necessary property of a device control language, although it is very desirable. As discussed above, dynamic reconfiguration of device clusters can help to avoid having single points-of-failure in a network of devices. Also, it allows for the repurposing of devices to meet changing mission plans (for example, converting some devices from tracking to pursuit of a target), or changing network conditions (for example, replacing a disabled device in a group). Also, it can be used purely for performance reasons, such as the example presented in Chapter 6, wherein the group elects a new supervisor node based on its proximity to the target.

- Support for concurrency and synchronization control: A network of devices should intuitively be able to perform tasks simultaneously. Also, in tasks which require multiple devices to act in concert,

the ability to force multiple devices to coordinate on some task at a specific time or times is often important.

- Support for reactive and deliberative computation: Devices controlled by a device control language should be able both to respond intelligently to unexpected events from the environment, and also to perform meaningful tasks in the absence of any such events.

- Ability to consistently integrate and reconfigure heterogeneous elements at multiple levels of command and control: A device control language should not only be platform independent, it should also allow devices with different hardware platforms to interact within the same network. The language should provide a meaningful way for these devices to communicate and cooperate, despite differences at the hardware level.

- A dynamic systems model which supports the development of formal performance metrics: The language should have the capability of dynamically reporting on the status of networked devices, allowing for analysis of device and network behavior.

- Adaptation and learning to provide robustness and intelligence: A device network should be able to adapt to changing environmental conditions.

A domain-specific augmentation of a general purpose programming language results from the addition of syntactic support for various methods, classes, data structures, control elements, etc. to an existing programming language such as LISP, C++, or Java. A Robot Control Language developed in this way has all the advantages and disadvantages associated with the parent language. Generally, this class of languages has greater flexibility and computational power than the others, owing to its general purpose nature. Also, such languages are platform-independent, provided a suitable compiler is available. However, these languages have significantly more overhead than other Robot Control Languages. Also, in complex problem domains, use of one of these languages may result in highly complex and nonintuitive code, which makes them difficult to write in and understand. Furthermore, it is difficult, if not impossible, to ensure the formal properties of such a language, owing to the complexity of the language and the interaction of the augmentations and the base language. Some examples of domain-specific augmentations of general purpose languages are PRS [6], TDL [7], and Rodney Brooks's Behavior Language [8].

Agent control consists primarily of two subproblems: agent coordination, and agent communication. Agent coordination represents the aspect of a multi-agent system that governs agent interaction and behaviors. There are two primary schools of thought in regard to developing a dependable agent coordination system. The first is to use strict protocols, providing a sort of global control over the agents.

The second is to incorporate control within each agent, and allow agents to interact via message passing. Languages such as FIPA [10] and KQML [11] use the former approach; C3L uses the latter.

In general, agent control languages are very suitable for abstract design and control of agents, and devices specified using one of these languages are generally flexible and adaptive. However, these languages are generally computationally inefficient, often do not offer the same degree of flexibility and efficiency of mathematical computation as other languages, provide little ability to directly control timing issues, and are sufficiently removed from low-level programming constructs so as to make actual physical implementation problematic.

Another class of multi-device control languages are the multi-layered hybrid control languages. Deshpande, Göllü, and Semenzato in their SHIFT language [17] identified the problem of description and simulation of dynamic networks of hybrid automata. They defined such networks as systems of components that can be created, interconnected, and destroyed over time, where the components themselves exhibit hybrid behavior, in the sense that they act in continuous-time punctuated by instantaneous discrete-time transitions. They also identified state variables, events, and dynamically reconfigurable interaction networks as components in a model of such a system. The SHIFT language is presented as a way of describing this model. C3L is similar to SHIFT in that it treats the system as a network of discrete devices (components in SHIFT), maps the state space using state variables (in C3L, parameter variables describe the plant state, and memory variables are one component of the controller state description), uses discrete-time events to transition between states, and supports dynamically configurable networks via group constructs which can be modified inside event handlers. C3L provides some additional functionality particularly relevant to the problem spaces of mobile sensor networks and mission planning. In SHIFT, components can only communicate by modifying the values of specified link variables. Communication in C3L is much more robust, as a device can communicate with any arbitrary subset of other known devices in the network by raising events and associated messages (sets of zero or more values) to one or more groups or device names. Also, SHIFT does not expressly offer recursion or iteration as commands in the language. C3L does offer recursion, achieved by raising controllable events, and because recursion is achieved by placing events on the tail of input queue, it is possible to show event fairness and prefix-closed controllability, even with the incorporation of recursion.

While SHIFT uses a hybrid automata model, other models that can be used for hybrid control of dynamic systems are Petri nets, Communicating Sequential Processes, temporal logic, or finite state machine models.

A fourth solution type commonly discussed in relation to the problem of multi-agent control is concurrent constraint programming. In their 1989 paper on Concurrent Constraint Programming [18], Saraswat

and Rinard discuss the cc family of languages, first developed in Saraswat's dissertation [19]. This family of languages generalizes constraint logic programming to a concurrent computing environment, and are characterized by the inclusions of concurrency and the atomic tell and blocking ask operations. The atomic tell operation adds a constraint to the store, or memory, which is initially empty. The blocking ask operation queries the store with a constraint, to see if that constraint is satisfied given the current state of the store. Concurrency here means that multiple processes can generate asks and tells concurrently. Computation terminates when no more information can be added to the store.

Concurrent constraint programming has since been extended to include stochastic behavior [20] and temporal computation [21], among other concepts. Timed concurrent constraint programming provides the minimum functionality required for implementing control of autonomous agents, and its reactive computation model corresponds intuitively to the reactive operation of typical robotic devices, where input is received, causing computation to occur, and then output is produced, often in the form of device action.

Inter-device communication in ccp is performed using the common store, in opposition to the event-driven message passing in C3L. The common store model may present difficulties in implementation of a network of autonomous devices, which may have limited computational resources: either each device must maintain a current copy of the store, which results in increased network traffic and requires synchronization across devices; or there must be a single controller for the system, which eliminates the problem of synchronizing the store across devices, but also removes the benefits of distributed dynamic control. Romero chooses the former in his paper [22] on mobile concurrent constraint programming, but does not offer a solution to the problem of inconsistency across multiple stores.

2.2 Progenitors of C3L

The current version of C3L does not represent the first attempt to generate a unified multiple device control language in this vein. Two previous attempts were made in this regard at ARL, namely, Generic Behavior Message Passing Language (GBML) [1] and a prototype common control language [30]. C3L inherits some syntactic structure from thesse earlier languages, and also builds upon the central theory behind the development of these languages, hierarchical discrete event control. The roots of hierarchical discrete event control lie in application of general discrete event control [25] to multi-device problems. This is the origin of ideas such as plant model definitions, communication via discrete events, and a combination of dynamic events. The notion of control used in C3L is derived from intelligent control [26] and these discrete event control models [27, 28].

The original work with programming languages incorporating these models was in the Ocean SAmpling MObile Network (SAMON) project [1, 29, 30]. SAMON was concerned with simulating and coordinating the operation of multiple unmanned undersea vehicles (UUVs). This project was an early attempt to apply the basic principles of control theory to a multi-robot application. These principles were applied to the languages described above and they were used to control UUVs. Of note is that the notion of a common control language was first put forward in [30], as a means of coordinating the behaviors of multiple heterogenous devices. While expanding the scope of these earlier languages to incorporate broader applications including agent control and sensor networks [2, 31], C3L has proven to be useful in UUV applications [32] and others.

2.3 Programming Language Semantics

In order to prove properties of programs or device specifications written in C3L, one must be able to generate a precise, mathematical model of the program. That is precisely what the semantics of a programming language allows us to do.

There are three primary classes of programming language semantics: operational, denotational, and axiomatic. Each of these represents a different approach to defining the "meaning" of a program, or a programming language. The following discussions are derived from those presented in [33] and [34].

In the operational approach, the meaning of a program is defined by the series of computation steps performed by the program running on some abstract machine. The semantic rules in an operational semantics define how the state of the machine is changed by the executions of commands in the language. Operational semantics are very closely linked to the syntax of the language, more so than either denotational or axiomatic semantics.

In the denotational approach, the meaning of a program is a mathematical function that maps the program input to program output. The individual steps in the execution are unimportant. Semantics generated from the denotational approach are powerful, but based on complex mathematical constructs such as partial orders and fixed-points of a set.

In the axiomatic approach, the meaning of a program is a logical statement regarding some property of the input and output of a program. Axiomatic semantics are intimately related to the concept of program proving, whereby a program's correctness can be directly verified by consultation of its axiomatic semantics.

In this project, the decision was made to generate the operational semantics for the C3L language. This is because operational semantics aid in the implementation of a language, since the operational

description itself can be thought of as an interpreter for the language, and also since the necessary conditions for correct execution of a command are explicit in the operational rules, which is useful in the generation of a pathology checker which tests for the well-formedness of a program. Lastly, the operational semantics are more intuitive than either the denotational or axiomatic semantics, which is an additional benefit.

In Sections 4 and 5, the operational semantics of C3L will be presented and specific characteristics of the language will be established through proofs and static analysis, respectively.

Chapter 3

Discussion of C3L

This chapter goes in to greater detail regarding the characteristics and motivations of C3L, and also presents the syntax of the language.

3.1 Description of C3L

C3L represents an attempt to design a device-oriented programming language which possesses the desired characteristics set forth in Section 2.1. C3L is an event-driven distributed control language designed to facilitate the implementation of large systems of heterogeneous autonomous devices [2]. C3L incorporates the plant/controller model of a controlled system into the language, such that a programmer defines a device in C3L by specifying first its plant and then its control. Communications are integrated into the device controller so that the user is not required to interact with communications on the protocol level. For a more detailed discussion of the features of the language, please consult [2].

3.2 Syntax of C3L

Following is the syntax of C3L presented in modified Backus-Naur form [36], using bold to indicate terminals and italics to indicate unspecified variables. This syntax was previously stated in [2].

device ::= **device** label [sp][par][unc][con][grp][mem][control]**end**

sp ::= **plant** label

par ::= **parameter** {var}

unc ::= **uncontrollable** {pevent}

con ::= **controllable** {pevent}

grp ::= **group** {label **:=** slist }

mem ::= **memory** {var}

control ::= **control** {cevent}

pevent ::= label ([cond])({assign})

var ::= label [**const**] type [(value{,value})]

cevent ::= **event** label [(label{,label})] [**authorize** slist] {command} **end**

slist ::= { string | label } [**except** { string | label }] **end**

command ::= choice | label [(expr{,expr})] | **raise** label [(expr{,expr})] [**direct** slist] |
gadd label slist | **grem** label slist | assign

choice ::= **choice** (cond) command {**cor** (cond) command} **end**

cond ::= (cond) boolop (cond) | **not**(cond) | expr comp expr | label |
string **in** label | label **in** label | boolean

boolop ::= **and** | **or**

comp ::= =| <| >| ≤| ≥| **!** =

assign ::= label **:=** expr**;** | label[[integer{,integer}]] **:=** expr**;**

expr ::= cond | (expr mathop expr) | *math*([expr{, expr}]) | - expr |
value | label[[integer{,integer}]]

mathop ::= +| −| ∗| /

type := **integer** | **real** | **boolean** | **string** | type[integer]

value ::= *integer* | *real* | **true** | **false** | string | **NULL**

string ::= **“**label**”**

label ::= {*char*}

The variables *integer*, *real*, and *char* represent integers, real numbers, and characters which are used as values in the language. The variable *math* generates a set of function names which are defined for a particular device.

3.3 Specific Motivations for C3L

The following are the specific stated motivations for the development of C3L, as stated in [2]:

1. Dynamic Organization - Devices must be able to adapt their communication and control structures in real-time in response to changes in their environment. Practically, this means that devices must be able to dynamically reconfigure their control hierarchy and communication pathways.

2. Accomodation of Uncontrollable Events - Devices must be able to process and respond to uncontrollable events in their environment.

3. Portability - A C3L specification of a device should be cross-platform portable, so long as the basic functionality of the underlying hardware platform is comparable.

4. Formality - Programs written in the C3L language should have verifiable properties.

5. Expressiveness - The C3L language should be at least as expressive as a recursive language, and should also provide capabilities for more complex programming, such as device communication and distributed computing.

6. Extensibility - The C3L language should allow for extension and modification of the language, with minimal changes to the existing syntax or semantics.

A further discussion of these motivations for the C3L language can be seen in [2].

3.4 Analysis of C3L Goals and Requirements

The goals set forth in Section 2.1 and the requirements of Section 3.3 provide a basis for judging the merits of the language as specified in [2]. Following is an analysis of C3L in light of the characteristics of a multiple device control language. Each of the desired characteristics from Section 2.1 will be considered individually.

- The minimum abstraction level should be low enough to support real-time machine programming and high enough for platform independence, task planning, and human supervision:
  C3L can be considered in some sense to be a fourth-generation programming language [35], designed explicitly for use in the command and control of autonomous devices. The structure of programs in the C3L language is explicitly based on the specification of state transitions occurring from controllable and uncontrollable events, which adds to the human-readability of the programs. Also, complex behaviors such as communication are hidden from the user, which benefits human-readability, in addition to contributing to the platform independence of the language, since low-level constructs can be modified without any changes to the language itself. C3L programs are currently interpreted; that is, a previously compiled C++ program (the interpreter) runs in real time, with the C3L source as its input. This is not a necessary condition; C3L programs could be compiled at some point, but at this point no compiler has been developed. The interpreted nature of C3L

gives some of the benefits of the lower-level language in terms of efficiency, while being platform independent to the degree that it requires only an existing C++ compiler to run on any hardware.

- The language should allow construction of group behaviors via semantic aggregation for generating and revising plans:

  C3L allows for group constructs, which are created at device initialization and contain a member list containing zero or more device names or previously-defined group names that belong to the new group, and an `except` list containing the same, which identifies devices or groups which are explicitly defined to not be in the group. The groups can be modified during runtime through commands in event handlers. Controllable events can be raised to a group or groups by supplying a direct list, and devices can identify which devices or groups from whom to accept events by adding them to an associated authorize list. The group constructs in C3L allow for creation and modification of multiple-device behaviors.

- The language should possess communication constructs suitable for distributed coordination:

  Communication constructs are built into C3L, and are abstracted from the user. Since the behavior of the language does not depend on the actual implementation scheme for communication, it is possible to use a communication scheme most suitable for a given application.

- The language should have the ability to control a wide range of devices (This implies platform independence as well as code mobility and dynamic migration of responsibility):

  As discussed previously, C3L is platform-independent to the degree that it only requires an extant C++ compiler. Also, the group constructs in C3L allow for dynamic migration of responsibility. C3L does not, however, implement code mobility. This is a potential future research direction for the language.

- The language should accommodate physical device constraints:

  The plant/controller model automatically addresses this issue, as the physical device constraints must be modeled in the C3L plant.

- The language should support modular system design, in order to allow dynamic reconfiguration of device clusters:

  The group and communication constructs, as discussed above, allow for dynamic reconfigurability of device clusters.

- The language should support concurrency and synchronization control:

  Computation in a system of C3L devices is inherently concurrent and asynchronous. Within a

single device, C3L is at this time limited to sequential computation. It is possible that parallel computation within a single device may be considered as a future research direction.

- The language should support both reactive and deliberative computation:
  C3L's event-driven computation contains support for both controllable (deliberative) and uncontrollable (reactive) events.

- The language should have the ability to consistently integrate and reconfigure heterogeneous elements at multiple levels of command and control:
  This is another feature of the group constructs and event-driven computation in C3L. If a device does not have the capability or the authorization to perform a given event, it will be ignored. This does not prevent another device with different abilities from performing the event. Different capabilities are indicated in a system of C3L devices by differences in the plant models of the devices. A network of devices controlled by C3L programs can incorporate heterogeneous devices, that is, not all devices in the network need to have the same capabilities, or even the same plant/controller models.

- The language should utilize a dynamic systems model which supports the development of formal performance metrics:
  The C3L model of event-driven communication and computation is naturally well-suited to simulation, or to observation of a network of devices. An example of the development of metrics for a given application can be seen in Section 6.

- The language should support constructs for adaptation and learning to provide robustness and intelligence:
  C3L events can modify group membership and state variables in order to constrain the state space in such a way as to simulate adaptation and learning. C3L does not have a strict hierarchical command structure, so the chain of command is robust in that it should always be possible to elect a new supervisor, provided one is necessary. Possible future research with C3L may include new constructs for learning and adaptation, including constructs allowing biologically-inspired computation.

Chapter 4

Technical Results

This chaper presents the primary research product of this thesis, that being the operational semantics for C3L. Before the semantic rules can be presented, there must be a discussion of the nature of devices as considered in the C3L language, and a definition of the symbols used in the statement of the rules. Following the presentation of the semantics, some of the characteristics of the language are discussed, and several potential pathologies that can occur even in syntactically-correct C3L programs are identified, as well as a brief discussion of which of these can be checked for as part of static analysis. An abbreviated version of these results are included in [31].

4.1 C3L Semantics

The following is a presentation of the complete set of operational semantic rules for C3L.

Formal results in a language ensure that specifications in the language will automatically possess desirable properties. These results are based on the foundation of a formal semantics for the language. Following the presentation of the semantic rules are several formal results which enhance the utility of specifications in C3L.

4.1.1 Preliminaries

C3L is an event driven language. This requires formal semantics which evaluate the input queue of a given device. Since the input and output queues, variables, and group structures could all be modified at any step a full description of the context of a device is necessary to describe the semantics of event response. In addition there are several static tables of structures, such as control events and uncontrollables. These require a formal look-up tables to incorporate in the semantic structure. Lastly, the initial state of the

context should be well defined and hence this starting context, Γ_0, is also described.

Device Context

The following are some definitions for structures which are needed to construct the device context. These structures permit a direct and unambiguous way to describe changes in the device state during event response.

Definition: A device queue, Q is a string of objects of the form $q_1 q_2 \ldots q_n$. $Q = qQ'$ implies that q is the first (head) element of Q and Q' is the queue containing the remaining elements $q_2 q_3 \ldots q_n$. $Qq = Q'$ implies that q is the last (tail) element of $Q' = q_1 q_2 \ldots q_n q$.

Definition: A look-up table, L is a string of ordered pairs of objects of the form $\langle l_1, v_1 \rangle \langle l_2, v_2 \rangle \ldots \langle l_n, v_n \rangle$. With this notation, $L(l_i) = v_i$ and $L(l_i) := v_i'$ replaces the pair (l_i, v_i) with (l_i, v_i') in the string. Lastly, $L' = L\langle l, v \rangle$ inplies that the pair (l, v) is the last element of $L' = \langle l_1, v_1 \rangle \langle 1_2, v_2 \rangle \ldots \langle l_n, v_n \rangle \langle l, v \rangle$. We refer to the first member of each pair as a label and the second member as a value. Note, if $l \notin \{l_i\}$ then $L(l) = \epsilon$, the empty object.

Definition: A device list D is a set of strings, each a device name.

Now that the internal structures of a device context are defined, the outer structure can be described.

Definition: A device context is a 5-tuple, $\Gamma = (I, O, G, P, M)$, where I is the input queue which has pairs of input event names and pairs of source names and ordered value lists as objects, O is the output queue which has pairs of output events and pairs of device lists and ordered value lists as objects, G is a group look-up table which contains group names as labels and device lists as values, P is a parameter look-up table with parameter names as labels and parameter values as values, and M is a memory variable look-up table with variable names as labels and variable values as values. Note using the $I(i)$ notation, the source names may be accessed $Source(I(i))$ and the value list may be accessed by $Value(I(i))$.

Static Device Structures

Three additional components need to be incorporated to describe the operation of the device. These three aspects are the controllables, uncontrollables, and event responses of the device. These are all derived from the device specification. In order to manage these structures three look-up tables need to be generated.

Definition: The controllable look-up table, C is a look-up table with labels being the names of the listed controllables and values being an ordered pair with the first element as the condition for the event and the second as the sequence of assignments caused by the event. If either of these is empty this is

denoted by ϵ. $Pre(C(l))$ denotes the condition listed for the controllable named l. $Post(C(l))$ denotes the sequence of assignments listed for the controllable named l.

Definition: The uncontrollable look-up table, U is the analogous look-up table based on the uncontrollables defined in the device. Otherwise U has exactly the same structure and features as the controllable look-up table, C.

Definition: The control event look-up table, E is a look-up table with labels being the names of events listed in the control section of the device and values being an ordered pair of authorize list and pairs of command sequences and ordered lists of variable names. If either of these is empty this is denoted by ϵ. $Auth(E(l))$ denotes the authorize list listed for the event named l. $Comm(E(l))$ denotes the sequence of commands listed for the event named l. $Vari(E(l))$ denotes the ordered list of variable names. Note also if multiple elements matching the name of i exist, the one which also matches $tlist(i)$, as defined below, with its variable name list will be returned. This allows a limited polymorphism based on message variable.

Initial Context

In addition to the static look-up tables required to encapsulate the fixed components of the device, the initial state of the device context is also fixed. I_0 is the initial input queue. This contains some, possibly empty, sequence of events. O_0 is the initial output queue and is treated as empty. G_0 is a look-up table constructed in the following way from the sequence of group declarations:

1) Create a new pair using the group name on the left-hand side of the declaration.

2) Generate the superlist of the group, starting with an empty superlist. For each element of the declaration list:

a) if it is a string: add it to the superlist if it is not already present.

b) if it is a label, l: add each string in $G(l)$ to the superlist if it is not already present.

3) Generate the exceptlist of the group, starting with an empty exceptlist. For each element of the except list:

a) if it is a string: add it to the exceptlist if it is not already present.

b) if it is a label, l: add each string in $G(l)$ to the exceptlist if it is not already present.

4) Remove all strings in the exceptlist from the superlist. The resulting list is the value of the group.

Lastly, P_0 and M_0 are both generated by taking each item in the parameter or memory, respectively, list and adding an element to the empty look-up table for each parameter or variable name as a label and the value in parenthesis for the value. If no such value is present then the value is set to **NULL**.

Γ_0 is defined as $(I_0, O_0, G_0, P_0, M_0)$ and is the initial context of the device.

4.1.2 Formal Semantics of C3L

Table 4.1 lists the standard symbols used in the semantics of C3L.

x	≡ label
i	≡ input event
l	≡ slist
Commands	≡ an arbitrary list of commands
n	≡ integer value
r	≡ real number value
b	≡ boolean value (true or false)
e	≡ expression (of any type)
d	≡ conditional (boolean expression)
v	≡ value (of any type)
s	≡ string value (denotes the string "x_s", where x_s is a label)
l_d	≡ device list (in the correct device list format, as opposed to an slist)
$\hookrightarrow$	≡ evaluation of commands and context into a new context

Table 4.1: Symbols used in C3L Semantics

Table 4.2 contains definitions for the function names used in the semantics. These functions do not necessarily represent how, if at all, they are implemented, in the C3L interpreter, but they are notationally useful in the semantics.

gset	- takes two inputs, the master group list (G) and an slist (l), and returns a properly formatted device list as defined by the slist.
tlist	- takes two inputs, the memory lookup table (M) and an input event (i), and returns the list of types of its message variables.
type	- takes one input, a variable name (x), and returns the type of its associated value.
val	- takes one input, a variable name (x), and returns its associated value from the P or M lookup table.
generates	- indicates that an uncontrollable event i occurs
call	- indicates that a controllable event i is now called
update	- takes memory look-up table, an ordered variable names list, an ordered value list, and gives a new memory look-up table with the variables assigned the matching values in the respective lists.
compute	- takes an ordered list of expressions and evaluates them into an ordered list of values using the current context.

Table 4.2: Function Definitions for C3L Semantics

These first five rules govern the dynamics of the input queue. The first two of these rules are the ones which govern the reception of uncontrollable events. **Uncon-not-cond** describes the case when an uncontrollable occurs but its preconditions are false, and **Uncon-cond** describes the case when an uncontrollable occurs and is added to the queue. These are the only rules for which generates(i) is true, for all other rules this predicate is assumed to be false for all i.

$$(\textbf{Uncon-not-cond})\ \frac{\langle Pre(U(i)), \Gamma\rangle \hookrightarrow false\langle c, (I, O, G, P, M)\rangle \hookrightarrow \Gamma'}{\langle c, (I, O, G, P, M)\rangle \hookrightarrow \Gamma'}\quad i \in U \wedge generates(i)$$

$$(\textbf{Uncon-cond})\ \frac{\begin{array}{c}\langle Pre(U(i)), (I, O, G, P, M)\rangle \hookrightarrow true\\ \langle e_1, (I, O, G, P, M)\rangle \hookrightarrow v_1\\ \langle e_2, (I, O, G, P\langle p_1, v_1\rangle, M)\rangle \hookrightarrow v_2\\ \langle e_3, (I, O, G, P\langle p_1, v_1\rangle\langle p_2, v_2\rangle, M)\rangle \hookrightarrow v_3\\ \vdots\\ \langle e_k, (I, O, G, P\langle p_1, v_1\rangle\langle p_2, v_2\rangle \ldots \langle p_{k-1}, v_{k-1}\rangle, M)\rangle \hookrightarrow v_k\\ \langle c, (Ii, O, G, P\langle p_1, v_1\rangle\langle p_2, v_2\rangle \ldots \langle p_k, v_k\rangle, M)\rangle \hookrightarrow \Gamma'\end{array}}{\langle c, (I, O, G, P, M)\rangle \hookrightarrow \Gamma'}\quad \begin{array}{c} i \in U \wedge generates(i) \wedge\\ Post(U(x)) = \langle p_1, e_1\rangle \ldots \langle p_k, e_k\rangle\end{array}$$

The following rule, **Term-rule**, is the termination rule for execution in C3L. Practically, a program is never expected to terminate, but we can treat a program which has no commands in the list and an empty input queue as having terminated. The only caveat is that an input event may occur at some time in the future.

$$(\textbf{Term-rule})\ \frac{}{\langle \epsilon, (\epsilon, O, G, P, M)\rangle \hookrightarrow (\epsilon, O, G, P, M)}$$

These two rules describe the dequeueing of input events. **Input-notE** describes removing an event from the input queue that will not execute a command sequence, while **Input-E** describes removing an event that will execute a command sequence.

$$(\textbf{Input-notE})\ \frac{\langle \epsilon, (I, O, G, P, M)\rangle \hookrightarrow \Gamma'}{\langle \epsilon, (iI, O, G, P, M)\rangle \hookrightarrow \Gamma'}\quad \begin{array}{c} i \notin E \vee Source(I(i)) \notin Auth(E(i)) \vee\\ tlist(i) \neq Types(E(i))\end{array}$$

$$(\textbf{Input-E})\ \frac{\langle Comm(E(i)), (I, O, G, P, update(M, Vari(E(i)), Values(I(i))))\rangle \hookrightarrow \Gamma'}{\langle \epsilon, (iI, O, G, P, M)\rangle \hookrightarrow \Gamma'}\quad \begin{array}{c} i \in E \wedge Source(I(i)) \in Auth(E(i)) \wedge\\ tlist(i) \in Types(E(i))\end{array}$$

The following rules, from the **Raise-nolist-nomess** rule to the **m-assign** rule, describe the execution

of commands in C3L. Of these, the first six rules describe the functioning of the `raise` command. The **Raise-nolist-nomess** rule describes when an event is raised with no associated direct list or message variables. The **Raise-me-nomess** rule describes when an event is raised with a direct list containing the calling device and no message variables. The **Raise-notme-nomess** rule describes when an event is raised with a direct list not containing the calling device and no message variables. The **Raise-nolist-mess** rule describes when an event is raised with one or more message variables and no associated direct list. The **Raise-me-mess** rule describes when an event is raised with a direct list containing the calling device and one or more message variables. The **Raise-notme-mess** rule describes when an event is raised with a direct list not containing the calling device and one or more message variables.

$$(\textbf{Raise-nolist-nomess})\ \frac{\langle c, (I\langle x, \langle me, \epsilon\rangle\rangle, O\langle x, \langle all, \epsilon\rangle\rangle, G, P, M)\rangle \hookrightarrow \Gamma'}{\langle raise\ x\ c, (I, O, G, P, M)\rangle \hookrightarrow \Gamma'}$$

$$(\textbf{Raise-me-nomess})\ \frac{\langle c, (I\langle x, \langle me, \epsilon\rangle\rangle, O\langle x, \langle (gset(G,l) - me), \epsilon\rangle\rangle, G, P, M)\rangle \hookrightarrow \Gamma'}{\langle raise\ x\ direct\ l\ c, (I, O, G, P, M)\rangle \hookrightarrow \Gamma'}\quad me \in gset(G,l)$$

$$(\textbf{Raise-notmenot-nomess})\ \frac{\langle c, (I, O\langle x, < gset(G,l), \epsilon\rangle\rangle, G, P, M)\rangle \hookrightarrow \Gamma'}{< raise\ x\ direct\ l\ c, (I, O, G, P, M)\rangle \hookrightarrow \Gamma'}\quad me \notin gset(G,l)$$

$$(\textbf{Raise-nolist-mess})\ \frac{\begin{array}{c}\langle c, (I\langle x, \langle me, compute(e_1, \ldots, e_k)\rangle\rangle,\\ O\langle x, \langle all, compute(e_1, \ldots, e_k)\rangle\rangle, G, P, M)\rangle \hookrightarrow \Gamma'\end{array}}{\langle raise\ x(e_1, \ldots, e_k)\ c, (I, O, G, P, M)\rangle \hookrightarrow \Gamma'}$$

$$(\textbf{Raise-me-mess})\ \frac{\begin{array}{c}\langle c, (I\langle x, \langle me, compute(e_1, \ldots, e_k)\rangle\rangle,\\ O\langle x, \langle (gset(G,l) - me), compute(e_1, \ldots, e_k)\rangle\rangle, G, P, M)\rangle \hookrightarrow \Gamma'\end{array}}{\langle raise\ x(e_1, \ldots, e_k)\ direct\ l\ c, (I, O, G, P, M)\rangle \hookrightarrow \Gamma'}\quad me \in gset(G,l)$$

$$(\textbf{Raise-notme-message})\ \frac{\langle c, (I, O\langle x, \langle gset(G,l), compute(e_1, \ldots, e_k)\rangle\rangle, G, P, M)\rangle \hookrightarrow \Gamma'}{\langle raise\ x(e_1, \ldots, e_k)\ direct\ l\ c, (I, O, G, P, M)\rangle \hookrightarrow \Gamma'}\quad me \notin gset(G,l)$$

The **label-not-C** describes the case when a device attempts to add a non-controllable event to its input queue. The **label-not-cond** rule describes the case when a device attempts to perform a controllable event, but the preconditions are not met. The **label-mess** describes the case when a device successfully performs a controllable event with associated message variables, and the **label-nomess**

describes the case when a device successfully performs a controllable event with no associated message.

$$(\textbf{Label-notC})\ \frac{\langle c, \Gamma\rangle \hookrightarrow \Gamma'}{\langle x\ c, \Gamma\rangle \hookrightarrow \Gamma'}\quad x \notin C$$

$$(\textbf{Label-not-cond})\ \frac{\langle Pre(C(x)), \Gamma\rangle \hookrightarrow false \langle c, \Gamma\rangle \hookrightarrow \Gamma'}{\langle x\ c, \Gamma\rangle \hookrightarrow \Gamma'}\quad x \in C$$

$$(\textbf{Label-mess})\ \frac{\begin{array}{c}\langle Pre(C(x)), (I, O, G, P, M)\rangle \hookrightarrow true \\ \langle e_1, (I, O, G, P, M)\rangle \hookrightarrow v_1 \\ \langle e_2, (I, O, G, P\langle p_1, v_1\rangle, M)\rangle \hookrightarrow v_2 \\ \langle e_3, (I, O, G, P\langle p_1, v_1\rangle\langle p_2, v_2\rangle, M)\rangle \hookrightarrow v_3 \\ \vdots \\ \langle e_k, (I, O, G, P\langle p_1, v_1\rangle\langle p_2, v_2\rangle \ldots \langle p_{k-1}, v_{k-1}\rangle, M)\rangle \hookrightarrow v_k \\ \langle c, \Gamma(I, O, G, P\langle p_1, v_1\rangle\langle p_2, v_2\rangle \ldots \langle p_k, v_k\rangle, M)\rangle \hookrightarrow \Gamma'\end{array}}{\langle x(e'_1, \ldots, e'_j)\ c, \Gamma\rangle \hookrightarrow \Gamma'}\quad \begin{array}{c} x \in C \wedge calls(x) \wedge \\ Post(C(x)) = \langle p_1, e_1\rangle \ldots \langle p_k, e_k\rangle\end{array}$$

$$(\textbf{Label-nomess})\ \frac{\begin{array}{c}\langle Pre(C(x_1)), (I, O, G, P, M)\rangle \hookrightarrow true \\ \langle e_1, (I, O, G, P, M)\rangle \hookrightarrow v_1 \\ \langle e_2, (I, O, G, P\langle p_1, v_1\rangle, M)\rangle \hookrightarrow v_2 \\ \langle e_3, (I, O, G, P\langle p_1, v_1\rangle\langle p_2, v_2\rangle, M)\rangle \hookrightarrow v_3 \\ \vdots \\ \langle e_k, (I, O, G, P\langle p_1, v_1\rangle\langle p_2, v_2\rangle \ldots \langle p_{k-1}, v_{k-1}\rangle, M)\rangle \hookrightarrow v_k \\ \langle c, \Gamma(I, O, G, P\langle p_1, v_1\rangle\langle p_2, v_2\rangle \ldots \langle p_k, v_k\rangle, M)\rangle \hookrightarrow \Gamma'\end{array}}{\langle x\ c, \Gamma\rangle \hookrightarrow \Gamma'}\quad \begin{array}{c} x \in C \wedge calls(x) \wedge \\ Post(C(x)) = \langle p_1, e_1\rangle \ldots \langle p_k, e_k\rangle\end{array}$$

The **Gadd** and **Grem** rules describe when a device adds or removes devices from a group.

$$(\textbf{Gadd})\ \frac{\langle c, (I, O, G(x, G(x) \cup gset(G, l)), P, M)\rangle \hookrightarrow \Gamma'}{\langle gadd\ x\ l\ c, (I, O, G, P, M)\rangle \hookrightarrow \Gamma'}$$

$$(\textbf{Grem})\ \frac{\langle c, (I, O, G(x, G(x) - gset(G, l)), P, M)\rangle \hookrightarrow \Gamma'}{\langle grem\ x\ l\ c, (I, O, G, P, M)\rangle \hookrightarrow \Gamma'}$$

The next four rules govern behavior of the **choice** command: The **Choice-no-cor-T** rule applies when there are no **cor** statements and the condition is true, the **Choice-no-cor-F** rule when there are no **cor**s and the condition is false, the **Choice-cor-T** rule when there are one or more **cor** statements and the condition is true, and the **Choice-cor-F** rule when there are one or more **cor**s and the condition is false.

$$(\textbf{Choice-no-cor-T})\ \frac{\langle d, \Gamma\rangle \hookrightarrow true\langle Commands\ c, \Gamma\rangle \hookrightarrow \Gamma'}{\langle choice\ d\ Commands\ end\ c, \Gamma\rangle \hookrightarrow \Gamma'}$$

$$(\textbf{Choice-no-cor-F})\ \frac{\langle d, \Gamma\rangle \hookrightarrow false\langle c, \Gamma\rangle \hookrightarrow \Gamma'}{\langle choice\ d\ Commands\ end\ c, \Gamma\rangle \hookrightarrow \Gamma'}$$

$$(\textbf{Choice-cor-T})\ \frac{\langle d_1, \Gamma\rangle \hookrightarrow true\langle Commands_1\ c, \Gamma\rangle \hookrightarrow \Gamma'}{\langle choice\ d_1\ Commands_1\ cor\ d_2\ Commands_2 \ldots cor\ d_k\ Commands_k\ end\ c, \Gamma\rangle \hookrightarrow \Gamma'}$$

$$(\textbf{Choice-cor-F})\ \frac{\langle d_1, \Gamma\rangle \hookrightarrow false\langle choice\ d_2\ Commands_2 \ldots cor\ d_k\ Commands_k\ end\ c, \Gamma\rangle \hookrightarrow \Gamma'}{\langle choice\ d_1\ Commands_1\ cor\ d_2\ Commands_2 \ldots cor\ d_k\ Commands_k\ end\ c, \Gamma\rangle \hookrightarrow \Gamma'}$$

The **m-assign** rule describes memory assignment statements. There is no corresponding **p-assign** rule, since commands cannot modify the device parameters.

$$(\textbf{m-assign})\ \frac{\langle e, \Gamma\rangle \hookrightarrow v\langle c, (I, O, G, P, M\langle x, v\rangle) \hookrightarrow \Gamma'}{\langle x := e;\ c, \Gamma\rangle \hookrightarrow \Gamma'}$$

The above rules all are structured by the following command meta-rule. This rule is used to reason about the general operation of commands in C3L.

$$(\textbf{Cmd-meta-rule})\ \frac{\langle c, \Gamma''\rangle \hookrightarrow \Gamma'}{\langle Commands\ c, \Gamma\rangle \hookrightarrow \Gamma'}$$

The following 28 rules describe evaluation of conditional statements. These are fairly self-explanatory. The and/or operators support short circuit evaluation (denoted by the "SC" rules). Also, all of the comparison operators can be applied to pairs of strings, with the result determined by the lexicographic ordering of the strings.

$$(\textbf{true})\ \frac{}{< true, \Gamma > \hookrightarrow true}$$

$$(\textbf{false})\ \frac{}{< false, \Gamma > \hookrightarrow false}$$

$$(\textbf{and-SC-F})\ \frac{< d_1, \Gamma > \hookrightarrow false}{< d_1\ and\ d_2, \Gamma > \hookrightarrow false}$$

$$(\textbf{and-2nd-F})\ \frac{< d_1, \Gamma > \hookrightarrow true < d_2, \Gamma > \hookrightarrow false}{< d_1\ and\ d_2, \Gamma > \hookrightarrow false}$$

$$(\textbf{and-T})\ \frac{< d_1, \Gamma > \hookrightarrow true < d_2, \Gamma > \hookrightarrow true}{< d_1\ and\ d_2, \Gamma > \hookrightarrow true}$$

$$(\textbf{or-SC-T})\ \frac{< d_1, \Gamma > \hookrightarrow true}{< d_1\ or\ d_2, \Gamma > \hookrightarrow true}$$

$$(\textbf{or-2nd-T})\ \frac{< d_1, \Gamma > \hookrightarrow false < d_2, \Gamma > \hookrightarrow true}{< d_1\ or\ d_2, \Gamma > \hookrightarrow true}$$

$$(\textbf{or-F})\ \frac{< d_1, \Gamma > \hookrightarrow false < d_2, \Gamma > \hookrightarrow false}{< d_1\ or\ d_2, \Gamma > \hookrightarrow false}$$

$$(\textbf{not-true})\ \frac{< d, \Gamma > \hookrightarrow true}{< not(d), \Gamma > \hookrightarrow false}$$

$$(\textbf{not-false})\ \frac{d, \Gamma > \hookrightarrow false}{< not(d), \Gamma > \hookrightarrow true}$$

$$(\textbf{eq-true})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2}{< e_1 = e_2, \Gamma > \hookrightarrow true}\quad v_1 = v_2 \vee v_1 =_L v_2$$

$$(\textbf{eq-false})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2}{< e_1 = e_2, \Gamma > \hookrightarrow false}\quad v_1 \neq v_2 \vee v_1 \neq_L v_2$$

$$(\textbf{less-true})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2}{< e_1 < e_2, \Gamma > \hookrightarrow true}\quad v_1 < v_2 \vee v_1 <_L v_2$$

$$(\textbf{less-false})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2}{< e_1 < e_2, \Gamma > \hookrightarrow false}\quad v_1 \not< v_2 \vee v_1 \not<_L v_2$$

$$(\textbf{greater-true})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2}{< e_1 > e_2, \Gamma > \hookrightarrow true}\quad v_1 > v_2 \vee v_1 >_L v_2$$

$$(\textbf{greater-false})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2}{< e_1 > e_2, \Gamma > \hookrightarrow false}\quad v_1 \not> v_2 \vee v_1 \not>_L v_2$$

$$(\textbf{leq-true})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2}{< e_1 <= e_2, \Gamma > \hookrightarrow true}\quad v_1 \leq v_2 \vee v_1 \leq_L v_2$$

$$(\textbf{leq-false})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 \quad < e_2, \Gamma > \hookrightarrow v_2}{< e_1 <= e_2, \Gamma > \hookrightarrow false}\quad v_1 > v_2 \vee v_1 >_L v_2$$

$$(\textbf{geq-true})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 \quad < e_2, \Gamma > \hookrightarrow v_2}{< e_1 >= e_2, \Gamma > \hookrightarrow true}\quad v_1 \geq v_2 \vee v_1 \geq_L v_2$$

$$(\textbf{geq-false})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 \quad < e_2, \Gamma > \hookrightarrow v_2}{< e_1 >= e_2, \Gamma > \hookrightarrow false}\quad v_1 < v_2 \vee v_1 <_L v_2$$

$$(\textbf{noteq-true})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 \quad < e_2, \Gamma > \hookrightarrow v_2}{< e_1 != e_2, \Gamma > \hookrightarrow true}\quad v_1 \neq v_2 \vee v_1 \neq_L v_2$$

$$(\textbf{noteq-false})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 \quad < e_2, \Gamma > \hookrightarrow v_2}{< e_1 != e_2, \Gamma > \hookrightarrow false}\quad v_1 = v_2 \vee v_1 =_L v_2$$

$$(\textbf{label-true})\ \frac{v = P(x)}{< x, \Gamma > \hookrightarrow true}\quad v = true$$

$$(\textbf{label-false})\ \frac{v = P(x)}{< x, \Gamma > \hookrightarrow false}\quad v = false$$

$$(\textbf{string-in-label-T})\ \frac{}{< \text{“}x_s\text{”}\ in\ x, \Gamma > \hookrightarrow true}\quad (x, l_d) \in G.x_s \in l_d$$

$$(\textbf{string-in-label-F})\ \frac{}{< \text{“}x_s\text{”}\ in\ x, \Gamma >\hookrightarrow false}\quad (x, l_d) \notin G \vee x_s \notin l_d$$

$$(\textbf{label-in-label-T})\ \frac{}{< x_1\ in\ x_2, \Gamma >\hookrightarrow true}\quad (x_1, l_{d1}) \in G.(x_2, l_{d2}) \in G.\forall x \in l_{d1} x \in l_{d2}$$

$$(\textbf{label-in-label-F})\ \frac{}{< x_1\ in\ x_2, \Gamma >\hookrightarrow false}\quad (x_1, l_{d1}) \notin G \vee (x_2, l_{d2}) \notin G \vee \exists x \in l_{d1} s.t. x \notin l_{d2}$$

The next 14 rules describe the behavior of arithmetic operators and math functions. These are also fairly self-explanatory. The typing works as in C or C++, that is, if both operands are integers, the result is an integer; otherwise, the result is a real.

$$(\textbf{int})\ \frac{}{< n, \Gamma >\hookrightarrow n}$$

$$(\textbf{real})\ \frac{}{< r, \Gamma >\hookrightarrow r}$$

$$(\textbf{add-int})\ \frac{< e_1, \Gamma >\hookrightarrow v_1 < e_2, \Gamma >\hookrightarrow v_2}{< e_1 + e_2, \Gamma >\hookrightarrow n}\quad v_1, v_2 \in \mathbf{Z} \wedge n = v_1 + v_2$$

$$(\textbf{add-real})\ \frac{< e_1, \Gamma >\hookrightarrow v_1 < e_2, \Gamma >\hookrightarrow v_2}{< e_1 + e_2, \Gamma >\hookrightarrow r}\quad v_1 \in \mathbf{R} \vee v_2 \in \mathbf{R} \wedge r = v_1 + v_2$$

$$(\textbf{sub-int})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2}{< e_1 - e_2, \Gamma > \hookrightarrow n} \quad v_1, v_2 \in \mathbf{Z} \wedge n = v_1 - v_2$$

$$(\textbf{sub-real})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2}{< e_1 - e_2, \Gamma > \hookrightarrow r} \quad v_1 \in \mathbf{R} \vee v_2 \in \mathbf{R} \wedge r = v_1 - v_2$$

$$(\textbf{mult-int})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2}{< e_1 * e_2, \Gamma > \hookrightarrow n} \quad v_1, v_2 \in \mathbf{Z} \wedge n = v_1 * v_2$$

$$(\textbf{mult-real})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2}{< e_1 * e_2, \Gamma > \hookrightarrow r} \quad v_1 \in \mathbf{R} \vee v_2 \in \mathbf{R} \wedge r = v_1 * v_2$$

$$(\textbf{div-int})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2}{< e_1/e_2, \Gamma > \hookrightarrow n} \quad v_1, v_2 \in \mathbf{Z} \wedge v_2 \neq 0 \wedge n = \left\lfloor \frac{v_1}{v_2} \right\rfloor$$

$$(\textbf{div-real})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2}{< e_1/e_2, \Gamma > \hookrightarrow r} \quad v_1 \in \mathbf{R} \vee v_2 \in \mathbf{R} \wedge v_2 \neq 0 \wedge r = \frac{v_1}{v_2}$$

$$(\textbf{div-zero})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2}{< e_1/e_2, \Gamma > \hookrightarrow NULL} \quad v_2 = 0$$

$$(\textbf{neg-int})\ \frac{< e, \Gamma > \hookrightarrow v}{< -e, \Gamma > \hookrightarrow n >} \quad v \in \mathbf{Z} \wedge n = -v$$

$$(\textbf{neg-real})\ \frac{< e, \Gamma > \hookrightarrow v}{< -e, \Gamma > \hookrightarrow r >} \quad v \in \mathbf{R} \wedge r = -v$$

$$(\textbf{math-fcn})\ \frac{< e_1, \Gamma > \hookrightarrow v_1 < e_2, \Gamma > \hookrightarrow v_2 \ldots < e_k, \Gamma > \hookrightarrow v_k}{< math(e_1, e_2, \ldots, e_k), \Gamma > \hookrightarrow v} \quad n \geq 0 \wedge v = math(v_1, v_2, \ldots, v_k)$$

The final two rules describe the array dereferencing operation. If it is supplied with a valid subscript n, the result is the nth element of the array's value parameter. If it is supplied with an invalid subscript, the result is NULL.

$$(\textbf{deref-valid})\ \frac{val(x) = (v_1, v_2, v_3, \ldots, v_k)}{< x[n], \Gamma > \hookrightarrow v} \quad v = v_n.n \in \mathbf{Z} \wedge 0 \leq n \leq k$$

$$(\textbf{deref-fail})\ \frac{val(x) = (v_1, v_2, v_3, \ldots, v_k)}{< x[n], \Gamma > \hookrightarrow NULL} \quad n \notin \mathbf{Z} \vee n < 0 \vee n > k$$

Using these semantics the formal properties of C3L can be explored, and several formal results, shown below, can be proven for the language.

4.2 Static Analysis of C3L Programs

Static analysis of a program attempts to determine some properties regarding the behavior of that program without actually executing the program. Some aspects of static program analysis are: checking for syntactic correctness, checking for type correctness, and basic control flow analysis. Each of these static analyses has been implemented for use with C3L. Static analysis, especially in the area of type correctness, is dependent on having a complete and correct semantics for the language. Checking for syntax correctness is a well-understood and simple problem, and will not be further discussed here.

Since C3L enforces the determination of variable types prior to runtime, it is a statically-typed language. C3L is also a weakly-typed language, meaning that there are exceptions to the strict enforement of type rules in the language, but only in the case of storing "integer"-type data in a "real"-type variable. In no other case is type conversion allowed. Therefore, C3L is a type-safe programming language.

Since C3L uses a plant/controller model, it is desirable to show that the program never enters a

"blocking state." This is an undecidable problem, since it can be reduced to the Halting Problem, as shown in [31]. However, the following is an incomplete list of possible causes of blocking in a C3L program that have been identified:

1. All uncontrollable preconditions evaluate to false. (Implies that no state transitions can ever occur in the plant. Controller state transitions are still possible)

2. All controllable preconditions evaluate to false. (No controllable exits from any state.)

3. No uncontrollables have event handlers, or, no uncontrollable has an event handler which raises a command. (This implies that no controllables are ever called, and so there is not a controllable exit from any state.)

4. No controllable events are raised for the group containing "me". (There are no controllable exits from any state.)

5. A variable is used in a precondition without being initialized. (There may be no resulting state transition, or the transition may be undefined.)

6. An event is raised to an undefined group. (It is unknown whether any device will receive the controllable.)

7. Input queue pathologies (Note: The "DoS" and "DDoS" pathologies used herein refer to "Denial of Service" and "Distributed Denial of Service", respectively. These terms are used in the normal way; a Denial of Service attack originates from one device, and a Distributed Denial of Service attack originates from many devices. The "attack" notation is used by convention, but this is not meant to imply malicious intent; these pathologies result from incorrectly designed plant models.):

    - Self DoS - One internal events calls two or more internal events, at least one of which is the same as the original event.
    - Initiating DDoS - One received event causes two or more events to be raised to the same direct list.
    - Inviting DDoS - One event raised to a direct list causes a response from every member of the list (This pathology is only applicable in a multiple-device setting).

Not all of these possible pathologies can be tested for prior to run time (*i.e.*, they are outside the realm of static analysis). However, pathologies 3, 5, 6, and the Self DoS component of 7 all can be, and are, identified statically. This gives the user at least some degree of certainty that the control flow of

his or her program is correct. This process is performed by a C++ program which also checks for the syntactic correctness of the C3L program. This program is not included here, in the interests of space and simplicity.

Chapter 5

Language Verification and Proofs

Given that there is a complete semantics for the C3L language, as specified in Section 4.1.2, it is possible to use these semantic rules to prove statements about programs written in C3L. These proofs can refer to specific programs, or as the proofs in this section do, to all programs written in C3L.

The first property that will be proved, event fairness, is the property that any event that is placed on the input queue of a C3L device must necessarily be dequeued in some finite amount of time. This allows us to say that "event starvation" is not possible for a device controlled by a C3L program. This is desirable because it assures the user that any device that can be received will eventually be handled, and also because it is a component in the proof of prefix-closed controllability that follows.

A language is prefix-closed controllable if it satisfies the following two constraints: First, that a specification cannot cause a command to execute if the state of device should prohibit its execution, and second, that the specification cannot prohibit an event from occuring, if that event is outside of the control of the device. Full controllability requires that a language possess a third property, that execution of a program cannot terminate in an undesirable state. As shown in [31], it is impossible to show full controllability in C3L, or any other language of equal expressive power (that is, any language that is at least Turing-complete and can have marked states). However, it is possible to show that a language is prefix-closed controllable, and so programs in that language do not depend on controlling uncontrollable events. The second proof in this section proves that this is the case for C3L. These results can also be found in [31].

5.1 Proof of Event Fairness in C3L

In a network of autonomous devices, each device is operating concurrently with the other devices. In order to ensure communication and basic functionality it is important to prevent the major risks of concurrent systems from affecting the network. One of the most important concerns in a concurrent system is starvation, where a particular process or event is left waiting indefinitely. This problem is especially common in the pathology of busy waiting, where a device will continually check for an event, and in doing so prevent that event from accessing the resources needed to be noticed. The property of a language that no events are starved is called event fairness.

In the C3L language the formal model prevents event starvation by ensuring that every event will eventually be removed from the input queue. This is the reason the C3L language lacks any explicit looping commands. This is also the reason why the functionality of procedure calls in other languages is replaced with that of raising events in C3L. Repeated raising of events acts as tail recursion, and by the nature of the C3L input queue, any events preceding the newly raised event occur without interruption. These design considerations are necessary for the following proof of event fairness.

Event Fairness: An event i, placed on the input queue at arbitrary time t, is always dequeued in finite time.

Remark: In order to prove finite queue waiting, it was assumed that the enqueue operation takes 0 processor time. If this were not the case, it would be theoretically possible that an infinite, or at least sufficiently large, number of uncontrollable events could consume all available processing time with enqueueing, effectively performing a DoS attack on the device. In practice, this does not represent a considerable weakening of the proof, since a certain degree of sparseness of uncontrollable events is observed. That is, uncontrollable events occur at most finitely often.

Proof: There are two device resources which can affect the dequeueing of an input event: The input queue, and the command sequence at any given time. These two resources will be taken as separate cases, and in each case, will be inductively proved to become empty in finite time. Since both resources being empty is the prerequisite for dequeueing of the desired event i, this is sufficient to prove finite queue waiting for any event.

Case I: Command List (proof by well-founded induction (WFI))

Let ϵ be the minimal element of any list of commands, and let the partial order relation be defined as the "is a sublist of" relation. There are 6 distinct commands in the C3L language: **label** [outmessage], **raise** label [outmessage] [**except** slist], **gadd** label slist, **grem** label slist, assign, and choice. None of these commands, when executed, can add to the command list. Also, none of these commands can ever be non-terminating in normal operation. Finally, all of these commands execute in finite time. Given these conditions, the execution of any command causes the length of the command list to decrease by at least 1 after a finite time step. Then, by WFI on the sublist relation, any list of commands can be reduced to the empty list, ϵ, in finite time.

Case II: Input Queue (proof by well-founded induction)

Let I_{pre} denote those events that make up the input queue immediately before time t, when event i is added to the queue, and let I_{post} denote the list of events that are added to the queue after time t. Only I_{pre} has an effect on the waiting time of event i, since those events in I_{post} cannot be dequeued before event i, or moved up in the list to be in front of event i. Therefore, only the state of I_{pre} is relevant to this proof. Any uncontrollable events that occur, or any events that are placed on the queue by execution of a command, are placed at the tail of I_{post}, and are therefore irrelevant to the proof. Now all that remains to show is: For any event i placed on the queue at time t, I_{pre} can be reduced to the empty list, ϵ, in finite time.

Let ϵ denote the minimal element of I_{pre}, and again let the partial order relation be defined as the "is a sublist of" relation. The dequeue operation is defined such that whenever the command list is empty, the event at the head of the input queue will be removed from the queue and handled (if applicable). As shown in Case I, the command list will always become empty in finite time. Therefore, a dequeue operation will always be able to take place after some finite time interval. By the definition of the dequeue operation, dequeueing causes the length of the input queue to be reduced by 1. Then, by WFI on the sublist relation, I_{pre} can be reduced to ϵ in finite time. Then, after the command list again becomes empty, in a finite amount of time, i is dequeued. Therefore, an event i placed on the input queue at arbitrary time t is always dequeued in finite time.

5.2 Proof of Prefix-Closed Controllability in C3L

In most languages significant care must be taken to ensure that a specification does not cause a command to execute when the state of the sensor node prohibits its execution. It is also important that the specification does not prohibit the occurrence of an event, if that event is outside of the control of the sensor node. A specification which satisfies both of these properties is called prefix-closed controllable. Full controllability requires an additional constraint, namely that execution cannot terminate in an undesired state. This requires the specification to both incorporate termination and to consider some sensor node states to be undesirable for termination, otherwise controllability is just prefix-closed controllability as described above [25]. Full controllability and its relationship to C3L and other languages will be discussed below.

One of the major formal properties of C3L is that all syntactically correct specifications in C3L are prefix-closed controllable. A sketch of this result is shown below.

Prefix-Closed Controllability: For all parameter configurations, all allowable uncontrollables can be received. ($\forall$ generates, $\forall$ P, $\forall t$, if $\langle Pre(U(i)), \Gamma\rangle \hookrightarrow$ true at time t and $i \in U$, then generates(i) at time $t \rightarrow I_{t+\epsilon} = I_t i$)

Proof: $I_{t+\epsilon} = I_t i \Leftrightarrow \exists$ a rule s.t. for any sequence of commands, evaluating those commands under $\Gamma_t = (I_t, O, G, P, M)$ and under $\Gamma_{t+\epsilon} = (I_t i, O', G', P', M')$ gives the same result state $\Leftrightarrow \exists$ an inference rule s.t.

$$\frac{\ldots \langle Commands, (I_t i, O', G', P', M')\rangle \hookrightarrow \Gamma'}{\langle Commands, (I_t, O, G, P, M)\rangle \hookrightarrow \Gamma'}$$

where $\Gamma_{t+\epsilon} = (Ii, O, G, P, M)$ and $\Gamma_t = (Ii, O', G', P', M')$. Since the statement to be proved is generates(i) $\rightarrow I_t = I_{t+\epsilon} i$, and the only rules for which the generates predicate can be true are the **(Uncon-not-cond)** and **(Uncon-cond)** rules, and the proof statement assumes that $\langle\ Pre(U(i)), \Gamma_t\rangle \hookrightarrow$ true, **(Uncon-cond)** is the only rule that can be applied. $\langle Pre(U(i)), \Gamma_t\rangle \hookrightarrow$ true, $i \in U$, and generates(i,t) are all part of the assumptions in the proof statement, and these three conditions are the necessary and sufficient conditions for an allowed uncontrollable. Since all expressions evaluate to values, this rule can be applied. Therefore, the **(Uncon-cond)** rule will always be applied in this situation, and is of the type required above. This is sufficient to show that for all parameter configurations, all allowable uncontrollables can be received.

Chapter 6

Demonstration: Dynamic Locality-Based Election

The semantics for C3L presented in Section 4.1.2 and formal proofs such as those presented in Chapter 5 serve as verification for C3L; that is, they make formal the properties and behaviors of the language, such that these properties and behaviors can be checked against the specifications for the language. Validation of a general use programming language such as C3L is less intuitive. Validation of a programming language is only meaningful in terms of validation for a specific application. In this case, a discussion of how this application validates C3L in terms of the characteristics stated in Section 3.3 will follow the presentation of the experimental results.

6.1 Experimental Setup

As a demonstration of C3L programming, the following program was developed. This program performs a detection-based cluster head election among a group of networked sensor nodes, which assumes that these nodes are occupying random distances from a specified target. The algorithm seeks to elect the nearest node to the target to act as the cluster head for the group, without using any global information. Frequency of target detections is used as a proxy for physical distance; that is, if a node receives more frequent detections of the target, it is assumed to be closer to the target. The text of the C3L program appears below:

```
device Locality_Elector
```

```
parameter

    Active integer (0)
    Last_Active integer (0)
    Doubt integer (5)

uncontrollable

    Start () ()
    Restart () ()
    Detection () (Active := (Active + 1);)

controllable

    Sync () (Last_Active := Active;)
    Inhibit (Active > 4;) (Active := (Active - 5);)
    Reduce_Doubt (Doubt > 0;) (Doubt := (Doubt - 1);)
    Increase_Doubt (Doubt <= 10;) (Doubt := (Doubt + 1);)
    Reset () (Active := 0; Last_Active := 0; Doubt := 5;)

group

    Potentials := all except me end
    Subordinates := end
    Supervisor := end

control

    event Start
        raise Update direct me end
    end

    event Restart
```

```
    Reset
    gadd Potentials all except me end
    grem Subordinates all end
    grem Supervisor all end
end

event Update
    choice (Active > 10;)
        raise Assert_Sup direct Potentials end
        Reduce_Doubt
        Reduce_Doubt
    end
    choice ((Active - Last_Active) < (Active / 2);)
        Inhibit
        Increase_Doubt
        Increase_Doubt
    end
    Sync
    raise Update direct me end
end

event Assert_Sup
authorize Potentials end
    choice (Doubt > 6;)
        raise Join direct you end
        raise Reject direct Subordinates end
        grem Subordinates Subordinates end
        grem Potentials Potentials end
        gadd Supervisor you end
    end
    Increase_Doubt
end
```

```
    event Join
    authorize all except me end
        choice (you in Potentials;)
            grem Potentials you end
            gadd Subordinates you end
            Reduce_Doubt
            raise Assert_Sup direct Potentials end
        cor (not(you in Subordinates);)
            raise Reject direct you end
        end
    end

    event Reject
    authorize Supervisor end
        gadd Potentials all except me end
        grem Supervisor you end
    end
end
```

Roughly, this program functions as follows: The Start event causes an event to call the Update event, which is then performed continually for the duration of the execution. Any detection increments a device's Active count. If the Active count exceeds 10, the device believes that it is a valid potential supervisor, and communicates this to every other device. If the device does not receive detections frequently (*i.e.* if Active - Last_Active is sufficiently small), it doubts that it can be a valid supervisor. When a device receives another device's Assert_Sup declaration, it either increases its degree of doubt, or if its doubt is already sufficiently great, it joins the group headed by the raiser of Assert_Sup. If a device becomes a subordinate to another device, and has subordinates of its own, it releases those subordinates, so that they are free to join another supervisor. When all devices have the same supervisor, the algorithm stops. This algorithm will always elect a cluster head, however, since it is stochastic (that is, random), it may take an arbitrarily large amount of time.

6.2 Results

Simulations were performed in order to determine some of the characteristics of the program presented in Section 6.1. First, the simulation was run with only a single device; 20 runs at each distance from 1 to 10, with a maximum number of steps of 10000. The runtime was capped in order to prevent pathological cases from monopolizing simulation time. The goal of this simulation was to establish a baseline of how long it would take a node at any given distance to recognize that it could be a potential supervisor. The results are summarized in the following table:

Distance	**Mean no. of steps**	**Median no. of steps**	**Percentage of failures**
1	21.65	27	0
2	22.8	24	0
3	21.45	15	0
4	17.9	51	0
5	45.5	21	0
6	151.45	267	0
7	5690	4586	0
8	10000	10000	100
9	10000	10000	100
10	10000	10000	100

Table 6.1: Single Device Simulation Results

The data of Table 6.1 can be summarized as follows. For distances from the target of 5 or less, the device always succeeded in recognizing itself as a potential supervisor, in a small number of steps (on order 10). For a device at distance 6 from the target, the device always recognized itself as a potential supervisor, although it took appreciably longer than for devices closer to the target (on order 100). For a device at distance 7 from the target, it was essentially random whether the device would recognize itself as a supervisor. The high difference between the median and the mean for devices at distance 7 implies that the cases in which the device did not recognize itself as a supervisor perhaps excessively color the mean value. For devices at distance 8 or further, a device never recognized itself as a potential supervisor. It should be noted that for such a small number of runs, it is not clear that more precise distinctions can be made for the number of steps needed at each distance than the order comparisons made above.

In addition, simulations were run for groups ranging between 2 and 20 devices, 1000 runs for each group size, with a maximum of 1000 steps placed on each run, again, to prevent particularly bad cases from monopolizing simulation time. The nodes were placed randomly, with a quadratic distribution of distances (*i.e.* 1 percent of devices are located at distance 1 from the target, 4 percent at distance 2 or nearer, 9 percent at distance 3 or nearer, etc.) The following relationships were observed during these simulations:

1. **Runtime (measured in number of steps) vs. Number of devices:**

   Independent variable: Number of devices

   Dependent variable: Runtime

2. **Runtime vs. Standard deviation of distances**

   Independent variable: Standard deviation of distances

   Dependent variable: Runtime

   It is thought that a more tightly grouped set of devices might have a different set of performance characteristics using this algorithm than a more widely-scattered group. The standard deviation of distances from the target is used to measure this characteristic of "tightly-grouped-ness".

3. **Quality of Supervisor vs. Number of devices:**

   Independent variable: Number of devices

   Dependent variable: Quality of supervisor

   Quality of supervisor must be well-defined for this comparison to be meaningful. I choose to define it as follows: 1 - (Fraction of devices with distance $\langle$ supervisor's distance). This is a simple metric; the best choice for supervisor would always rate as 1, and the worst would rate as 0. It does not penalize the supervisor for ties in quality, since a choice among tied nearest nodes is entirely arbitrary. It is possible that this metric is not harsh enough; for example, if the algorithm were to choose the 5th best supervisor out of 10 nodes, it's quality would still be rated as .5, although in actuality, a rating this low would reflect a spectacularly bad choice on the part of the algorithm. A square of this measurement might more accurately reflect this.

4. **Quality of supervisor vs. Standard deviation of distances**

   Independent variable: Standard deviation of distances

   Dependent variable: Quality of supervisor

   See (3) for a definition of quality of supervisor.

Results are presented in the following charts and discussions.

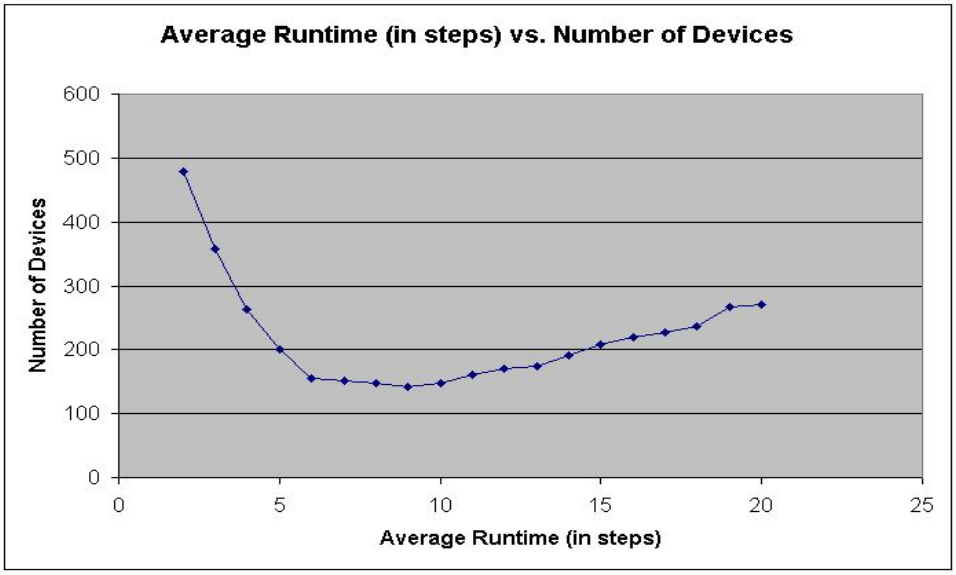

Figure 6.1: Simulation Results Relating Runtime to Number of Devices

Figure 6.1 shows the relationship between runtime and number of devices using this algorithm. The interesting characteristic of these data is that there is a "sweet spot" between five and ten devices in the network. Networks with fewer than 5 devices have particularly bad runtimes as a result of "thrashing", wherein multiple devices each consider themselves to be the supervisor, and the network takes a very large amount of time to reach a consensus on the best supervisor, causing a disproportionately high rate of failure with a maximum of steps.

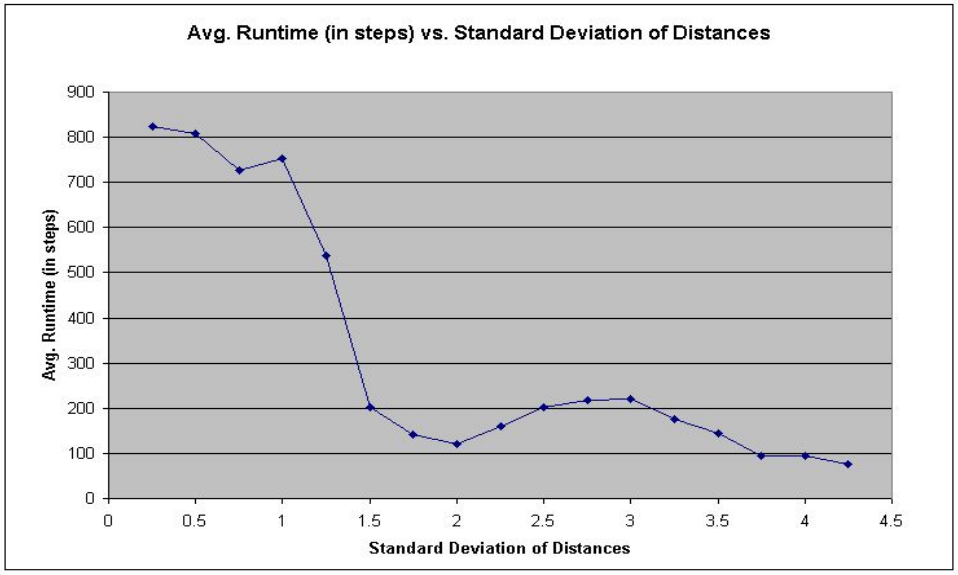

Figure 6.2: Simulation Results Relating Runtime to Standard Deviation of Distances

Figure 6.2 shows the relationship between supervisor quality and number of devices. Again, this chart shows a "sweet spot" around ten devices. For fewer than twelve devices, the algorithm tends to elect an average supervisor that is better than the second best possible supervisor in any single run. As the number of devices increases, the algorithm elects a relatively poor supervisor with greater frequency.

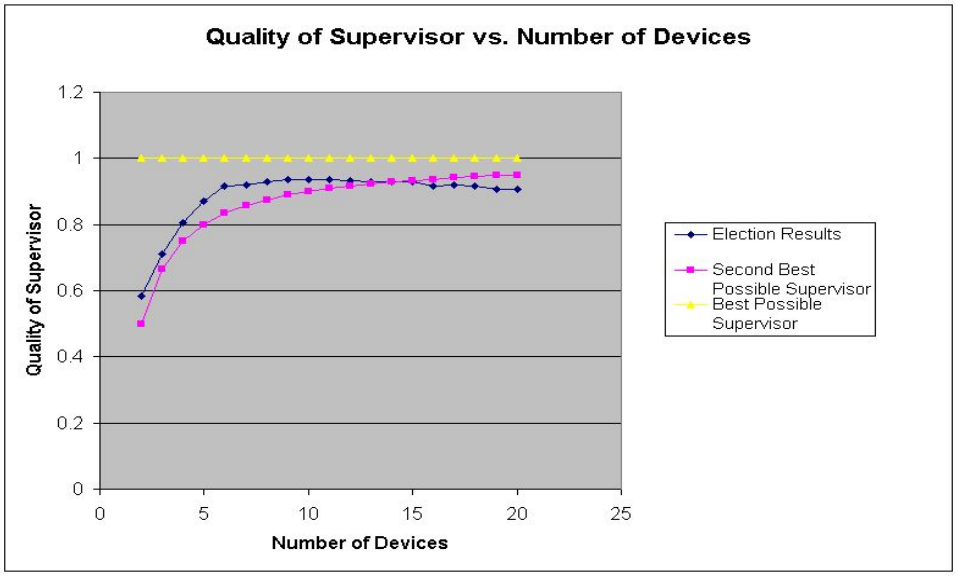

Figure 6.3: Simulation Results Relating Quality of Supervisor to Number of Devices

Figure 6.3 shows the relationship between runtime and the standard deviation of distances from the target. This chart shows a very clear trend, in that very tightly-grouped networks of devices (standard deviation of distances less than 1.5), runtimes are exceedingly high (above 700 steps on average),

indicating a very large amount of failures to reach consensus on a supervisor. For standard deviation of distances greater than 1.5, the runtimes are consistently low. This indicates a strong correlation of runtime with a threshold value of standard deviation of distances, somewhere between 1 and 1.5.

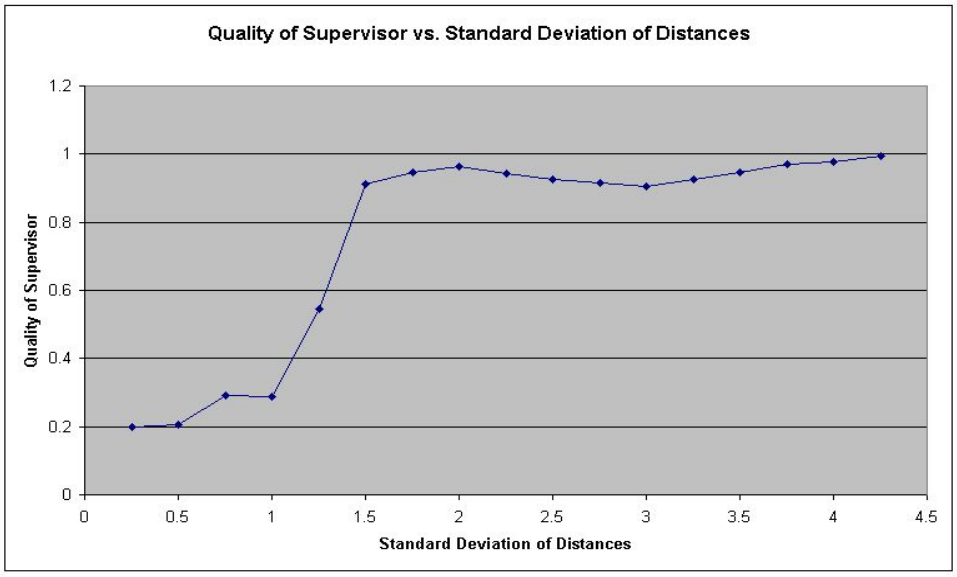

Figure 6.4: Simulation Results Relating Quality of Supervisor to Standard Deviation of Distances

Figure 6.4 shows the relationship between supervisor quality and the standard deviation of distances from the target. In light of the previous chart, the results are as expected. Below the threshold value (between 1 and 1.5), supervisor quality is very poor, indicating a large amount of failures. Above that threshold value, supervisor quality is very close to 1. This reinforces the notion that performance has a very strong correlation to the "clumping" in a network, but only insofar as whether the "clumping" is more or less than some known threshold.

6.3 Validation of Results

The six specific motivating characteristics of the C3L language, as stated in Section 3.3, are dynamic organization, accomodation of uncontrollable events, portability, formality, expressiveness, and extensibility. This section seeks to show that the example program shown previously in this chapter validates C3L for this specific application by performing the stated task and taking advantage of these characteristics. This program obviously demonstrates dynamic organization of groups in C3L. Devices join and leave groups in real time in response to target detections and communication events. This example is also dependent upon the accomodation of uncontrollable events, as a detection of the target is an uncontrollable event, which causes modification of device parameters and dynamic changes to device groups. This

specific example does not directly show the portability of the C3L language, however, the C3L program is interpreted in C++, so it could run on any platform that has an existing C++ compiler, and it is also portable across different C3L devices, as long as they share the variables and events presented in the program text above. This program does exhibit formality, by possessing at least the properties of event fairness and prefix-closed controllability. It also exhibits the necessary expressiveness, demonstrating the use of recursion in the `Update` event, and also demonstrating the device communication and group formation functionality in the language. Extensibility is not validated by this example, owing to the fact that this example could be entirely composed in the existing C3L language. An example which would validate the extensibility of the language would be, say, adding a global stack to the language.

Chapter 7

Future Research Directions

While implementation of the base of C3L, as described in [2], is largely concluded, there are several extensions to the language that may be implemented in the future.

First, as discussed in [2], C3L is naturally suited to use in simulation applications. Development in this area is underway, in which the C3L interpreter is being integrated with the open-source Player/Stage platform. In this integrated system, called Robotalk, a user could specify the behavior of multiple devices using the C3L language, define the parameters of the environment, and run the C3L program, with results being displayed in the Player/Stage interface.

Other potential extensions to the language include the addition of a polymorphic stack construct to the device context, the addition of records to the language, the addition of structured parallelism to the language, and the addition of logical programming constructs to the language. With any of these additions, the semantics of the language would also have to be amended to describe the new behavior of the language.

Other extensions, mentioned earlier, include biologically-inspired programming constructs and support for mobile code.

Bibliography

[1] Eberbach, E., and Phoha, S., "SAMON: Communication, Cooperation and Learning of Mobile Autonomous Robotic Agents," Proceedings of the 11th International Conference on Tools and Artificial Intelligence, Chicago, IL, Nov. 9-11, 1999.

[2] Phoha, S. and Schmiedekamp, M., "A Common Control and Communications Language for Distributed C3 of Dynamically Networked Autonomous Devices," Sensor Networks Operations, IEEE Press, In Press.

[3] Phoha, S., "Robot Control Languages," Internal ARL Technical Report: Literature Survey, June 1999, University Park, PA.

[4] Phoha, S., Eberbach, E., Stover, J., Culver, L., and Peluso, E. "A Proposal for a Generic Behavior Message-Passing Language for Collaborative Missions of Heterogeneous Autonomous Underwater Vehicles," Distributed Simulation-Based Design of Collaborative AOSN Missions (SAMON II) Project, Task Report 5.5, Nov. 27, 1998.

[5] Frenger, P., "Part One: A Review of Robotics Languages," in SIGPLAN Forth Report: Robot Control Techniques. IEEE. 1997.

[6] Ingrand, F.F., Chatila, R., Alami, R., Robert, F., "PRS: a high-level supervision and control language for autonomous mobile robots," Proceedings of the 1996 13th IEEE International Conference on Robotics and Automation. Part 1 (of 4), April 22-28 1996, v 1, Minneapolis, MN, p 43-49.

[7] Simmons, R., Apfelbaum, D., "Task description language for robot control," IEEE International Conference on Intelligent Robots and Systems Innovations in Theory, Practice, and Applications. Part 3 (of 3), v 3, Oct 13-17 1998, Victoria, Canada.

[8] Brooks, R.A., "The Behavior language: User's Guide" AI Memo 1227. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, April 1990.

[9] Van Deursen, A., Klint, P., and Visser, J., "Domain-Specific Languages: An Annotated Bibliography." http://homepages.cwi.nl/~arie/papers/dslbib/. February 2000, Amsterdam, The Netherlands.

[10] Foundation for Intelligent Physical Agents, "FIPA Communicative Act Library Specification." Standard Specification, Foundation for Intelligent Physical Agents, December 2002. Available at http://www.fipa.org/specs/fipa00037/SC00037J.html.

[11] Eberbach, E., and Phoha, S., "SAMON: Communication, Cooperation, and Learning of Mobile Autonomous Robotic Agents," Proceedings of the 11th International Conference on Tools and Artificial Intelligence, Chicago, IL, Nov. 9-11, 1999.

[12] Aknine, S., "Reasoning Structures for Multi-Agent Meta-Programming," in Proceedings of the International Symposium on Intelligent Control/Intelligent Systems and Semiotics. IEEE. 1999.

[13] Atkin, M., King, G., and Westbrook, D., "Hierarchical Agent Control: A Framework for Defining Agent Behavior," in AGENTS. ACM. 2001.

[14] Beckhoum, K., Chen, L., and Clapworthy, G. "A Logical Approach to High-Level Agent Control," in AGENTS. ACM. 2001.

[15] Browning, B., Go, J., Kaminka, G., Veloso, M., and Vu, T., "MONAD: A Flexible Architecture for Multi-Agent Control," in AAMAS. ACM. 2003.

[16] Denko, M., "The Use of Mobile Agents for Clustering in Mobile Ad-Hoc Networks," in SAICSIT. ACM. 2003.

[17] Deshpande, A., Göllü, A., and Semenzato, L. "The SHIFT Programming Language for Dynamic Networks of Hybrid Automata," IEEE Transactions on Automatic Control, Vol. 10, No. 4, p 584-587, April 1998.

[18] Saraswat, V., and Rinard, M., "Concurrent Constraint Programming," Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 232-245, San Francisco, CA, December 1989.

[19] Saraswat, V, "Concurrent Constraint Programming," Logic Programming and Doctoral Dissertation Award Series, MIT Press, Cambridge, MA, March 1993.

[20] Gupta, V., Jagadeesan, R., and Panangaden, P. "Stochastic Processes as Concurrent Constraint Programs," Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 189-202, San Antonio, TX, 1999.

[21] Saraswat, V., Jagadeesan, R., and Gupta, V. "Foundations of Timed Concurrent Constraint Programing", Proceedings of the 9th Annual IEEE Symposium on Logic In Computer Science, pp. 272-285, IEEE Computer Press, July 2004.

[22] Romero, N. "Mobile Concurrent Constraint Programming," Lecture Notes in Computer Science, Vol. 2328/2002, pp. 811-818, Springer-Verlag, Heidelberg, 2002.

[23] Crawford, L., Fromherz, M., Shang, Y., and Zhang, Y., "A General Constraint-Based Control Framework with Examples in Modular Self-Reconfigurable Robots," in Proceedings of the International Conference on Intelligent Robots and Systems. IEEE. 2002.

[24] Saraswat, V., Gupta, V., Jagadeesan, R., "Default Timed Concurrent Constraint Programming," in POPL. ACM. 1995.

[25] Zhong, H and Wonham, W.M., "On the Consistency of Hierarchical Supervision in Discrete-Event Systems," IEEE Transactions on Automatic Control, vol. 35, no. 10, October 1990, pp. 1125-34.

[26] S. Phoha, E. Peluso and R. Brooks, "A Constructivist Theory of Distributed Intelligent Control of Complex Dynamic Systems," JFACC Symposium, San Diego, CA, Nov. 15-16, 1999.

[27] Peluso, E., Phoha, S., and Goldstine, J., "Normal Processes for Modeling the Desired Behavior of Distributed Autonomous Discrete Event Systems," Journal of Automata, Languages, and Combinatorics, Vol. 7, No. 1, pp. 127-142, 2002.

[28] Peluso, E., "A Hierarchical Structure of Interacting Automata for Modeling Battlefield Dynamics: Controllability and Formal Specification," Ph.D. Dissertation, Department of Computer Science, The Pennsylvania State University, 1996.

[29] Phoha, S., Eberbach, E., and Brooks, R., "Coordination of Multiple Heterogeneous Autonomous Undersea Vehicles (AUVs)," Special Heterogeneous Multi-Robot Systems Issue of Autonomous Robots, Kluwer, Winter 2000.

[30] Phoha, S., Peluso, E., and Culver, R.L., "A High Fidelity Ocean Sampling Mobile Network (SAMON) Simulator," invited paper, IEEE Journal of Oceanic Engineering, Special Issue on Autonomous Ocean Sampling Networks, Vol. 26, Issue 4, pp. 646-653, Jan. 2002.

[31] Schmiedekamp, M., Skarbez, R., Phoha, S., "Formal Properties and Validation of the Common Control and Communication Language (C3L)," Sensor Networks Operations, IEEE Press, In Press.

[32] Phoha S., Schmiedekamp, M, "Common Control and Communication Language (C3L) in Autonomous Networks of Unmanned Underwater Vehicles," Sensor Networks Operations, IEEE Press, In Press.

[33] Schmidt, D., "Programming Language Semantics," in ACM Computing Surveys, v 28, n 1, March 1996.

[34] Winskel, G., "The Formal Semantics of Programming Languages: An Introduction," MIT Press. Cambridge, MA. 1993

[35] Wikipedia, "Fourth-generation programming language," June 3, 2004, http://en.wikipedia.org/wiki/Fourth-generation_programming_language.

[36] Naur, P., et al. "Revised Report on the Algorithmic Language ALGOL 60." Communications of the ACM, Vol. 6, Issue 1, pp. 1- 17, Jan. 1963.

Appendix A: Academic Vita

Richard T Skarbez

708 Marion St

Browndale, PA 18421

814 883 7760

skarbez@gmail.com

Education:

Bachelor of Science in Computer Engineering with minors in Mathematics and Philosophy, The Pennsylvania State University, University Park, PA, December 2004

Thesis Title: A Presentation of the Semantics and Formal Properties of C3L, an Event-Driven Distributed Control Language

Academic Honors and Awards:

- Member of the Schreyer Honors College
- National Merit Scholarship Winner
- Rensselaer Medal for Outstanding Achievement in Math and Science
- Bausch & Lomb Honorary Science Award
- Xerox Humanities/Social Science Award

Publications:

Schmiedekamp, M., Skarbez, R., Phoha, S., "Formal Properties and Validation of the Common Control and Communication Language (C3L)" Sensor Networks Operations, IEEE Press, In Press.

Professional Experience:

Student Researcher, Applied Research Lab, University Park, PA

March 2004 to Present

While at ARL, I have been working on the C3L project in the Emergent Sensor Plexus research group. As part of this research, I developed the semantics for the C3L programming language, wrote several formal proofs concerning the properties of the C3L language, designed and coded several pieces of software used in the implementation of the C3L language, including a parser and a pathology checker. Also, I contributed to several papers regarding C3L, and am developing the software design methodology for the C3L language.

Intern - Instructional Technology Department, Forest City Regional School District, Forest City, PA

June 2003 through August 2003

Working at Forest City Regional School District, I repaired, configured, built, and installed computers for use in classrooms, I updated, installed, and used various network administration tools, and interacted with the district administrators on a daily basis.

Intern - Front-End-Of-Line Processing (Drills), IBM Microelectronics Division, Endicott, NY

May 2002 through August 2002

During this internship, my primary project was the development of a new procedure to automatically generate drill machine programs for the deep-deletion of panel defects. This is a procedure which was previously done manually by an engineer, by which panel defects, such as short circuits, can be drilled out of the panel, and the circuit can be rerouted over the surface of the panel, salvaging the circuit board. In the development of this process, I worked with an interdepartmental team of engineers. In addition to this project, I reviewed and revised documentation for the testing of drills and drilled panels.

Intern - Digital Video Process Group, IBM Microelectronics Division, Endicott, NY

May 2001 through August 2001

During my internship with DVPG, my primary task was to design and implement a relational database to store test procedures and test results, replacing the existing hardcopy filing system. In addition to the design of this database, I administered both hardware and software validation tests, and wrote test code, using assembly, an internal command language, and C++.

Technician, Does Not Compute, Forest City, PA

July 1999 through August 2000

At Does Not Compute, my primary responsibility was to provide telephone and in-person technical support for users of NEP Internet. Other tasks included web developing, upgrading and repairing computers, and ordering equipment.

Activities and Leadership Experience:

- President/Editor-in-Chief, PHROTH, Penn State's Humor Magazine
- Recording Secretary and Webmaster, HKN, Electrical & Computer Engineering Honors Society
- Member, Golden Key International Honour Society
- Member, competing debater, Penn State Debate Team (defunct)
- Attendee, 8th annual GE/PSU Student Leadership Conference
- One of two System Architects, LUDWIG project (Fall 2004)