Flexible Embedded Software Networks

by

Marc d'Entremont

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTERS OF SCIENCE

in the Department of Computer Science

© Marc d'Entremont, 2003

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part by photocopy or other means, without the permission of the author.

Supervisor: Dr. J. Jahnke

ABSTRACT

My thesis investigates the collaboration and evolution of micro controller networks. In the thesis, I have developed a framework for flexibly connecting embedded systems, called microSynergy. The framework gives to users the ability unify a network of micro controller-based devices into logically defined collaborative networks. It establishes a communication infrastructure across a variety of networking protocols and allows dynamic reconfiguration of the micro controller network by means of a visual language. The infrastructure includes a mechanism for the construction and/or reuse of stateful coordination logic that manages the inter-communication of the network enabled devices and software components to create new functionalities and behaviours from a set of predefined software and hardware components.

microSynergy looks at only a few of the issues, such as how to integrate devices with minimal impact on the code base, integrating devices of various protocols, building easy to understand and modifiable networks of devices that can act as a unified system. It include the design, implementation and analysis, through an informal user study, of a specific implementation of visual programming paradigm and its relation to collaborative networks.

Table of Contents

Abstract ii

Table of Contents iv

List of Figures ix

List of Tables xi

Acknowledgement xii

Dedication xiii

1 Introduction 1
- 1.1 microSynergy: High Level Description . . . 3
- 1.2 Document Structure . . . 5

2 Related Work 7
- 2.1 Ubiquitous Computing . . . 7
  - 2.1.1 A short history of UC . . . 7
  - 2.1.2 Benefits of UC . . . 9
    - 2.1.2.1 Instant Information . . . 10
    - 2.1.2.2 Cost Reduction . . . 10
    - 2.1.2.3 Safety and Security . . . 10
    - 2.1.2.4 Aggregation of Devices . . . 11
    - 2.1.2.5 Scalability and Stability . . . 11
  - 2.1.3 Problems with Ubiquitous Computing . . . 12
    - 2.1.3.1 Privacy and Security . . . 12
    - 2.1.3.2 Social Changes . . . 12
    - 2.1.3.3 Safety . . . 13

2.2 Components . . . 13
2.3 Visual Languages . . . 14
2.3.1 Connection-Based Programming . . . 15
2.3.2 Specification and Description Language . . . 15

3 Requirements of microSynergy **18**
3.1 Collaboration of Controllers . . . 18
3.2 Reuse of Logic . . . 19
3.3 Evolution . . . 19
3.4 Changes to microCommander . . . 19
3.5 Ease of Use . . . 20
3.6 Language Choice . . . 20

4 System Design **21**
4.1 microCommander . . . 21
4.1.1 Software Architecture . . . 22
4.1.2 Hardware Architecture . . . 22
4.1.3 Hardware Resource Distribution . . . 23
4.1.4 mVisual . . . 24
4.1.5 mTarget . . . 25
4.1.6 mHub . . . 27
4.2 microSynergy . . . 28
4.2.1 The Editor . . . 28
4.2.2 Runtime . . . 30
4.2.3 Initialization . . . 32
4.3 Messages . . . 32
4.4 File Formats . . . 34
4.4.1 Connector Description Language (CDL) . . . 35
4.4.2 Connector Execution Language (CEL) . . . 35

5 Visual Language **37**
5.1 Motivation . . . 37
5.2 microSynergy's Visual Language . . . 37
5.2.1 Connection-Based Programming . . . 37

5.2.2 State-Based Modelling 38
5.2.3 Syntax 38
5.3 Language Core Semantics 39
5.3.1 Connector Components 39
5.3.2 Embedded Components 40
5.3.3 Threads 40
5.3.4 Gates 40
5.4 Language Core Syntax 41
5.4.1 System Level 41
5.4.1.1 Controllers 41
5.4.1.2 Connectors 42
5.4.1.3 Channels 44
5.4.2 Connector Level 44
5.4.2.1 Threads 46
5.4.2.2 States 48
5.4.2.3 Inputs 48
5.4.2.4 Priority Inputs 48
5.4.2.5 Outputs 48
5.4.2.6 Conditionals 48
5.5 Extended Language Concepts 49
5.5.1 Semantics 50
5.5.1.1 Templates 50
5.5.1.2 Hyper-States 50
5.5.1.3 Default Inputs 51
5.5.2 Extended Syntax 51
5.5.2.1 Templates 51
5.5.2.2 Hyper-states 52
5.5.2.3 Default Input 52
5.5.3 Pragmatics 54
5.5.3.1 Home automation 54
5.5.3.2 Industrial Automation 55
5.6 Application Scenario 56

5.6.1 Details 57

6 Evaluation and Case Study **59**

6.1 Background 59

6.2 Results of Pilot Study 59

6.3 Improvements to Prototype 60

6.3.1 Templates 60

6.3.2 Improvements to the user interface 60

6.4 Goals of Main Study 62

6.4.1 Scaling to industrial-strength 62

6.4.2 Intuitive Visual Modelling Paradigm 62

6.4.3 Prototype as a Method of Discovery 63

6.5 Methodology 63

6.6 Tasks 64

6.6.1 General Tasks 64

6.6.2 Logic Tasks 65

6.6.3 Template Tasks 65

6.7 Task Evaluation 66

6.7.1 Quantitative Evaluation 66

6.7.2 Qualitative Evaluation 66

6.7.3 Task Timings Analysis 70

6.7.4 Qualitative Observation 72

6.7.5 General Tasks 72

6.7.5.1 Synchronize with the network 72

6.7.5.2 Make a new connector 72

6.7.5.3 Connect two controllers 72

6.7.6 Logic Tasks 72

6.7.6.1 Specify the Forwarder Logic 72

6.7.6.2 Specify the And Logic 73

6.7.7 Template Tasks 73

6.7.7.1 Import the Forwarder Template 73

6.7.7.2 Mapping of Forwarder Logic 73

6.7.7.3 Use the And Template 73

6.8 General Comments 73
6.9 Conclusions 74

7 Reflections and Future Work **76**
7.1 Contributions 76
7.2 Future Work 77
7.2.1 Arbitrary Data Types 77
7.2.2 Encapsulation of Logic 77
7.2.3 Atomization of Connector Logic 78
7.2.4 Dynamic Logic Loading 78
7.2.5 Roles and Auto-Mapping 78
7.2.6 Model Checking and Verification 79
7.2.7 SOAP Services 79
7.2.8 Real-Time Support 80
7.2.9 Definition of a Global Component Network 80
7.2.10 History State 80
7.2.11 Quantitative Evaluation 81

References **82**

Appendix A An Example of a CDL File **84**

Appendix B Annotated CEL Sample **87**

List of Figures

Figure 1.1 A high-level view of organization of a microSynergy Network . 4

Figure 2.1 Taxonomy of UC research 8
Figure 2.2 A Simple Nassi-Schneiderman diagram 14

Figure 4.1 A high-level overview of the microCommander architecture . . 22
Figure 4.2 The microCommander Hardware Architecture 23
Figure 4.3 The microCommander Software Architecture 23
Figure 4.4 Message Sequence Diagram portraying the ping-pong protocol 24
Figure 4.5 The embedded component panel 25
Figure 4.6 Visuals that can be attached to microCommander Components 25
Figure 4.7 A typical embedded component configuration dialog 26
Figure 4.8 A high-level view of a microCommander controller 27

Figure 5.1 An annotated sample of a system level program 42
Figure 5.2 System Level Core Syntax 43
Figure 5.3 A system level view with visible gates and connections shown 44
Figure 5.4 Inside a connector 45
Figure 5.5 Connector Level syntax 47
Figure 5.6 An example of the use of templates with the connections expanded 49
Figure 5.7 Additions to system level syntax 52
Figure 5.8 Sample screen shot of extended language concepts 53
Figure 5.9 Additions to connector level syntax 53
Figure 5.10 Mapping of Gates in microSynergy 54
Figure 5.11 A interaction diagram describing the interaction of components 57

Figure 6.1 Templates hold logic for later reuse 61
Figure 6.2 The gate mapping interface 61

Figure 6.3 Graph of perceived difficulties 67
Figure 6.4 Forwarder logic 68
Figure 6.5 And logic 69
Figure 6.6 Graph of the timings based upon user class 71

List of Tables

Table 4.1 Table of CEL Commands 36

Table 5.1 Allowable connections matrix 46

Acknowledgement

I would like to express my appreciation to all members of the microSynergy team. Special thanks to Andrew McNair, and Jens Jahnke, who have been instrumental to the conceptual development and physical development of microSynergy, and without whom the project would have been greatly constrained. Thank you to all the readers, especially Tricia d'Entremont who has provided greatly needed insight into the writing of this document.

Dedication

To my wife, Tricia.

Chapter 1

Introduction

Computers are commonplace. They exist in many domains and interact with humans and the environment in an ever increasing number of ways. Indeed, the proliferation of PCs has placed a computer in everyone's home. What most lay people do not realize is that they already had computers in their home. They did not perceive them as such as they are in the same form mechanically-based devices. In today's world, VCRs, microwave ovens, coffee makers, clocks and many other devices have small computers in them. It is these, so called, *simple devices* upon which the microSynergy project focuses.

There are two classes of simple devices: those based upon circuits, and those based upon microprocessors. The microSynergy project is directed at microprocessor-based devices as they are reprogramable and therefore can evolve and act in a fundamentally more flexible fashion than their circuit-based counterparts. These microprocessor-based devices are often referred to as micro controllers, or simply as controllers, as they often control physical mechanisms exterior to the computing system itself. The flexibility of controller-based devices is a significant strength over inflexible circuit-based systems. Circuit-based simple devices have physical circuits that act in a very fixed fashion to perform specific tasks. As the circuits can not be easily changed they must replaced when the scenario varies greatly.

Many simple devices have traditionally been designed with minimalist resources. This simplicity kept them small and cost effective. It is uncommon for them to have large amounts of memory or computing power, their communication resources are often limited to slow-speed serial communication channels, such as RS232 and can only recall a program via EPROM [1]. The new generation of devices entering the con-

[1] Erasable Programable Read Only Memory

sumer market have significant resources in comparison. They can be reprogrammed easily via fast, robust communication channels such as Ethernet, Firewire™, or Bluetooth™. They can also have significant computational resources, such as those used for decoding music or video.

Currently many simple devices act completely independently of other devices in their local vicinity. If the devices could communicate with one another they could act in a smarter fashion or even collaborate on tasks.

If every device possessing a micro processor could share its unused resources with neighboring devices, significant resources could be leveraged. There are many barriers to leveraging these resources. microSynergy looks at only a few of the issues, such as how to integrate devices with minimal impact on the code base, integrating devices of various protocols, designing easy to understand designs and modify systems of devices.

Integrating simple devices is not a simple task. With access to the source code and/or the proper infrastructure, two devices can be made to communicate. This, although commonly done, is a non-trivial task. Once two devices can communicate, adding communication capabilities to communicate with a third device further complicates the task and likely to increase the complexity of the code on each controller. This increasing quantity and complexity of code is an issue to be avoided. microSynergy integrates devices using their pre-existing network communications infrastructure to minimize the changes to the device and speed development.

Developing devices tends to be a long, expensive and complex effort with many steps involved in creating, distributing and disseminating programs to their execution platform. The tasks involved require deep understanding of related technology and tends to be error prone. Simplifying the development cycle will speed up the development process. The C programming language is the defacto standard for the programming of controllers due to its power, simplicity and performance. microSynergy builds on this with a runtime system designed to receive new programs and interpret them.

Maintenance of embedded code is also a complex task. Developers often must read and understand cryptic code in order to fix a problem or add a capability. Using a higher-level language is often not acceptable, as higher-level languages, such

as Visual Basic, usually require more runtime resources then lower-level languages. As processors increase in power and memory, higher-level languages may become acceptable in the performance realm, but there are still issues in flexibility, portability and convenience. microSynergy establishes a reasonable balance in this area.

Simple devices have largely been designed for only one task, this simplicity has allowed for simple designs, large production runs and low costs. Devices are difficult to expand and/or update. By minimizing changes to the embedded code base devices can be quickly and cost effectively be made microSynergy compatible and take part in microSynergy networks. By federating many devices, available resources would be expanded, new functionalities, and behaviours would be added to the system simply by adding a new device. As the resources available to each system increase the federation could gain more capabilities. microSynergy leverages the networking of devices to share resources.

1.1 microSynergy: High Level Description

microSynergy's goal is to investigate the collaboration and evolution of micro controller networks and in so doing, to make the collaboration of simple devices easier, faster and less error prone. Figure 1.1 describes a very high-level view of microSynergy architecture. microSynergy connects these devices and coordinates them so that they may act as a unified system. When a user interacts with a device other elements of the system can be made aware of the interaction and can be requested to act in a specific fashion. microSynergy supports collaborative networks of controllers by coordinating the communications of controllers. Controller interaction is defined by a generated program that is interpreted by a runtime interpretive environment. It defines thin interfaces for the sharing and discovery of networked resources, and a visual language (VL) that simplifies the tasks of understanding, designing and programming networks of controllers. Programs defined in the VL determine when the components on specified controllers communicate with each other. The VL also makes the interrelationships between controllers clear and simple to change.

The visual programs are converted to binary programs that can be downloaded to a special controller, the *master controller*, that has the role of coordinating other

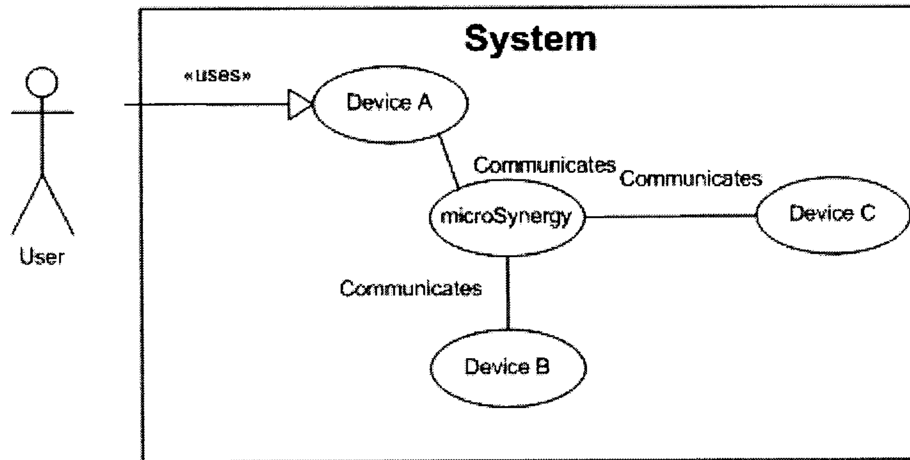

Figure 1.1. *A high-level view of organization of a microSynergy Network*

controllers. The master controller acts as a bridge for multi-protocol communication between all devices on the network and as an interpreter for the programs created by microSynergy editor as shown in 4.2.1.

This architecture allows the federated groups of devices to act as a unified entity, yet still act independently at performing their specified tasks. Sets of controllers can also be grouped in their own separate environments so that they are effectively isolated from one another while still sharing the same coordinator. The coordination of devices into a unified environment, or partitions thereof, places microSynergy as an infrastructure for ubiquitous computing, as described in section 2.1.

The simple devices are usually limited to one communication protocol. In order to coordinate as many different types of simple devices as possible, we have added multi-protocol support to the master controller. Currently controllers with support for TCP/IP, RS232, Bluetooth™, CANBus and X11[2] can take part in microSynergy networks. This allows the communication protocol to be abstracted and therefore become largely irrelevant. Newly added devices, that support an error-free communication protocol, can be automatically discovered by microSynergy

Non-microSynergy controllers that have communication infrastructure require only

[2]The X10 protocol is a radio frequency protocol specification that allows wireless communication and communication via power lines

slight modifications to interact with a microSynergy network. A thin wrapper that handles messages and performs function calls is all that is necessary to integrate into a microSynergy network. This design keeps the complexity of communication separate from the code required for the performance of the device's task, while allowing the task performed by the device to be minimally affected by communication requirements.

When more extensive communication logic, such as conditional routing of messages and messaging multiple destinations, is required communication can be done via a connecting component. A *connecting component* allows the developer to specify modified finite state machines that coordinate communication.

Once the controllers can communicate, they must also be able to act upon the communications of other controllers in the network. This implies that activities are being done without the interaction or intervention of a person.

1.2 Document Structure

Chapter 2 Discusses related work. The key areas discussed are ubiquitous computing, distributed computing, component technologies, distributed middle-ware and visual languages.

Chapter 3 Requirements and goals of the project are discussed.

Chapter 4 Discusses the architecture of the prototype. This includes the an explanation of related areas of microCommander, and how microSynergy integrates with microCommander. Explanations of the software and hardware components that make up microCommander and microSynergy and their distribution throughout the system are provided leading to a clear understanding of the overall architecture of microSynergy. Also included is a description of message types and formats used in communication, as well as the intermediate and final formats created by microSynergy editor and interpreted by microSynergy runtime.

Chapter 5 Discusses the motives leading to the choice of a visual language and the specifics of the visual language. The syntax, semantics and pragmatics of the language are established. Finally a use case scenario is provided to illustrate

the practical use of the VL.

Chapter 6 Evaluates the prototype's usability in an attempt to determine the viability of the VL and the VL programming infrastructure. It explores the merits to future study in this domain.

Chapter 7 Discusses contributions of the project,and future work.

Chapter 2

Related Work

2.1 Ubiquitous Computing

"Ubiquitous computing is the method of enhancing computer use by making many computers available throughout the physical environment, but making them effectively invisible to the user[1]." We have the beginnings of small environments that profess to function in this manner. For the most part *ubiquitous computing* (UC), also known as *pervasive computing* by the business community, has not yet become reality. There are many potential reasons for this. Some reasons may be that computers are still in their infancy, that devices are too expensive to distribute in a ubiquitous fashion, or that devices are too awkward to use or to power. As technology improves, many of these hurdles are increasingly irrelevant. Technology exists that allows people to interact with computers in a continuous fashion in almost any location on the planet. While that interaction still does not always occur in a completely invisible fashion, users demand that computers ease their workload. The integration of these small, inexpensive devices into a collaborative framework is key to UC.

2.1.1 A short history of UC

UC represents a major evolutionary step in a line of work dating back to the mid-1970's. Two distinct earlier steps in the evolution are distributed systems and mobile computing[2].

The realm of computing has been in consistently evolving since its inception. Computing started on a single system. In the late 60's computers were given the ability to communicate. Tasks could then be distributed across many different computers.

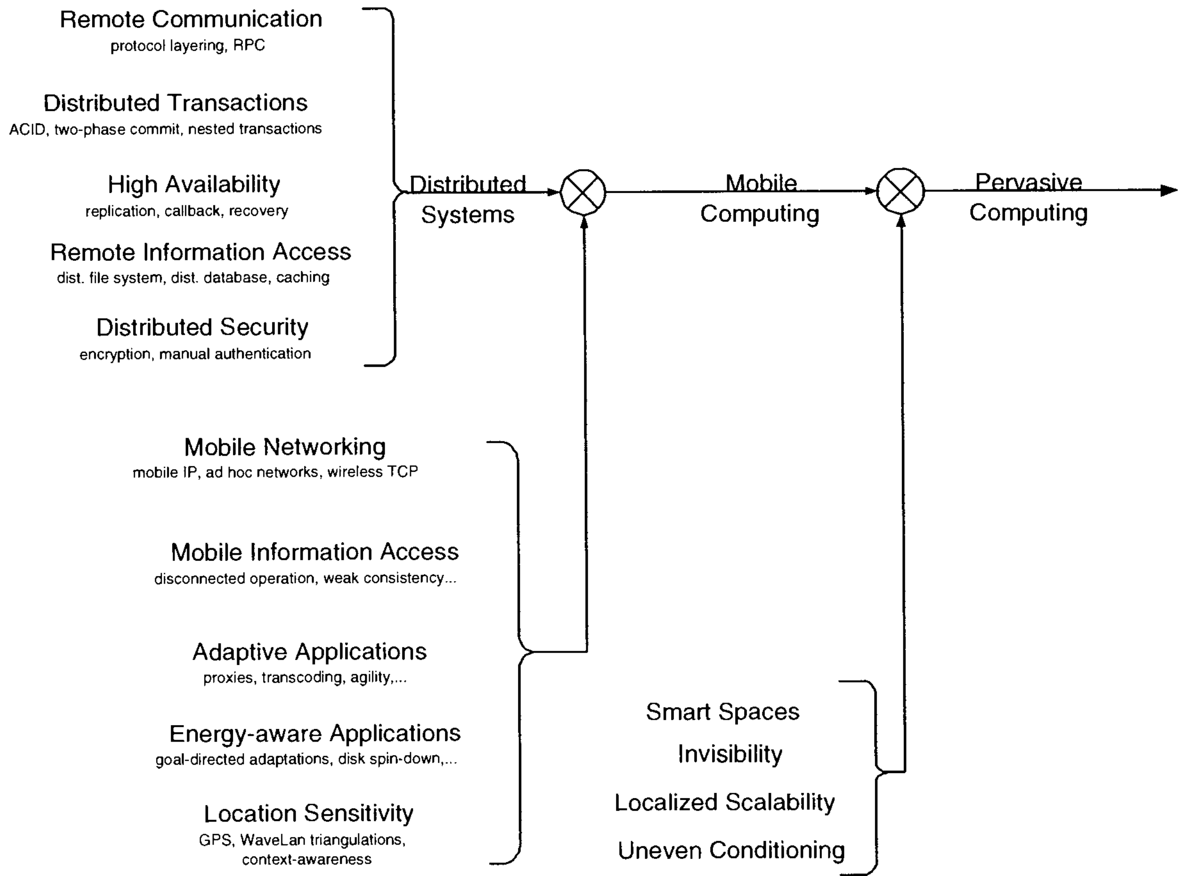

Figure 2.1. *Taxonomy of UC research*

[2]

Computers have become increasingly mobile due to smaller size and power requirements. Now computers can be found in many homes, offices, vehicles and various other environments. This is the beginning of UC. Figure 2.1 depicts the progression of technology that has allowed UC to exist. It shows the movement of technology from centralized computation to distributed computing to ubiquitous computing.

One of the first major pushes toward creating ubiquitous computing environments was during the mid-90s at Xerox PARC. At the forefront of UC was Mark Weiser who referred to ubiquitous computing as "The third wave of computing is that of ubiquitous computing"[3]. Mark Weiser, and many other members of the Xerox PARC research community, created a system with which users could interact and collaborate, irrespective of location, using various devices. The devices ranged from large wall-based screens (LiveBoard), desktops (PARCPAD) to handhelds (PARCTAB). Users could share screens and interact with each others via shared displays and voice.

Calm Technology[3], was another concept from Xerox PARC that claimed users could better interact with their computer environments by allowing information to be incorporated into the user consciousness by placing indicators into the periphery. Via the periphery, users could view the state of the computer world at their convenience without continually having to switching their focus from their main effort(s).

The *Active Badge* was a technology developed by Olivetti where user would wear a badge that could be read by the environment's information system[4]. The badge allows the user to be identified as they roamed throughout the workplace. This allows the automated logging into immediately adjacent computers, the routing of telephone calls, and the automated logging out and/or moving of processes and user interfaces to different terminals as a user moves throughout the work environment. Ethical issues such as privacy where dealt with by allowing the user the ability to disable the badge by removing it and placing it face down at any time. Also badges were not monitored in washroom areas.

2.1.2 Benefits of UC

Just a few years ago the conception of UC was unfathomable. This has quickly changed. The small size and low expense of small mass produced devices has promoted demand for small devices. The proliferation of small devices is a key step toward UC.

Once people have access to devices there is the potential to make significant gains by integrating these devices and thus creating a collaborative environment.

2.1.2.1 Instant Information

The rapid adoption of Internet aware devices, such as cellular telephones and personal digital assistants (PDAs), has opened new consumer markets for both hardware and software vendors. For example, users may want to be notified of changes in the stock market. They may only want to know about very specific trends of a stock in relation to another set of stocks. So there is a certain amount of processing that a user may desire on raw data to create insightful information. The user may be a doctor who while on route to work wishes to see how their patients health has changed during their absence. They could have up-to-date information on their PDA. A person driving their vehicle may desire the fastest route home or the closest available parking spot. These scenarios, and many others, could benefit by ubiquitous computing technology.

2.1.2.2 Cost Reduction

Mass production has allowed devices to produced inexpensively. The small size of many devices allows few resources to be used and large gains can be made via economies of scale. Small inexpensive devices can be distributed on a vast scale. This could provide inexpensive, accurate, an dramatically enhanced resolution data. Inexpensive devices may be simply replaced if they become faulty.

Some devices are so small that they are difficult to see with the human eye e.g., RFiD Journal shows Radio Frequency tags (RF tags) that are only 77 microns in size[5], [6]. Devices can be implanted in products distributed throughout an environment or implanted directly into people and/or pets[7], [8].

2.1.2.3 Safety and Security

Embedding devices into products allows tracking of products as well as automated interaction with other devices. For example, a robot on a flexible assembly line can identify the product on which it is working. This allows the robot to adapt its actions based upon this information; thereby allowing a more flexible assembly line.

Embedding devices into people and pets allow for real-time monitoring of location and important medical information. Some important medical facts that can be monitored are pulse, 3-lead ECG, blood pressure, blood chemistry (Oxygen), and core temperature [9].

Advanced clinical testing may be done more accurately, safely and conveniently then ever before. Devices are less awkward or intrusive due to smaller form factors, reduced power requirements, and more flexible ways of providing power to the systems. For example, Applied Digital Solutions has created *Thermo Life*, a device that captures energy from the mammalian body. It is implanted directly into the body and converts heat to electrical energy [10]. Front Edge Technologies has produced *NanoEnergy*, which are flexible solid state batteries that are only 0.1mm thick yet yield .5mA [11]. Such power sources allow computers to exist inside and outside the body. These technologies, as well as solar and other ambient energy sources permit computing resources to exist any where on the face of the earth and more.

Devices are not only faster and more accurate than humans, but also work in hostile conditions that humans cannot safely handle. UC systems can tirelessly monitor processes and places adding another level of protection for those who work in dangerous environments.

2.1.2.4 Aggregation of Devices

Accessing a device often can provide all the information the user desires. An example of this would be the temperature outside or in Puerto Vallarta. In other instances, the data provided by a single device does not provide the information the user desires. By allowing the system to view many devices that are spread throughout the environment, a dramatically different view of the information is sometime available. For example, hundreds of thousands of the populations personal weather stations could be unified to provide a very detailed image of the weather, potentially leading to new insights.

2.1.2.5 Scalability and Stability

Decentralization is often a mechanism used to provide both scalability and stability. If a load on a system is too great, part of the load may be moved to another part of

the system. If the process is important, it may be important to add redundancy and distribute processing across many systems and architectures to ensure the process is completed correctly and in a timely fashion. Small devices tend not to have large resources, but small tasks may be effectively distributed to many devices thereby providing both performance and redundancy.

2.1.3 Problems with Ubiquitous Computing

With the creation of new technologies there are always problems. Some are great and some are small. The size of the problem seems linked to the technology's impact on society. "The important waves of technological change are those that fundamentally alter the place of technology in our lives. What matters is not technology itself, but its relationship to us" [3] and our environment. If this indicator has any merit then UC is poised to have many problems. There are many issues to be dealt with and, given that we cannot control access to the technology, we will be forced to deal with these issues.

2.1.3.1 Privacy and Security

Privacy and security are often the first issues put forth in refusing to adopt ubiquitous technologies. We often hear of security issues in the technology that we currently use. Security issues have become so prevalent and important that many controlling and information-based organizations have been created to track these issues, e.g. Computer Emergency Response Team (CERT). For ubiquitous technology to function well, it must have information about the user and/or their location. If that information were to be captured by those with malicious intent, serious damage could quickly ensue.

2.1.3.2 Social Changes

What about the people who cannot afford to buy ubiquitous technology? Will they live at a lower standard of living? Will they have the free time to be equally educated if they must spend their time doing menial tasks. There are many issues that must be dealt with in this realm, but many are issues for public policy not for technology.

2.1.3.3 Safety

The consumer may not realize that they are using the new technology in a safety critical scenario. Who is responsible if some part of the technology failed in its function? How do we prove the technology is robust enough for safety critical tasks?

2.2 Components

The idea of component software has its roots in the great success that component-based manufacturing has had in the hardware sector. Component-based software systems assemble a number of pre-existing pieces of software called software components (plus additional custom-made program code). Software components should be (re)usable in many different application contexts. Particularly, these components should be usable in unpredicted applications and/or by third parties.

The term Commercial-Off-the-Shelf (COTS) component was coined in the mid 90's as a concept for a binary piece of commercial software with a well-defined application programmer's interface and documentation. The component market has gained momentum from its inception with technologies such as COM and CORBA technologies to Sun Microsystems' Enterprise JavaBeans and COM+.

Using the component-paradigm for software construction has various benefits: it increases the degree of abstraction during programming, provides proven solutions for certain aspects of the application domain, increases productivity, and facilitates maintenance and evolution of software systems.

Szyperski views a "component as a unit of composition with contractually specified interfaces and explicit context dependencies only. Components can be deployed independently and are subject to composition by third parties" [12]. The Information Technology for European Advancement (ITEA) group refines Szyperski's definition. Their view is, a component is a logically highly cohesive, lowly coupled, documented software module that can be used as a unit of development, reuse, composition and adaptation. It therefore is an exchangeable architectural element of a software system that acts as a part with a larger whole [13, p. 5].

microCommander's embedded software is designed in a component oriented fashion. microSynergy adopted some aspects of the microCommander component model,

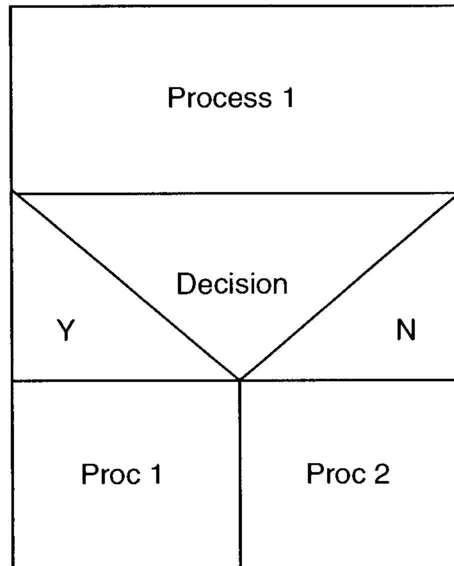

Figure 2.2. *A Simple Nassi-Schneiderman diagram*

as it is very simple and thin[1]. For devices to interact with microSynergy simply have to create a component interface. All software that currently works with microSynergy technology is component-based; it therefore, leverages the benefits of the component-paradigm. Other message-based component models could be supported in the future as new technologies and demands allow.

2.3 Visual Languages

"Pictures are often a more powerful means of communication than words because they can convey much meaning in a concise unit of expression."[14] One reason for their increased expressiveness is that pictures have a two-dimensional nature in contrast to sequential program text, which uses only one dimension. Visual formalisms have been used extensively as design aids and to visualize algorithms including their control flow or data flow.

A classic example of such formalisms is Nassi-Schneiderman diagrams, as portrayed above in figure 2.2. Flow charts of this kind can easily be mapped to equivalent

[1] microCommander's network-based component model is similar to JavaBeans

constructs in textual programming languages. More recently, similar concepts have been adopted in the Unified Modelling Language (UML) in the form of Activity Diagrams and State Charts. Several software engineering tool vendors offer development environments that can automatically map a text-based language to a visual based language and back again. This can make visual programming languages ideal for low or medium complexity applications. Today, visual programming languages are often used in combination with textual languages. Moreover, visual languages and software visualization paradigms are increasingly used to increase human understanding of existing program code in legacy systems. One sub-domain of visual languages is connection-based programming.

2.3.1 Connection-Based Programming

Rather than being an independent paradigm in its own right, the introduction of connection-based programming has been driven by the introduction of the previously discussed paradigms, namely component software and visual languages. Traditional software programs have followed the procedure-call paradigm, where the procedure is the central abstraction. A client calls a specific procedure to accomplish a specific task. This method of programming requires that the client have intimate knowledge about the procedures (services) provided by the server. In connection-based programming, the components and connections are the central abstractions. Components communicate with one another via defined connections. Connections can have logic and/or rules to facilitate their use, i.e. only disallow connections between elements that should not be connected. Connections also quickly depict the presence of a relationship among the interconnected elements. This is a relatively intuitive visualization that eases the understanding of the interconnectedness of the components.

2.3.2 Specification and Description Language

Specification and Description Language (SDL) has elements of a connection-based language. It attempts to satisfy the goals of intuitive program understanding and non-ambiguous program semantics. "SDL is a standard language for specifying and describing systems" [15]. SDL is a high-level object oriented language designed by the

International Telecommunications Union (I.T.U.) for use in communication systems. It was initially developed in 1976 with a text-based paradigm, but was augmented by providing a visual representation [15] and object oriented primatives. The symbolic representation of SDL expresses a modified finite state machine [2](fsm), i.e. a fsm with variables. The automaton's primary role is to determine movement of messages from one part of the system to another while being able to maintain a memory of interactions. Therefore the interaction of parts of the system, also known as subsystems, can be coordinated.

Some of the properties of SDL, which make it attractive for microSynergy, are:

- SDL has a graphical representation making it easy to use and learn
- Definitions and relations are simple, unambiguous and clear
- There is the ability to check the validity and consistency of statements (before runtime)
- SDL is platform independent
- SDL's connection oriented conceptual model seemed simple and intuitive for the connecting of components

SDL has been developed for the specification of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete signals. Being developed by the ITU, SDL was initially intended to serve as a specification language for telecommunication systems; however it has been increasingly adopted for other application areas, in particular, in the domain of engineering embedded systems.[16] A major advantage of SDL over alternative specification languages, such as Unified Modelling Language (UML), is its formally defined and standardized syntax and semantics. Moreover, SDL has a graphical notation as well as an equivalent textual representation. This feature facilitates the exchange of SDL data among different tools. Since its introduction in 1976, SDL was updated and extended approximately every four years until 1992 after which it has remained relatively stable. Its newest revision (SDL 2000) has been extended by additional support for object-oriented modelling. UML class models have now been formally defined as an integral part of SDL, extending its functionality in this area. A thorough

[2] Also known as a finite state automaton, (fsa)

introduction to SDL is beyond of the scope of this paper, more can be found in SDL: Formal Object-oriented Language for Communicating Systems [15].

Chapter 3

Requirements of microSynergy

The main goal of the project is to facilitate the collaboration of controllers, enable the reuse of logic, and allow network evolution in hobbyist and non-safety critical environments.

Some of the initial requirements that came to be as a direct result of the industrial collaboration are the following:

- Minimal interference with current microCommander development, company reputation and user interaction
- Changes to microCommander should be very small or non-existent
- microSynergy must be easy to use and integrate with the microCommander system
- Generated code be portable to new platforms

3.1 Collaboration of Controllers

The microCommander system is comprised of programs that exist on a host PC and on controllers. The microCommander architecture is discussed in chapter 4. microCommander's PC based software can only interact with one controller at a time. The goal of collaborating controllers necessitates that more than one controller act independently of microCommander's PC-base program and/or user interaction. To attain this goal we envisaged a graphical programming environment that allowed the definition of collaborative logic and a controller-based runtime engine that interpreted that logic.

3.2 Reuse of Logic

A robust mechanism for reuse of logic was required. This necessitated that logic be abstracted from the software and hardware components which it controls. The logic could then be easily reused with other software and hardware components. The ability to program with interface descriptions and later bind with the appropriate components facilitates reuse.

3.3 Evolution

Networks, and the demands upon them, tend to change through time. microSynergy had to support this change. The recognition of changes of this prompted the desire to provide mechanisms to assist in this reality. Some of the mechanisms that microSynergy must support are the following:

- Multiple controllers must be able to consolidated into one controller.
- Each controller in the system should be easily substituted by one or more controllers.
- Functionality must be able to be moved to different parts of the network.
- The interrelationships in the system must be easily changed.

3.4 Changes to microCommander

microSynergy required an abstract interface for interaction with microCommander components. The interface provided natively by microCommander components required predefined knowledge of all the components. As microSynergy was to dynamically bind with components the microSynergy system would not have the required information so a simple abstract interface was required. Intec agreed to providing in-gate and out-gate components. These component would be used to augment microCommander components so that they could integrated into a microSynergy network.

3.5 Ease of Use

Ease of use implied that the system was:

1. Simple to understand
2. Inviting to use
3. Mistakes were simple to fix
4. The user required as little interaction as possible

As we have seen many home users have gravitated toward using GUI based applications. To have an application that was console based seemed like a step backward. It would be unlikely that home users would accept it. A GUI based tool therefore became a requirement for microSynergy.

Facilitating the collaboration of controllers requires that users define coordination logic. A text based language has too many degrees of freedom. Some problems with text-based languages are typing mistakes, infinite loops, and unintelligible code. Our mechanism was to decrease or remove these issues from the user. We wanted a program that was non-ambiguous and intuitive for the home user. A visual language for the programmer seemed like a reasonable course of action.

3.6 Language Choice

Which language was used on the PC was viewed as largely inconsequential, but the language used on the controller was viewed as very important. The language used on the controller had to allow for the static allocation of memory, efficient memory management, efficient execution and small size. C was viewed as the optimal choice.

Chapter 4

System Design

microSynergy leverages microCommander concepts and technology. An understanding of microCommander is crucial to the low level understanding of microSynergy. This chapter will discuss some key aspects of microCommander as well as microSynergy's software and hardware components, communication protocols, message structures, and file formats.

microSynergy's role is to coordinate controllers not to create the programs that perform the embedded functionality. microCommander is a piece of software that allows users to program controllers with pre-built *embedded components*, i.e. components on the controller. These components are configured on a PC, the configurations are downloaded to a controller and the state of the embedded components is visualized in real-time through the Internet.

To gain a better understanding of the structure of the microSynergy system and how it integrates into microCommander, it is important to understand some aspects of microCommander's architecture.

4.1 microCommander

microCommander is a component based tool/framework for visualizing, interacting with and configuring microCommander embedded components, i.e. components that exist on a controller. microCommander allows the real-time visualization of the state of embedded components through the Internet. microCommander is composed of many software components distributed throughout various hardware components. Figure 4.1 depicts the distribution of both hardware and software components.

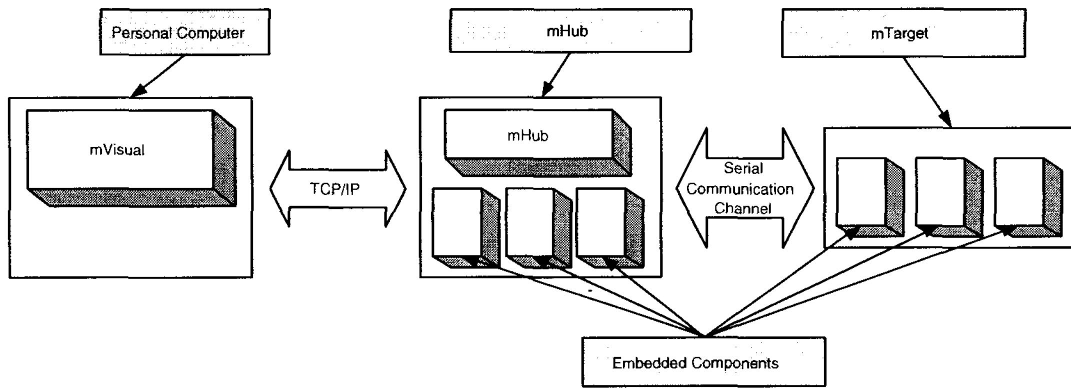

Figure 4.1. *A high-level overview of the microCommander architecture*

4.1.1 Software Architecture

The software architecture of microCommander has three main components: mVisual, mTarget, and mHub. *mVisual* is the Microsoft Windows PC based program that configures embedded components and visualizes the state of embedded components. Further discussion about mVisual can be found in section 4.1.4. *mTarget* is the collection of embedded components that exist on the controller. Greater detail is provided in section 4.1.5. *mHub* is the communication protocol bridge that brokers messages between mVisual and a microCommander controller. mVisual communicates to mHub via TCP/IP, and microCommander controllers communicate via serial. mHub's role is to manage the translation between TCP/IP and serial communication. More details of mHub can be found in section 4.1.6.

4.1.2 Hardware Architecture

The microCommander hardware architecture has three main parts: the controllers that host mTargets, the controller that hosts mHub, and the Windows based PC that hosts mVisual. Figure 4.2 depicts microCommander hardware and the relationships between the hardware components.

The controllers with which mVisual wishes to communicate tend to have limited memory, processing power, and communication infrastructure. Many do not have the resources required for TCP/IP, so they can not directly communicate to the Internet.

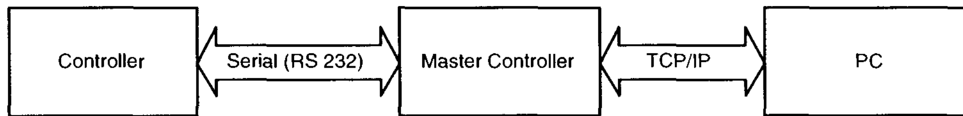

Figure 4.2. *The microCommander Hardware Architecture*

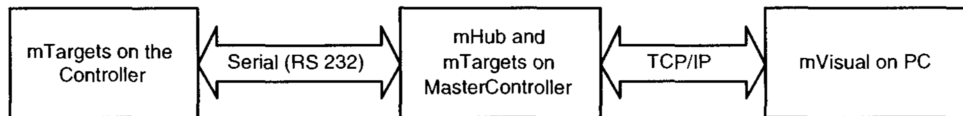

Figure 4.3. *The microCommander Software Architecture*

They must communicate through an intermediate controller that does have TCP/IP infrastructure.

The currently supported controllers are based upon the Motorola 68HC16, and the Intel 80386 and compatible CPUs. The hardware requirements of the PC are quite minimal. It currently must be running Microsoft Windows (win95 or higher) and have TCP/IP support. It must be connected to the Internet.

4.1.3 Hardware Resource Distribution

The controller that hosts mHub will be referred to as the *master controller*; the other controllers will be referred to as the *child controllers.* Simply *master* and *child* will be used where knowledge of the context allows.

The master communicates to mVisual via TCP/IP. Most child controllers do not support TCP/IP. They communicate with the master via RS232 protocol. The master controller needs to support TCP/IP and serial protocols.

Communication is completely coordinated by the master controller and thus child controllers can not independently initiate communication. That is, child controllers are passive; they are polled by the master controller in what is often called a ping pong protocol, as depicted in figure 4.4. The *ping pong protocol* implies that the child controllers only respond to requests,and therefore can not notify the network when they are plugged in or send event notification until requested.

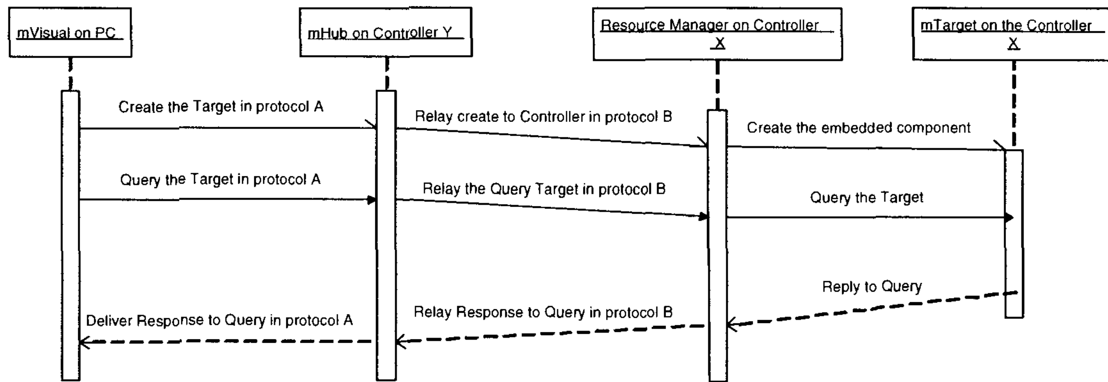

Figure 4.4. *Message Sequence Diagram portraying the ping-pong protocol*

4.1.4 mVisual

mVisual is one of the more complicated parts of microCommander as it contains a great deal of the functionality. Its role is to visualize the state of embedded components and to configure embedded components. The following simple scenario will provide an illustration of its functionality.

When mVisual is started, a controller must be chosen. When a specific controller is connected, mVisual lists all of its currently configured components and previously selected GUI components connected to the embedded components. The user can select embedded components from a toolbar, shown in figure 4.5. Upon selection of an embedded component, the user is prompted with a configuration dialog, shown in figure 4.7. The dialog allows naming of components, and facilitates defining a component's input and output. By setting a component's inputs to be a pin, i.e. a wire on the controller, the component can take input from the physical world. By making the output of a component a pin, the component can output information to the physical world. Components can be *pipelined*, i.e. one component's output can be used as input to another component.

Visuals may be attached to the embedded components to view their state. Visual components can be dragged onto a canvas from a toolbar; shown in figure 4.6. Simple visuals such as LEDs and stripchart readers allow users to view current state as well as log the changes in the state of a component. Visuals such as potentiometers, text boxes, sliders, buttons, etc. allow users to directly manipulate components via their

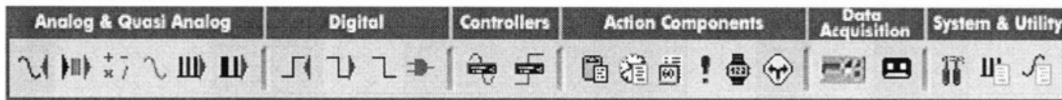

Figure 4.5. *The embedded component panel*

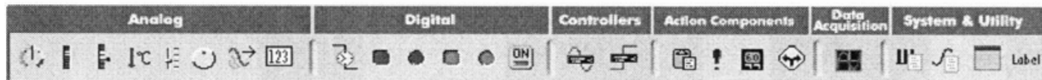

Figure 4.6. *Visuals that can be attached to microCommander Components*

their desktop with standard input devices such as the keyboard and mouse.

4.1.5 mTarget

mTarget is the name of the software on the controller. It consists of embedded components, (EC) and logic that controls communication and resource usage. Figure 4.8 depicts the mTarget's architecture on a controller. As stated in section 4.1.3, some mTargets exist on controllers that passively communicate, but this does not imply that they are passive in other aspects of their functionality. They continue to function even when mVisual is not running or not connected. That is, as long as the components do not require input via mVisual they will preform their tasks.

mTargets have unique identification number on the controller. A gate's identification number is abstracted from the programmer by the use of a name. On microCommander devices there is a supervisor component that marshals messages from networking subsystem to the embedded component in an asynchronous fashion [1]. Other devices that are microSynergy enabled must provide their own mechanism for handling translation of messages to an action, such as a function call, and from an action to a message.

mTargets are designed on the premise that they are used in a net-centric environment. Therefore the role of each specific mTarget is not related to any other

[1] asynchronous implies the supervisor does not wait for a message to be returned, it simply continues with its next task.

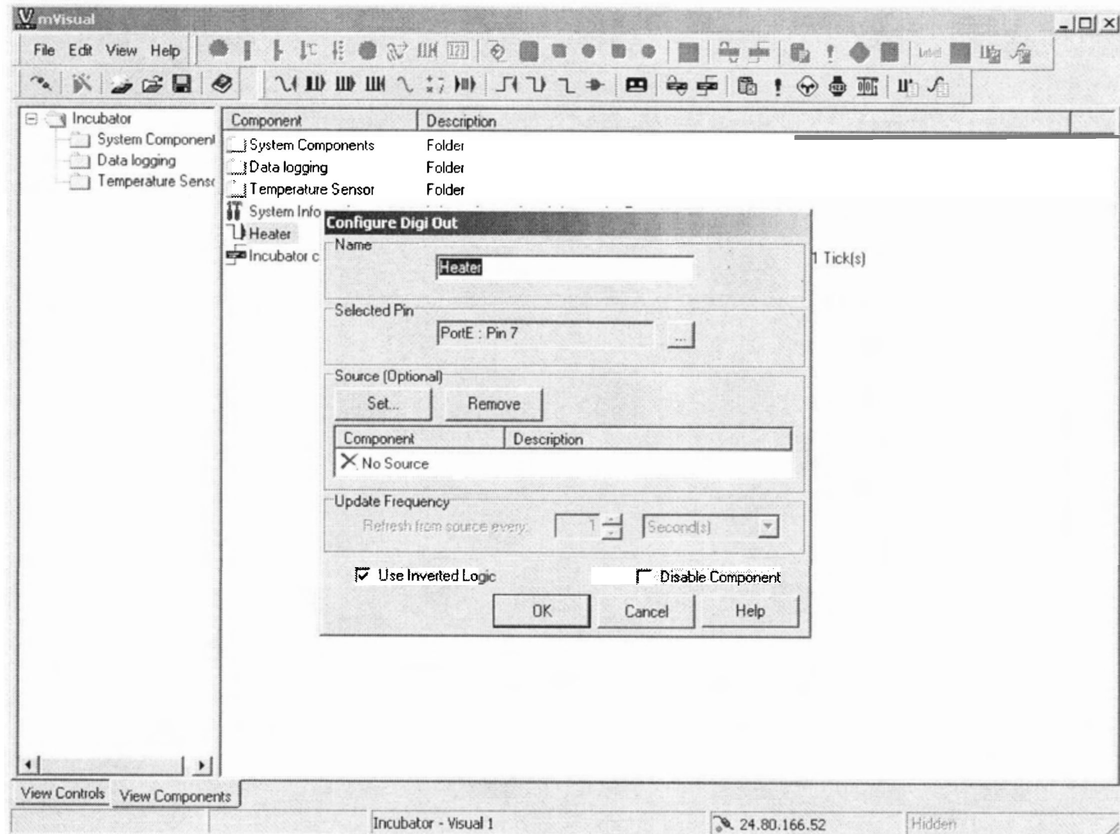

Figure 4.7. *A typical embedded component configuration dialog*

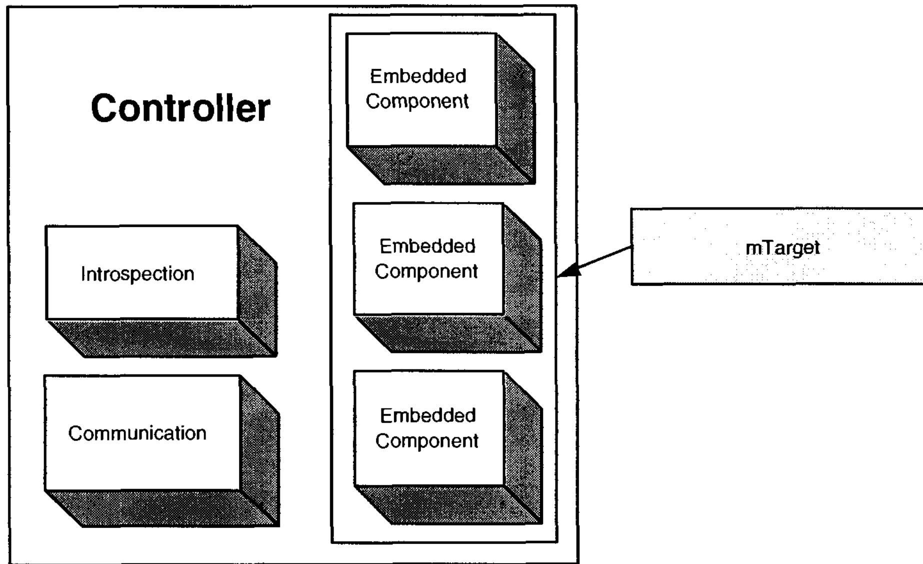

Figure 4.8. *A high-level view of a microCommander controller*

mTarget. This results in very low coupling between components and therefore are easier to reuse.

The mTargets and embedded components have well defined interfaces that allow easy access to the mTarget's resources as well as the ability to dynamically create, configure, and run an embedded component. mTargets all support introspection of their interfaces. This allows mVisual to query the controller to determine what components exist on the controller and what functions the component offers.

4.1.6 mHub

mHub's primary role is bridging the different communication protocols. It receives the TCP/IP packets sent by mVisual via the Internet and converts them to serial so that they may be received by the mTargets [2]. The mHub can co-exist, on the same controller, with an mTarget allowing it to fulfill both functions at the same time. This makes a more cost effective controller network.

mHub contains a scheduler that schedules the communication between the mHub

[2] mTargets all listen on the same RS232 connection for messages with their target identification number. All other messages are ignored.

and each mTarget on a round-robin basis. This requirement helps to ensure that communication is not corrupted and limits concurrency, thereby simplifying interaction.

4.2 microSynergy

microSynergy consists of two separate applications: microSynergy Editor, referred to hereafter as *Editor*, and microSynergy Runtime, referred to hereafter as *Runtime*. The Editor resides on the developer's workstation, typically the same PC as mVisual, but not necessarily. It provides a graphical user interface for the developer to visualize the mTargets attached to the mHub, and visually specify the communication logic. microSynergy Editor enables the user to download new logic, known as the *roadmap*[3], to Runtime. Additional microSynergy Editor details are provided in 4.2.1. Runtime resides as a service on the master controller. Its interpretation subsystem is responsible for interpreting the programs created by the Editor. Its communication subsystem add significant multi-protocol communication support and its introspection subsystem handles the (de)registration of devices. Runtime details are provided in section 4.2.2.

As previously stated, microSynergy's role is to coordinate devices and thereby oversee the network. microSynergy's programming model encourages the encapsulation of coordination logic in a connector component i.e., a component that connects other components. This programmatic design is analogous to the distribution of logic throughout the system. microSynergy's architecture separates *coordination logic*, the logic in a connector component, from *application logic*, the logic defined in the embedded components. This division keeps the coordination logic very simple and pushes the responsibility of reacting properly to incoming messages onto the embedded component.

4.2.1 The Editor

microSynergy Editor is the user interface that supports the programmer in creating, integrating, and understanding the micro controller network. It allows the programmer to:

[3]Roadmap refers to the fact that the logic guides interaction as a map would guide an individual.

1. visualize the controllers attached to mHub
2. specify the coordination logic
3. distribute the logic to the network
4. import and export and reuse pre-built connector logic

microSynergy Editor is implemented in Java, and consists of 4 packages: Editor, Model, Parsing and Communication.

Editor The Editor is the graphical portion of microSynergy editing environment. It provides facilities to the developer such as:

- the ability to introspect and visualize the network
- the ability to create, edit and reuse connector logic, and associate the connector logic with specific embedded components
- the ability to load and save connector logic

Model The model is the abstract underlying representation of the network and all its components. It provides a system of typed nodes and arcs that specify how nodes can connect and validates communication logic.

Parsing The parsing subsystem is responsible for converting the editor's internal representation of the system to a Connector Description Language (CDL) file and back again. The parsing subsystem also converts the CDL file to a Connector Execution Language (CEL) file that is downloaded to the microSynergy runtime system. The parser has access to the model and traverses the model building the CDL file as it goes. When the model is generated from a CDL file, it is loaded into a Xerces XML parser, the resulting Document Object Model (DOM) is traversed updating the internal model as it goes.

The creation of a CEL file from a CDL file is done by outputting strings while traversing the DOM object created by the parsed CDL file.

Communication The communication subsystem is responsible for handling the communication between microSynergy Editor and microSynergy runtime. It allows the editor to introspect the network and download roadmaps via TCP/IP sockets. All introspection messages received by the master controller are forwarded in the appropriate protocol to each controller on the system.

4.2.2 Runtime

microSynergy Runtime is a dynamic communication routing engine over a logical network embedded components. It allows controllers to fallout of the network in a graceful fashion and be reintegrated with relative ease. If an embedded component falls out of the network the messages it was to receive a not received. Runtime will attempt to resend the message multiple times. After a number of failures it will stop sending the message. When an embedded component is reinserted into the network it can, if on a device that supports a collision aware communication protocol, make Runtime aware that it is present. The process that depended upon the embedded component has only to be restarted.

It is implemented in C, and is designed to be efficient, in terms of memory and speed, as reasonably possible. Its role is to coordinate the network by controlling the distribution of messages between each controller in the network. Runtime does this by executing logic contained in its CEL file each time a message is received.

Runtime is designed in a modular fashion. Its three basic parts are communications, interpretation, and introspection.

Communications The communications subsection, *RTComm*, is critical to the functioning of microSynergy. It attempts to insulate the interpreter from the protocols with which it interacts. It also takes messages in a variety of formats and puts them into a format compatible with Runtime's interpreting environment. When messages are sent to devices, it sends messages in the format expected by the device.

This provides the flexibility to support many differing multiple controllers with different communication protocols. The developer does not have to be concerned with which communication protocols the controllers use or any other communication issues. The developer need only know that the communication protocols the controller uses are supported by microSynergy.

Each controller and embedded component on a controller has a specific identity. Runtime abstracts the low-level identity of the controller and its embedded components from the programmer by the use of a name. On microCommander devices there is a supervisor component that marshals messages from networking

subsystem to the embedded component in an asynchronous fashion [4]. Other devices that are microSynergy enabled must provide their own mechanism for handling translation of messages to an action, such as a function call, and from an action to a message.

Interpretation When a new message arrives, the interpretation subsection of Runtime is activated. It checks its input buffers for a new message. Given the current state of the system it expects a variety of messages as defined in the CEL file. The system buffers messages as they arrive and interprets them in order of arrival. All events are atomic in the execution of the program, that is one task is complete before the next begins. There is no concurrency in execution of runtime logic and therefore be no race conditions. This potential reduction in the efficient use of resources was viewed as a reasonable tradeoff for the simplicity that was maintained.

Introspection The subsystem tracks all microSynergy compliant devices on the network. It can be queried at any time to determine what devices are on the network. Devices that actively communicate can register their presence with the introspection service. Therefore, if the device interacts through a communication protocol such as TCP/IP or Bluetooth©, the self disclosure is allowed.

Passively communicating devices can be discovered by polling a predefined address space. This is due to the fact that passively communicating devices are passive for a reason. That is, that they are connected via communication channels that do not support concept of logical connections. Therefore, a device's attempt at self disclosure could seriously impact the stability of the whole communication channel. Given that self disclosure is not the case, microSynergy must query or introspect the network when an update of the network is required. This involves querying all the controllers that can not disclose their presence to determine their in-gates and out-gates.

The information gathered is displayed in the left-hand tree-view of the Editor, as shown in figure 5.1.

---

[4] asynchronous implies the supervisor does not wait for a message to be returned, it simply continues with its next task.

4.2.3 Initialization

There are two issues in the initialization process that are of interest. The CEL programs are not currently stored in EPROM and must, therefore, be downloaded after the master controller has booted. When the master controller is powered up, it loads Runtime into memory. Runtime waits on a designated TCP/IP socket for a message containing a CEL file. Once Runtime receives the CEL file, it statically allocates the memory needed for interpretation and introspection.

As the system may be composed of passive devices an outside influence has to be inserted into the system to start the collaboration[5]. Currently, a message is sent from mVisual. As a web server has recently been integrated into the Runtime communication subsystem, the initialization process may eventually be initiated by a web service call.

4.3 Messages

microSynergy communicates to devices with messages. The message must be of a specific form so that both the microSynergy and the connected devices can appropriately communicate. This section defines the message formats for all messages currently used in the system.

Valid message types Runtime can send/receive are:

1. INTROSPECT SYSTEM
2. INTROSPECT TARGET[6]
3. GET NAME
4. DOWNLOAD ROADMAP
5. EXECUTE ROADMAP
6. MESSAGE TO TARGET
7. MESSAGE FROM TARGET

[5]In the future, the system could be set to automatically start or if the conception of time was added to the system it could be started at a specific time.

[6]Note: as microCommander used the term target to refer to what I have called a controller the message names refer to targets.

INTROSPECT SYSTEM This message is sent to Runtime by the Editor to discover what targets exist in the network. When RTComm, which keeps a list of all the targets, receives this message, it replies to the Editor with a list of each target on the system.
Message Format:
When microSynergy Editor sends the message to RTComm, there is no payload. When RTComm replies, the payload holds the identifiers for the controllers: int targetID1, int targetID 2, ..., int targetID N.

INTROSPECT TARGET This message is sent to Runtime by the Editor to discover what gates a target has. RTComm sends a message to the target, and passes the reply to the Editor.
Message Format:
When the Editor sends the message to RTComm, the payload is the targetID. When RTComm replies, the payload is: int in-gate0, int gateID, int out-gate0, int gateID, int in-gate1, int gateID, int out-gate1, int gateID, ..., int in-gateN, int out-gateN.

GET NAME This message is sent to runtime by the Editor to discover the name of a target or gate. RTComm sends a message to the target, and passes the reply back to the editor.[7]
Message Format:
When editor sends the message to RTComm, the payload is: int type (either GATE or TARGET), int ID.
When RTComm replies, the payload is: char0, char1, char2, ..., charN

DOWNLOAD ROADMAP This message is sent to runtime by microSynergy Editor to give runtime a new roadmap.
Message Format:
The message starts with 6 integers. The 6 integers in the header of the message count as 2 instructions. Each integer has a meaning. They are:

1. The version of the CEL that is being sent. This is not currently used, but could be helpful in maintaining backward compatibility.

[7]This message has been deprecated a replaced with GetTargetName and GetGateName in the newest version

2. The number of Connectors in the CEL.
3. The number of CEL instructions. Each instruction consists of 3 integers.
4. The number of Targets in the CEL.
5. Not currently used.
6. Not currently used.

The message will then follow the pattern described in the CDL DTD found in section 4.4.1. The types of instructions are indicated in the first integer of the instruction.

EXECUTE ROADMAP This message is sent by the Editor to RTComm to begin executing the new roadmap

Message Format: The message will be empty.

MESSAGE TO TARGET This message is sent by RTComm to a target to get it to change the values on a list of in gates.

Message Format:

This message consists of pairs: int i: inGateID, int i+1: newVal for the in-gate

MESSAGE FROM TARGET This message is sent from a target to RTComm

Message Format:

This message consists of pairs: int i: outGateID, int i+1: updated value for the out-gate

4.4 File Formats

The Editor generates *Connector Description Language* (CDL) which is the communication logic specified in XML format. The CDL files are converted to *Connector Execution Language* (CEL) before being downloaded to Runtime. It is a highly condensed format that is small enough to exist on the limited resources of the controllers.

The Editor's model could be converted directly to CEL but is converted to an intermediate language so that 1) the Editor could be used with runtime engines other then microSynergy's Runtime, 2) the model can more easily recreated from an XML file then a binary file and 3) the CDL file is much simpler to parse and can therefore be more easily integrated with model checking programs[8]. The CDL file is used for

[8]microSynergy does not currently have any model checking to determine that the program is

storing the model to disk and recreating the model when the editor loads the file. A file with a correct model would imply that each component in the model properly fulfills it's specific function and that the system as a whole function correctly.

4.4.1 Connector Description Language (CDL)

In using XML as an intermediate form, we have insulated the microSynergy Editor from the generation of the executable code. This allows for easier evolution of the runtime system or Editor, as they are not strongly coupled. This separation of concerns offers two other advantages: validation and portability. The XML form has a document type definition (DTD) associated with it. Therefore, the XML parser can ensure that a program is well formed. This would catch many simple mistakes. Having an intermediate format is to simplify porting to to systems that support different binary forms. A sample CDL file may be viewed in Appendix A

4.4.2 Connector Execution Language (CEL)

The CEL file is a binary file used for encoding a program interpreted by Runtime. It is relatively small and concise. Its general form is that of a two line header, followed by the body of the file. The header and body is composed of many commands. Each command in the CEL language is composed of three integers. Some specifics about commands are:

- The first integer determines the instruction type
- The Thread instruction are always followed by an In Instruction
- Conditional Instruction are always followed by 2 Param Instructions
- Out Instruction are followed by a Param Instruction

Command descriptions are shown in table 4.1.

well-formed only XML schema validation to ensure the document is well-formed.

Instruction Type	Second Int	Third Int
Connector	connector id	number of thread's/connector
Thread	number of Inputs in thread	unused
Input	Controller id	out-gate id
Cond	number of true instructions	number of false instructions
Output	targetID	inGateID
Trans	thread to transition to	unused
Const	value	unused
Var	targetID	out-gate Id

Table 4.1. *Table of CEL Commands*

Chapter 5

Visual Language

5.1 Motivation

Reducing complexity and removing user intimidation are some of the main goals of the microSynergy project. The use of a visual programming language was viewed as a reasonable means of achieving some of these goals. The world hosts a very large number of visual languages, but very few were appropriate for microSynergy. Many languages such as VHDL are large and complex where as many others are domain specific, which makes them small and easy to learn, but not necessarily appropriate for microSynergy's tasks.

5.2 microSynergy's Visual Language

microSynergy's language can model the network of controllers. The interrelations among controllers and the state of the network is reflected within the language.

5.2.1 Connection-Based Programming

Collaboration of devices is fundamental to the project. It was clear that devices had to work together toward a common objective. In reality, there is no global understanding at the component level, there is only response to stimuli. The issue became how to share the stimuli of the sensors on one or more controllers with other, appropriate, controllers. The use of connection-based programming seemed simple, intuitive, and powerful. Therefore, we used a connection-based language in which we could encapsulate logic that coordinates the flow of signals throughout a network

of controllers. microSynergy implements its own language based upon the ITU's Specification Description Language (SDL).

5.2.2 State-Based Modelling

States are used to maintain memory of past events and define what logic is associated with current and future events. Memory of events can be captured in microSynergy by the use of states. When a specific event happens microSynergy can remember the event by transitioning to a different state. The model requires only one state, which implies that it can act in a stateless fashion, i.e. transitions move from one state back to the same state.

In reality, states do not have to reflect the actual state of the network. The developer defines states and what they mean. It is advantageous that the model have some association with network's state for purposes of understandability, but this is not a requirement. The resolution of the model must only be as fine grained as required by the developer, i.e. the model does not have to represent all possible states, only those which are deemed appropriate for the required tasks.

5.2.3 Syntax

The symbols used in microSynergy's VL are simple. The primary reasons for the selection of the symbols is their simple nature and expressivity. They are by no means unique and could easily be replaced by another visual representations, such as UML state charts or petri-nets, if deemed appropriate.

microSynergy's language has a set of core primitives that can be combined to express complex concepts. The symbols express the state of the system, sending and receiving of messages, conditional branching, and the distribution of logic throughout the embedded network.

In an effort to simplify understanding of the valid syntactical statements in microSynergy's visual language the rules that describe valid syntax are expressed in a layered graph grammar form as presented by Rekers and Schurr in 1997 [17]. Layered graph grammars are a natural way to describe VL as they provide formalisms to express characteristics of a VL. They allow the expression of grammars more generally

then context free grammars. Symbols within the microSynergy VL have the notion of context. The Editor controls the how the symbols interrelate to one another based upon their context.

The semantics of each symbol are well described in Ellsberger's book *SDL: Formal Object-oriented Language for Communicating System* [15]; I will leave it to the interested reader to investigate.

5.3 Language Core Semantics

5.3.1 Connector Components

Connector components are the encapsulation mechanism for communication logic. It is important to recognize the connectors have nothing to do with the logic that exists in embedded components only their communication. A connector's role is to link embedded components to one another. The relationship among components can be one to one, one to many or many to many. This allows an embedded component to communicate with any number of embedded components.

Connector components differ slightly from Ellsberger's SDL definition of a block or process [15]. A connector component, also simply known as a *connector*, essentially merges the semantics of a block and a process. A connector can be connected to by channels, just as a block, but it has a description of logic within it, just as a process. The primary motive for this divergence, from the SDL standard, was simplicity for the programmer. SDL has systems that hold blocks that hold processes that hold logic. Connectors simply hold logic and connect opaque blocks, i.e. controllers. Connectors exist only within the system level, as depicted in section 5.4.1. There can be many connectors within a system, but there is only one system, the default workspace. This restriction reduces the ability to express very complex models and encourages simple interrelationships between network components.

The layer of abstraction between the logic that controls communication and the entities that are coordinated by the logic. Connectors allow for grouping of logic into coherent bundles, that eases reuse.

5.3.2 Embedded Components

An embedded component is a software component that exists on a controller. It encapsulates a specific piece of functionality that exist on the controller, e.g. a pulse width modulator function. Embedded components can interact with one another to perform complex tasks. They can also interact with the exterior world through the sensors or pins by which they are attached. The sensors and pins allow the functions to receive inputs or produce output to the physical environment. The concept of an embedded component is critical reusing embedded code in microSynergy.

5.3.3 Threads

Threads are programmatic representations of a series of instructions. A connector is always in a specific state and can be viewed as a series of conditional state transitions. A thread represents the instructions completed in the transition from one state to another or from one state to itself. The current state defines the input types analyzed upon receipt of a message. If the message type is not listed then it is discarded.

5.3.4 Gates

A controller has in-gates and out-gates that define a controller's publicly accessible functions or interfaces. A connector has inputs and outputs, which may receive or send signals to controllers. The source and destination types of the gates and inputs/outputs define the labels. Communication between in-gates and out-gates are visualized by lines between connectors and controllers[1].

In-gates are the incoming interfaces to embedded components. They are used to receive messages from the network. When the connector generates an output, it is outputting to an in-gate.

microCommander connects in-gates to embedded components. The programmer must add an in-gate to an embedded component to receive a message from the network. When a device is not a microCommander-based device, it must define its own

[1] The channels delay messages in transport for a non-deterministic duration. This results in non-determinism in the order of arrival for messages sent at the same time [15, p. 33].

in-gates to interact with microSynergy. In microSynergy editor, an in-gate's visualization is a blue circle attached to the side of a controller, as shown in 5.3.

Out-gates are the outgoing interfaces associated with controllers. They are used to send out going messages from embedded components to the network.

microCommander connects out-gates to embedded components. The programmer must add an out-gate to an embedded component to send a message to the network. When a device is not a microCommander-based device, it must define its own out-gates to interact with microSynergy. In microSynergy editor, an out-gate's visualization is a red circle attached to the controller symbol, as shown in 5.3.

5.4 Language Core Syntax

There are two levels of core syntax. The *system level* describes the logical interrelationships of controllers. The *connector level* describes the logic that coordinates the system of controllers viewed inside a connector. Within the system level, the allowable symbols/objects are controllers, connectors, and channels. Within the connector level, the allowable symbol/objects are states, inputs, outputs, priority inputs, and conditionals.

5.4.1 System Level

Connections in the system level portray the interrelationships between controllers. This provides a high level perspective of controller collaboration. Figure 5.1 shows a valid system level program. Target2 is connected to mSynergyHub via Connector1. Figure 5.2 describes valid syntax for the system level.

Figure 5.2 states that a system can have many independent groupings of controllers and connectors. The syntax diagram views these as subsystems. A connector can be connected to many controllers. Many connectors can be connected to the same controller.

5.4.1.1 Controllers

Controllers represent the micro controllers in the system. The visualization is a yellow-labelled rectangular symbol. Shown in figure 5.1. The controllers have specified

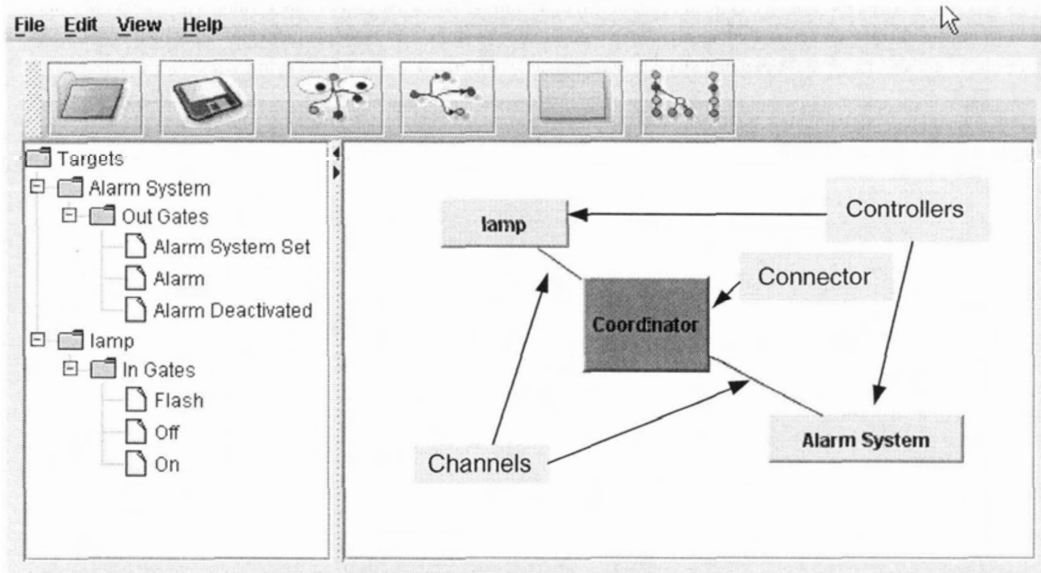

Figure 5.1. *An annotated sample of a system level program*

in-gates and out-gates that are the access points to the embedded components.

The in-gates and out-gates act as input and output interfaces to a controller. The in-gates and out-gates are attached to specific embedded components that exist on the controller. Controllers are attached to one another via connector components.

5.4.1.2 Connectors

Connectors encapsulate pieces of logic that connect one or more embedded components. Each connector defines a state machine that coordinates the communication between embedded components on one or more connected controllers. Connectors are represented as grey labelled rectangles in figure 5.1. Connectors contain modified finite state machines (fsm) that determine message routing within the system. Connectors connect any number of controllers to one another. Controllers can only communicate when connected via a connector. Connectors created by linking two controllers, dragging from one controller to another, have a default fsm that passes all messages from one controller to the other. Connectors created by clicking the new connector button or menu option are empty by default.

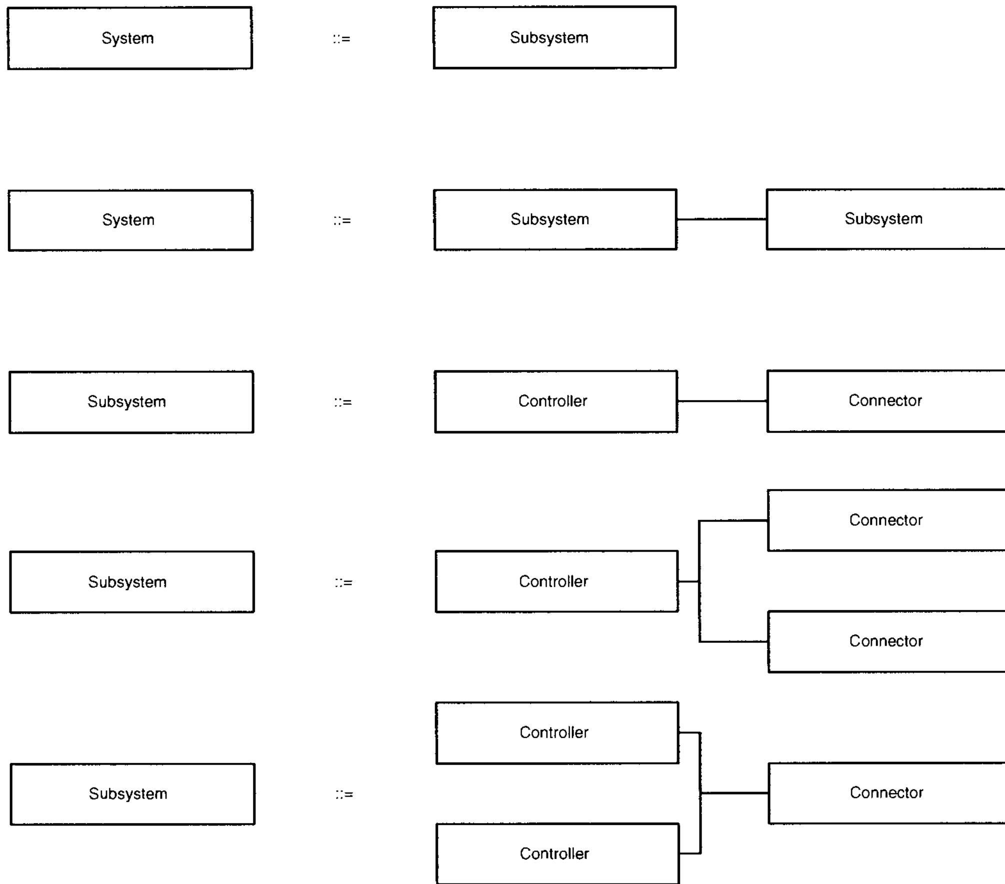

Figure 5.2. *System Level Core Syntax*

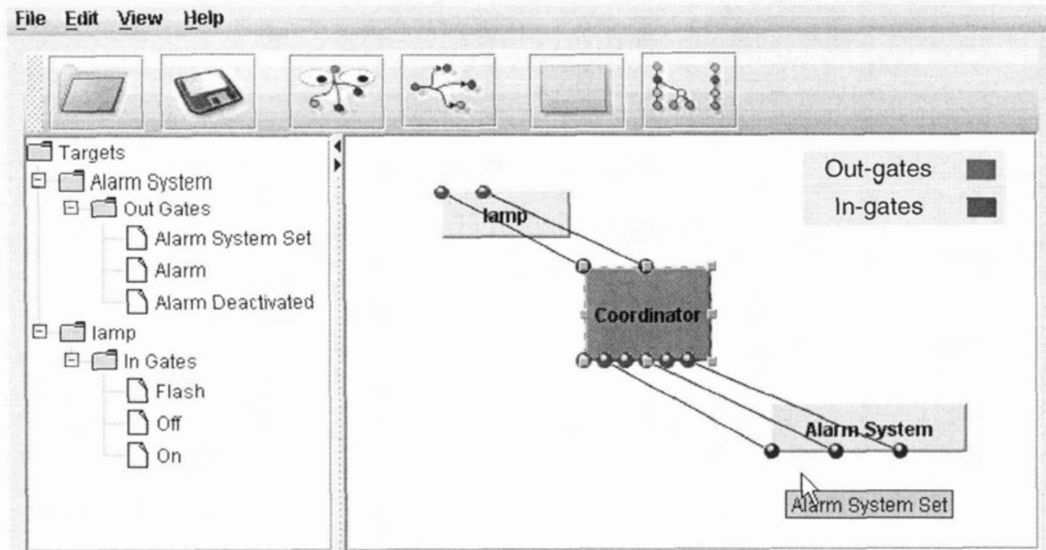

Figure 5.3. *A system level view with visible gates and connections shown*

5.4.1.3 Channels

Channels are pathways that connect controllers and connectors; they appear as lines, in figure 5.1. If the line is labelled, the label indicates the source and destination interfaces that the channel connects. A channel connects access points on controllers to connectors.

5.4.2 Connector Level

The connector level describes the fsm that coordinates controllers. Each symbol defines an instruction. The symbols within the connector level may connect with one another in very specific ways to define transitions from one state to the next. Most connector level elements have attachment points on their top and bottom. When a connection from one element is being made, elements that are acceptable for connecting have their appropriate connecting points highlighted. Figure 5.4 is a simple connector level program.

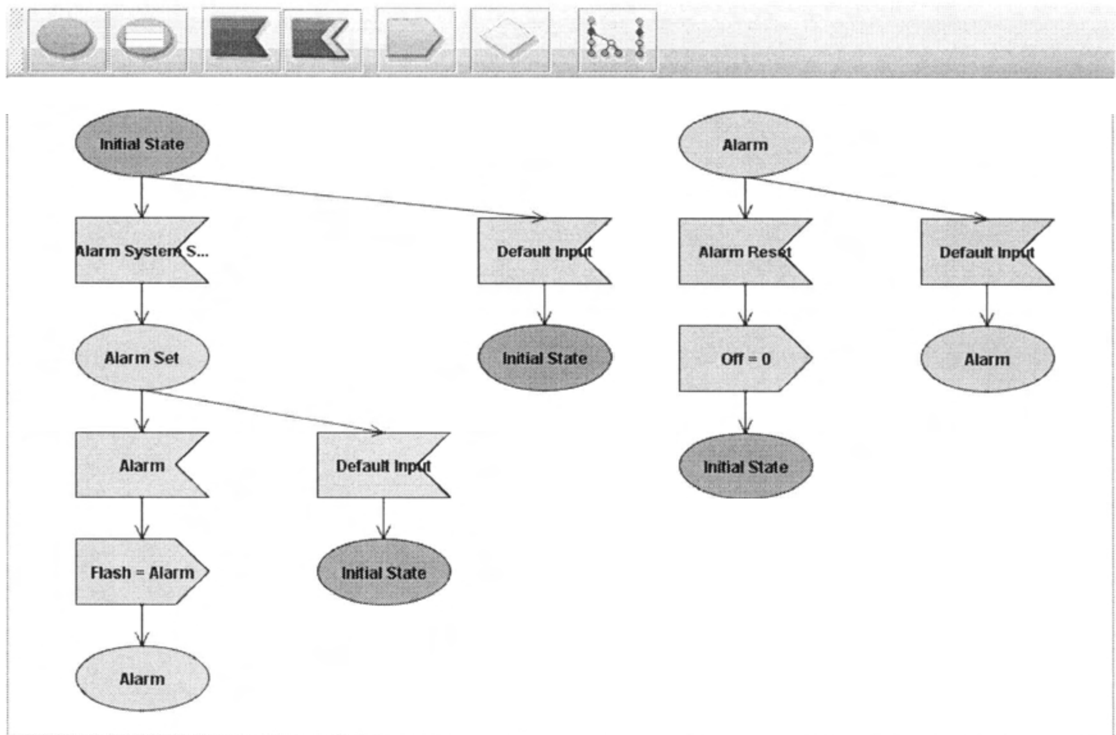

Figure 5.4. *Inside a connector*

	State	Input	Conditional	Output
State		X		
Input	X		X	X
Conditional	X		X	X
Output	X		X	X

Table 5.1. *Allowable connections matrix*

Figure 5.5 describes vertical groupings of connector elements called threads. Threads are to be read from top to bottom. All threads start with one state and end with one or more states. The second element is always an input symbol as is visible in 5.4. The subsequent elements are either conditionals or outputs.

The following symbols can only exist within the connector.

5.4.2.1 Threads

All elements within a connector must be ordered and grouped in a specific fashion, as shown in figure 5.4. Each grouping of symbols is referred to as a thread. Threads are sequences of instructions done in transition from one state to the next. The symbols are ordered by connecting symbols with arcs. Each arc connects only two symbols. The direction of an arc is analogous with the direction in which instructions are executed, i.e. instructions are executed sequentially.

Within the Editor, when a connector component is attached to a target, the underlying options for the inputs and outputs are adjusted so that the input options for the connectors agree with the specific targets attached to the connectors, similarly for the outputs.

There can be multiple threads within one connector, each starting with a unique state/input combination. They are always presented in a top down fashion and always begin and end with a state symbol. The layouts as well as the arcs assist the eye in visualizing the ordering of sequences of instructions.

Table 5.1 depicts the allowable connections of the symbols that form threads. The element in the left most column can be connected to the elements marked by an "X".

Figure 5.5. *Connector Level syntax*

5.4.2.2 States

States define the memory of a connector. The name of the state appears within the symbol, as shown in figure 5.4. A state can be given a name. The only name that it cannot be given is start as that name is reserved for the starting state of the connector, which is shown in green. Since all threads begin and end with a state symbol, the connector is always in a defined state.

5.4.2.3 Inputs

Inputs are used to receive specified messages types from the network. Their representation is a rectangle with a wedge shape removed from a vertical side, as shown in figure 5.4. The text label on the input symbol shows what type of message is being expected.

5.4.2.4 Priority Inputs

A priority input is similar to an input except that priority messages are handled before other inputs. They appear as rectangle with a wedge removed from one of the vertical sides and the wedge is high-lighted by an extra line, as shown in figure 5.4. Priority inputs are acceptable where ever inputs are used. There can only be one priority input per thread.

5.4.2.5 Outputs

The output symbol is analogous with the send command in standard message passing systems. The text label on the symbol depicts the destination of the message, as shown in figure 5.4. Outputs are associated with output buffers that are allocated in the Runtime. Outputs are depicted as a rectangle with a wedge added to the left or right side. The name of the destination appears on the symbol.

5.4.2.6 Conditionals

A conditional is a mechanism for comparing two elements, its symbol is shown in figure 5.4. They provide branching logic to the fsm. The conditional statement is either true or false. If true, the statement on the path labelled true is executed, otherwise

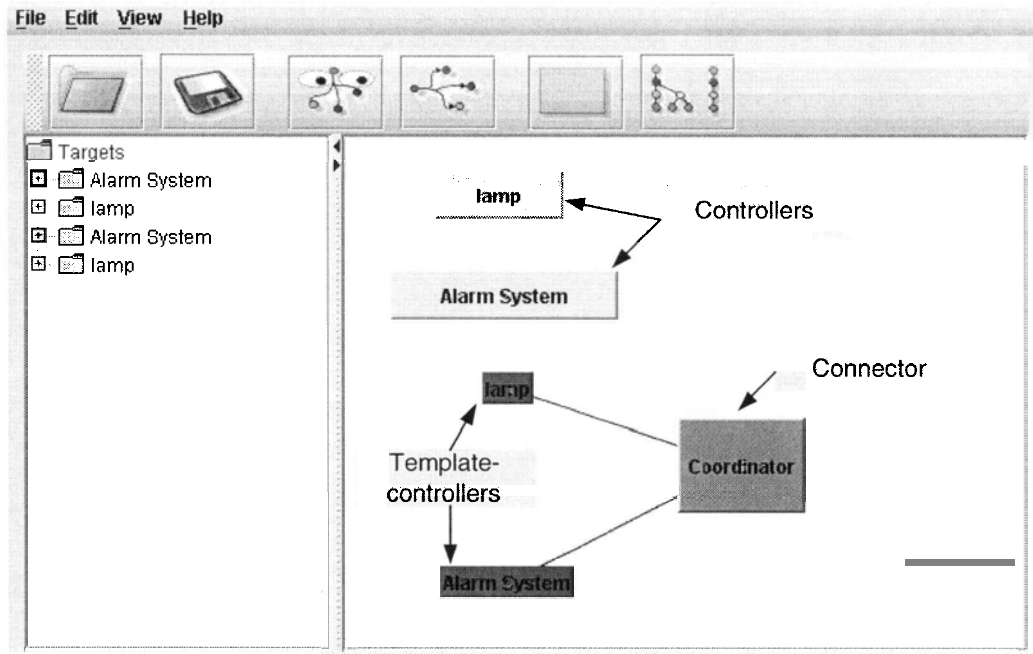

Figure 5.6. *An example of the use of templates with the connections expanded*

the false path is executed. Connecting conditionals to successive conditionals can make complex statements.

5.5 Extended Language Concepts

Extended language concepts refer to building new concepts on top of core level concepts. All functionality can be expressed with core concepts, but the extended concepts provide facilities for high level design.

As extended language concepts do not exist within the SDL specification, their explanation is broken into three sections 1)semantic, 2)syntax, 3) pragmatics.

5.5.1 Semantics

5.5.1.1 Templates

A template is a facility to ease the reuse of connector logic. A *template* is a composition of template-controllers, template-connecters and their respective associations. A *template-controller* is symbolic of a controller and a *template-connector* is a connector that is linked with one or more template-controllers. The logic of the template-connector is bound to the interface of the template-controller. To reuse the logic of the template-connecter it must be bound the interface of an actual controller. The template-controller is therefore a mechanism to display the template-connector's current bindings and to allow the programmer to easily modify those bindings.

Gate-mapping is the process of changing the bindings of a template-connector from a template-controller to controller, and thereby change a template-connector into a connector. The term gate-mapping came from the fact that the template-connector's logic is bound to specific gates on a specific template-controller and these bindings must be mapped to different gates, likely with different names, on an actual controller.

5.5.1.2 Hyper-States

A *hyper-state* allows the programmer to define one transition from multiple states. As portrayed in the syntax section 5.5.2.2, any state may belong to a hyper-state. If the connector's state moves to a state that belongs to the hyper-state and a message arrives of the type accepted by the input associated with the hyper-state, then the thread associated with the hyper-state will be executed. This implies that the state/input combination cannot be defined elsewhere in the connector, as ambiguity would result. This language construct is especially useful for re-initialization and exception handling.

The hyper-state is disjunctive, that is the fsm is in one of the states that the hyper-state contains, but the hyper-state itself is not an independent state. This implies the fsm is cannot be in two states at the same time. The system could be augmented to generate code in a conjunctive fashion, there by implying that the hyper-state itself has some semantic meaning. As this a paradigm shift that has no change in the visual

representation of a program it may create considerable confusion and may require the addition of a new symbol into the language.

The hyper-state supports the concept of a *default state.* A default state is analogous to the initial state in UML Statecharts. The default state is the state that is transitioned to when there is a transition to a hyper-state. Therefore a default state is only relevant when the hyper-state is being transitioned to and is irrelevant when transitioning from a hyper-state.

5.5.1.3 Default Inputs

Default inputs handle all non-specified incoming messages. Normally, if the connector receives a message of a type not specified, it is discarded. The default input allows all message types to be accepted. All non-default input statements are checked before the default input statement is checked. If the connector receives an unspecified input while in a state that has a default input, the flow of logic is along the default input's path. This reduces the complexity of connector logic for error detection or exception handling.

5.5.2 Extended Syntax

Extended concepts were implemented at both the system level as well as the connector level. At the system level, the extended syntax adds connector templates. Hyper-states and default inputs are added to the connector-level syntax.

5.5.2.1 Templates

Templates are a reuse mechanism for connector logic. Figure 5.6 shows a connector attached to two yellow controllers and a Template, composed of a Connector attached to two red controllers. The red controllers represents *template-controllers.* The connectors attached to the template-controllers are referred to as template-connectors. The red is to indicate that the configuration is not consistent. A red controller is called a template-controller. Template-controllers are place-holders for actual controllers. Hold specific interface definitions to which template-connectors are bound.

A template can be created by importing any pre-existing system definition as a

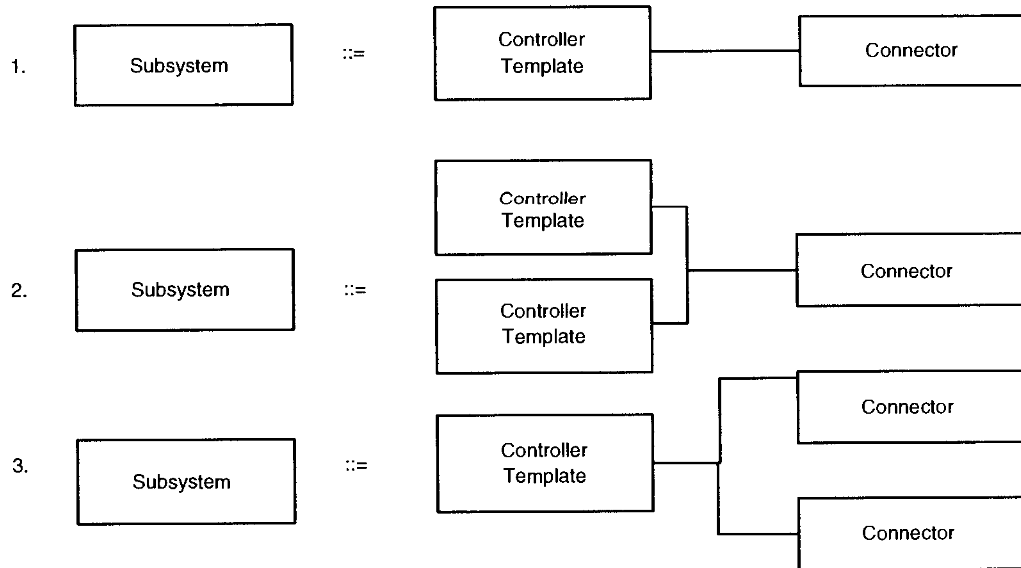

Figure 5.7. *Additions to system level syntax*

template. All connectors, controllers and channels that existed within the imported definition are shown as connectors, template-controllers and channels. The imported connector logic can be associated with different controllers by dragging a template-controller to a controller or a controller to a template-controller. Figure 5.7 defines additions to the language core's system level syntax.

5.5.2.2 Hyper-states

Figure 5.8 shows the hyper-state visualization. A hyper-state can be used anywhere a state is used. In the list view the highlighted item is the default state.

5.5.2.3 Default Input

The default input's visualization is an input symbol with the word default labelling the symbol. Figure 5.8 provides an example of its use. It can be used anywhere an input is used. There can only be one default input per state as all other non-specified inputs are subsumed by the use of a default input.

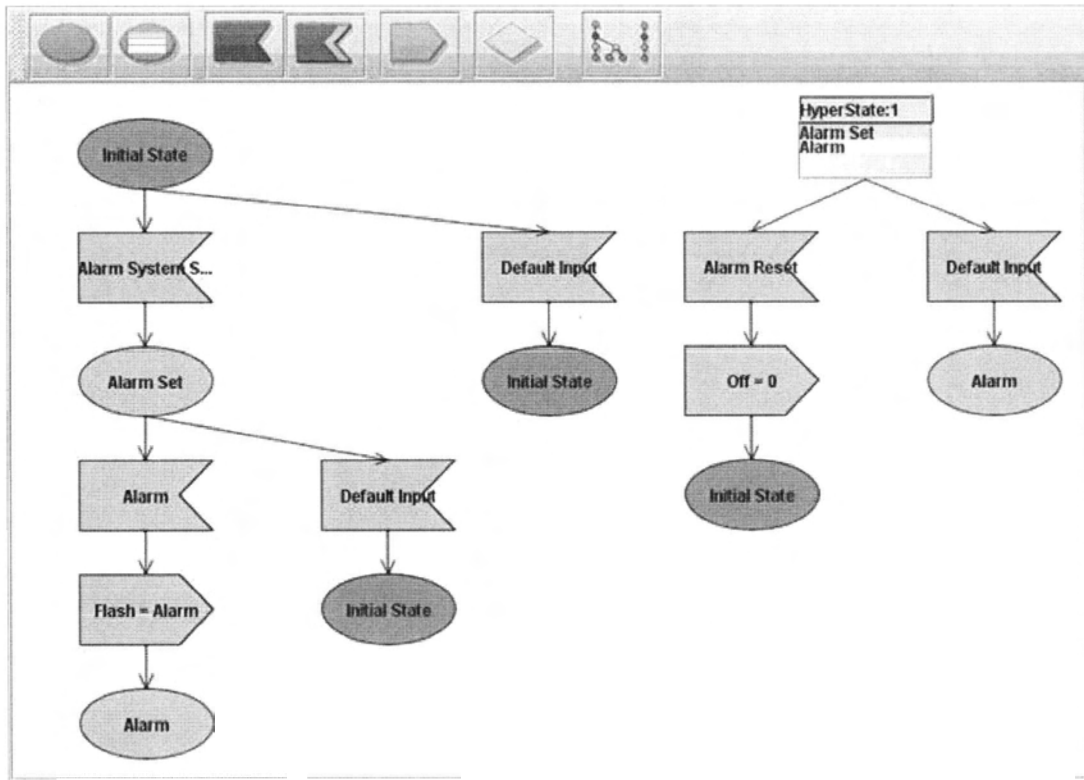

Figure 5.8. *Sample screen shot of extended language concepts*

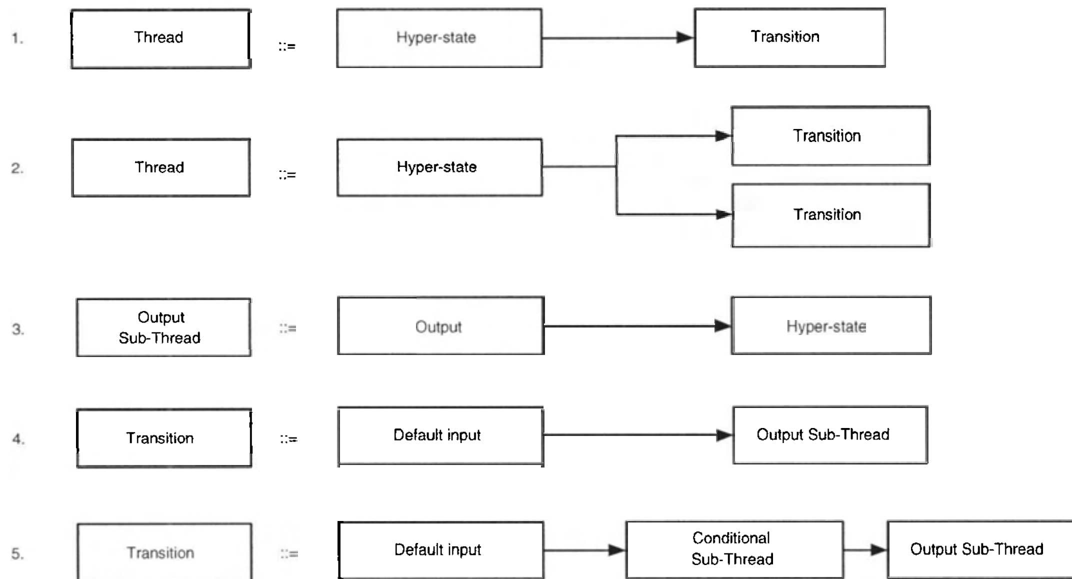

Figure 5.9. *Additions to connector level syntax*

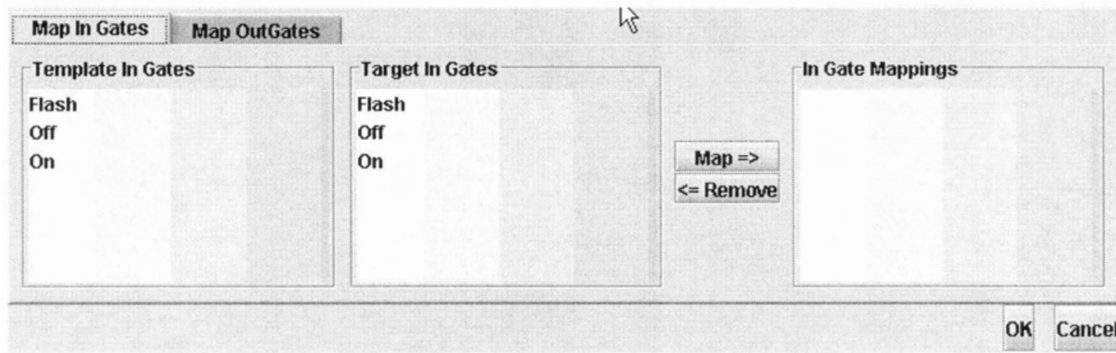

Figure 5.10. *Mapping of Gates in microSynergy*

5.5.3 Pragmatics

In order to provide insight into the use of the microSynergy language, a few simple scenarios are offered below. The first scenario is based upon a home user. The *home user* is someone who has no training in computer programming and is largely ignorant of the complexities of computing systems. The second scenario is based upon a professional software engineer who is managing a large complex system. The software engineer has training and is aware of the complexity of system with which they are dealing.

5.5.3.1 Home automation

The home user buys a device which they wish to integrate into their existing network[2].

Work Process Overview

1. The user plugs in the device, if it is not wireless
2. The user uses microSynergy to introspect the network
3. The new device appears in microSynergy's system-level view
4. The user defines logic to integrate the new device into the system
5. The user downloads the program into the system

Detailed work process

The home user buys a new lamp and wants it to flash when the security alarm is

[2]This is not necessarily a good design in practice and is meant only for illustrative purposes

tripped. The manufacturer includes in-gates and out-gates that allow the state of the lamp to be queried and the state of the light to changed along with a flash option. The user plugs in the lamp and walks over to their PC. microSynergy introspects the system to refresh the list of all controllers in the network as well as their in-gates and out-gates. The microSynergy environment visualizes all the controllers on the network. It is clearly visible that the new lamp is not connected to any other controllers.

The user then draws a line between the lamp's representation and the alarm system controller. A new *default connector*, which has a minimal set of logic that passes signals from one controller to the next, is created that connects the lamp's controller to the alarm system controller. The user opens the connector and sees that the input and the output in the thread have no in-gates and out-gates associated with them. The user right clicks on the input and is given a list of the available outputs from the various controllers. The user has only to choose the "alarm tripped" out-gate. The user then right clicks on the output and is given a list of in-gates that exist on the connected controllers. The user has only to choose the "flash" in-gate. The program is then complete. The user closes the connector view and downloads the program to the network.

When the alarm is tripped a signal is sent from the alarm system. The signal is then forwarded by the microSynergy runtime system to the lamp's in-gate. As long as the in-gate receives alarm tripped, messages the light will continue to flash.

5.5.3.2 Industrial Automation

Work Process Overview

1. The engineer assembles all the physical parts of the system
2. microSynergy introspects the system
3. The engineer replaces template-controllers with actual controllers
4. The engineer downloads the new coordination logic to the system

A firm may be opening a new production facility. Their current facility simply does not have the resources to produce as much of the product as desired. Therefore more machinery is assembled to increase production. The company already has a

tried and true configuration that they wish to reuse. The system's hardware is all assembled and networked.

microSynergy introspects the system finding all the controllers on the network. It generates a visualization of all controllers on the network. The logic from the previous system is then loaded as a template. The previously built logic of the template maps to the new controller's in-gates and out-gates via a simple drag and drop interface. The gates of the template-controllers are then mapped to the gates of the new controllers via the gate-mapper dialog. For each template-controller that is dragged onto a controller a dialog pops up listing in-gates of the template-controller and the desired controller on a tab and similarly for out-gates. The programmer can select which template-controller in-gate matches with which controller in-gate by select the two in-gates and pressing the button labelled map, as can be viewed in figure 5.10. When all gates of a template-controller are mapped to the gates of the controller the template-controller disappears and the associations adjust within the connector. If any logic must be modified the engineer has only to double click on a connector to view and modify the logic. The engineer is assisted in their wiring process by highlighted connection points on connectable symbols. When all template-connectors become connectors and any other changes are complete the program can be downloaded to the system by the click of the download button.

5.6 Application Scenario

The following scenario was constructed by the NetLab team[3] to help illustrate the value of microSynergy concepts and collaborative networking.

When dealing with terminally ill patients that are in a great deal of pain and are often placed in hospital-like environments and given morphine injections to ease their discomfort. Not only is this an expensive solution, it institutionalizes the ill person robbing them of comfort and familial contact. Morphine dispensing technology is available to be placed in the home, but there are concerns with respect to over medication and under medication if the illness grows more severe. The optimal scenario would be to allow the patient to stay at home and allow the doctor to check in with

[3]The NetLab team is composed of graduate students and employees managed by Jens Jahnke

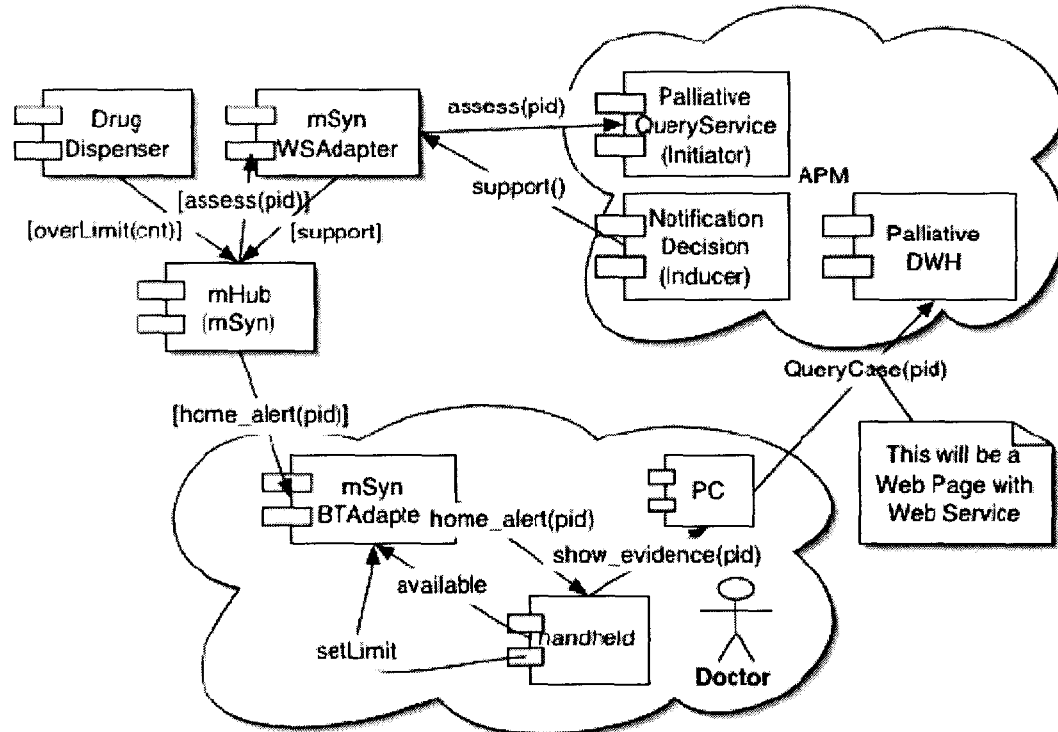

Figure 5.11. *A interaction diagram describing the interaction of components*

the patient from time to time. microSynergy could facilitate this.

The patient could be issued a dispenser, a pump[4] attached to a morphine supply, that pumps directly into a patient's vein when they push a button. If the pump were controller-based, it could act in an intelligent fashion. The pump could monitor the times and quantities of drug dispensed and be remotely monitored and adjusted. This technology would work for any intravenous drugs.

5.6.1 Details

Each button press would dispense morphine up to a daily maximum. If the maximum were reached, the doctor could be notified via their PDA, cellular or other device. The doctor would have the ability to remotely 1) increase the daily limit, 2) dismiss

[4]The demo used lights instead of real pumps.

the notice, or 3) compare the requests of the patient with upto date demographic data. The doctor would only need to press a button on their device to perform any of these actions. The population statistics would allow the doctor to make educated and informed decisions with respect to the patient's demands. If the insight desired by the doctor is still lacking, they could initiate an Internet-based interview with the patient, via tools such as NetMeeting, to get more insight into the patient's status.

The doctor may desire interfacing the drug dispenser with a heart monitor. This can be done via the following process:

1. Adding the heart monitor to the network
2. Opening the microSynergy editor and introspecting the network
3. Connecting the visual of the heart monitor to the connector component
4. Adjusting the logic of the connector component

Chapter 6

Evaluation and Case Study

6.1 Background

In an attempt to provide direction and determine the value of our ideas, we built an initial prototype followed by an second prototype. The initial prototype was for an initial evaluation, i.e. a pilot study, aimed at discovering any significant issues that were not adequately addressed. Many issues in the initial prototype were found by the developers at Intec Automation Inc. (Intec), who were the users in the pilot study. The issues found were addressed, and a significantly better a prototype was used for the secondary evaluation, i.e. the user study. Note: The original prototype did not have a template mechanism. The secondary evaluation attempted to determine if the ideas and implementation were sound and if they provided an improvement over established paradigms.

6.2 Results of Pilot Study

Intec found two areas of weakness that needed to be addressed in our original prototype. The first was the lack of reusability of communication logic. Using the original prototype, the user would be able to specify the communication logic within a specific network. However, there was no way for the user to apply that logic to similar networks.

The second area of weakness was in the usability of the user interface. In developing the prototype, we had focused on creating functionality. For the second prototype, we knew that non-technical users would require as much help as possible from the user interface to understand how to utilize the functionality. Users would

require multiple ways of performing the same tasks. Users would also need the steps to perform the task to be made as simple as possible.

6.3 Improvements to Prototype

6.3.1 Templates

Templates are a mechanism we added to aid in the reuse of connector logic. When a connector's logic is created, the logic binds to specifically defined gates on known controllers. Templates allow a connector's logic to be reused with other gates and controllers, by redefining what gates the connector logic binds with.

Figure 6.1 shows a connector. Connector1, connected to two template-controllers, i.e. red boxes. The template is composed of Connector1 and all of the template-controllers to which the connector is linked. To easily reuse the connector logic of the template, the programmer has only to indicate which controller will assume the role of a template-controller, i.e. match up a yellow box with a red one. To match up a template-controller with a controller, a user needs only to drag a connector to a template-controller.

Once the user has indicated which controller they want to match with which template-controller, they are presented with the Gate Mappings dialog shown in 6.2. The dialog indicates which gates exist on the controller and template-controller, and provides a graphical interface to map controller gates to template-controller gates. With the addition of the mapping mechanism, communication logic can now be used on any network meeting the minimal requirements specified in the template.

6.3.2 Improvements to the user interface

In improving the user interface, we tried to make it as accessible as possible. We first looked through the application for confusing text that could be made simpler. For example, we changed the label "introspect" to "synchronize with network."

We also gave the users multiple ways of accomplishing each task. We added a menu rather than just having the operations accessible via the toolbar. We also added some shortcuts. For example, linking two controllers automatically creates a

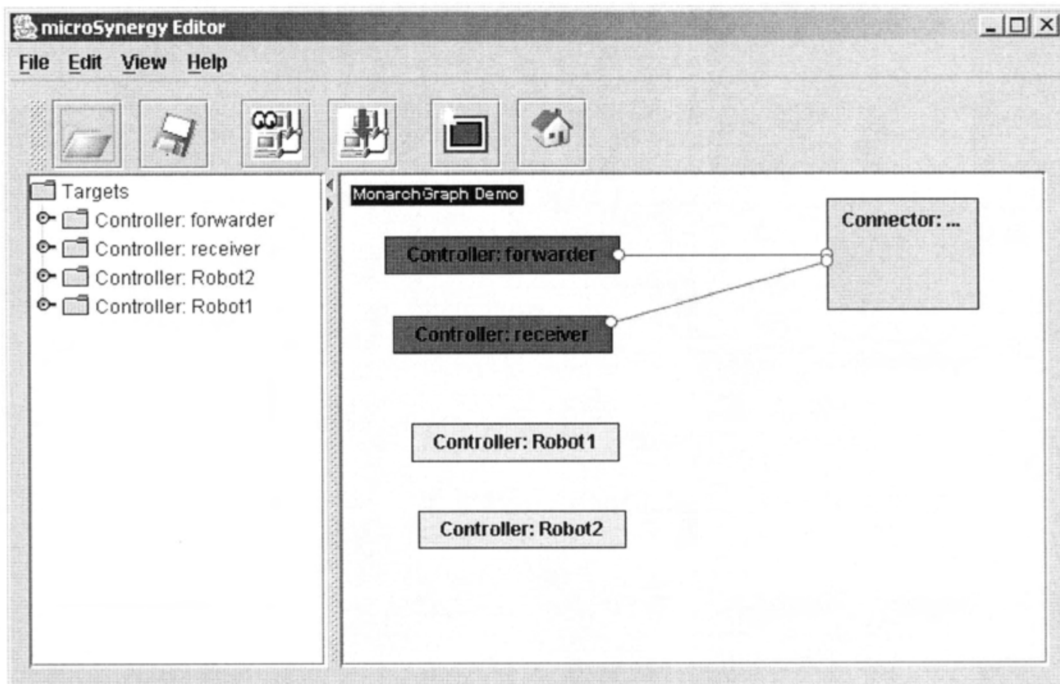

Figure 6.1. *Templates hold logic for later reuse*

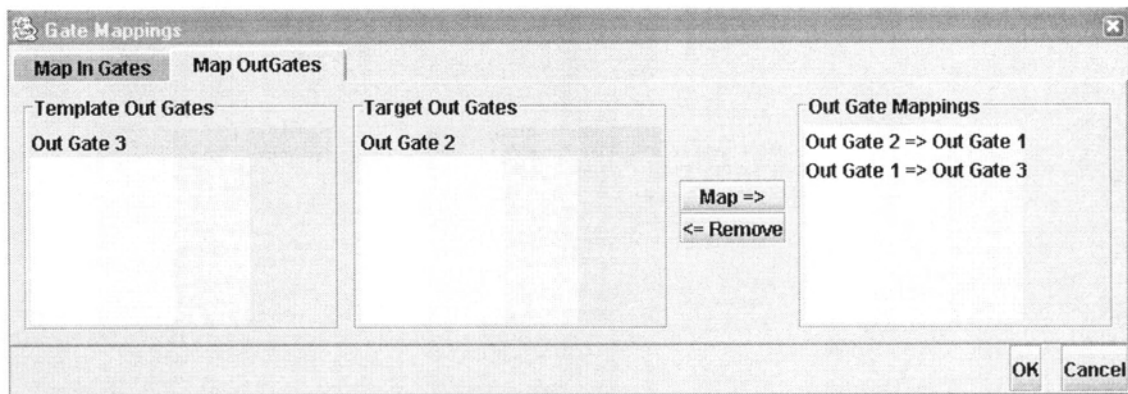

Figure 6.2. *The gate mapping interface*

connector between them.

6.4 Goals of Main Study

The goal of the study was to evaluate the suitability of the prototype tool for use in an actual industrial product. Before we began the study, we decided that it must perform three functions: (1) determine if the approach scales to industrial-strength application scenarios, (2) determine if the selected visual modelling paradigm and process can be understood by non-software engineers, and (3) discover what refinement steps should be undertaken in order to turn the prototype into a robust tool.

6.4.1 Scaling to industrial-strength

Determining if the approach scales to industrial-strength application scenarios is difficult. Each industrial case is different and has vastly different and often conflicting requirements. For example, the number one requirement for using microSynergy in the networking of a hospital might be fault-tolerance. We would need to guarantee the reliability of the network. However, each improvement in fault-tolerance might increase the minimum requirements for micro-controllers in a microSynergy network. This would increase the cost of our solution, and low cost might be the number one requirement for using microSynergy in home automation.

We decided, in the short run, that microSynergy should be a general-purpose tool for non-critical environments. It could be modified to meet the requirements of many different users in many settings if the need entailed. We would allow Intec, our industrial partner, to evaluate the prototype microSynergy and its suitability to their industrial domains.

6.4.2 Intuitive Visual Modelling Paradigm

Software engineers already have the ability to create networks of controllers. One of the goals of microSynergy was to drastically decrease the cost and development time for software engineers to create these networks. However, another goal was to enable non-software engineers to quickly and easily create these networks.

By non-software engineer, we do not mean the "average" person. We agreed that our evaluation would make no attempt to interview enough people to claim that the average person could use the prototype. Obviously, it would be extremely difficult to explain the concept of networking controllers to someone who knows almost nothing about computing. Instead, we wanted to ensure that computer users who had never programmed before could use microSynergy. A primary reason for our use of a visual modelling paradigm was that it should be easy to create and modify well designed code. Another reason was that we should be able to guide the user in creating code by preventing them from making mistakes. If our visual modelling paradigm met our expectations, then non-software engineers should be able to understand it with minimal training.

6.4.3 Prototype as a Method of Discovery

In order to discover the necessary refinement steps, we planned to make conclusions based on the data gathered in the study. In order to determine what functionality needed to be added to our prototype, we would examine how well our prototype scales to industrial-strength application scenarios.

To determine how much the user interface of the prototype needs to be improved, we would examine how well non-software engineers understood our prototype's visual modelling paradigm and process.

6.5 Methodology

According to Nielsen [18], 5 subjects results in 75% of usability problems being discovered. For the formal evaluation of our prototype, we interviewed 5 subjects drawn from a wide range of experience in software engineering. Unfortunately, due to time and budgetary requirements, a larger study was not possible.

Each subject was taken separately to our laboratory where the equipment was set up for them. We began by giving the participant a handout briefly describing microSynergy. After the subject had an overview of the program, we asked a series of questions about their prior knowledge of software engineering, distributed programming, and SDL. Based on the subject's answers, we grouped them into one of three

categories: beginner, intermediate, and advanced.

We then had the subject complete a series of tasks. One evaluator explained the task to the subject and answered any of the subject's questions. The other evaluator recorded what the subject was doing, any comments that the subject made, and the time that it took the subject to complete the task.

While the subject performed the tasks, we asked them to describe what they were doing, what they were thinking, and any confusion that they were having. At the end of each task, we asked the subject to rate the difficulty of the task and provide any suggestions about what could have been done to make the task easier.

After the subject had completed the tasks, we asked them to answer a few questions regarding the overall usability of the system. Finally, we asked the subject to provide us with any suggestions that could be made to improve the program.

6.6 Tasks

There were three categories of tasks that we asked the subjects to perform: general, logic, and template tasks.

We first asked the subject to go through a set of very simple tasks. They were designed to give the subject a basic orientation to microSynergy so that the user would have some idea as to where options may be found in the interface as well as to remove intimidation by achieving success. The simple tasks also allowed us to measure the general understandability of the interface. Most of the tasks could be done with the click of a button. They did not involve any specification of SDL logic or reuse of logic.

6.6.1 General Tasks

Synchronize with the network To complete this task, subjects had to either press the "Synchronize with network" button in the toolbar or use the menu item in the edit menu. This would perform an introspection of the network and return a list of controllers and their gates to the user.

Make a new connector To complete this task, subjects had to press the "New Connector" button in the toolbar. This would create a new connector node.

Connect two controllers To complete this task, subjects had to press and hold down the shift key while they dragged the mouse from a controller to a connector.

6.6.2 Logic Tasks

The specification of logic was primarily designed to determine whether SDL was an appropriate language for users and to determine if the meaning of the SDL symbols was intuitive and understandable.

Specify the Forwarder Logic For this task, subjects were asked to specify logic so that in-gate 1 would receive whatever value was in out-gate 1. A sample solution to this problem is shown in figure 6.4.

Specify the And Logic For this task, subjects were asked to specify logic so that in-gate 1 would receives a 1 if both out-gate 1 and out-gate 2 have produced a 1. A sample solution to this problem is shown in figure 6.5.

6.6.3 Template Tasks

Finally, we asked users to use templates to reuse the same logic that they had previously attempted to specify by hand.

Import the Forwarder Template To complete this task, users had to select "Import Template" from the file menu, and then choose the forwarder template file from a list of acceptable files.

Map the Forwarder Template's gates to the actual controller's gates To accomplish this task, users had to first select the template and actual controller that they wanted to map. The user could indicate this mapping by either selecting both and choosing the "Map" item in the edit menu, or drawing a link between them.

Use the And Template To accomplish this task, users needed to import the template, and then properly map the gates.

6.7 Task Evaluation

To help us better understand user interaction with microSynergy, we attempted to classify the subjects into either of a Beginner, Intermediate or Expert class. Subjects were asked a number of questions to get insight about their pre-existing skill set and their own impressions of their skills.

6.7.1 Quantitative Evaluation

The sampling of five subjects limits any value of the quantitative analysis to being informal, and not statistically significant. The hope was that the timings of each task would solidify our human observation of human computer interactions with the microSynergy system. Figure 6.6 summarizes the task timings. For each task we asked the subject to rate the level of difficulty, from 1 (very easy) to 5 (very hard). Graph 6.3 summarizes the results of the users perceptions.

6.7.2 Qualitative Evaluation

The first three tasks were supposed to be easy. They were designed to remove intimidation, provide confidence, and familiarize the user with the tool. The first two tasks were seen as easy, but the third task, connecting controllers was seen as quite difficult. The third task required connecting diagram elements. Monarch Graph, the toolkit that we were using to draw the nodes on the screen, specifies that holding down the shift key and dragging the mouse creates links. As this is not standard, it has a tendency of confusing people.

Using templates was seen as easier than specifying the logic, but not by much. This is probably because the users had difficulty with the gate mapping. They were not sure what gates referred to what. This could have been solved with better names associated with the gates.

For example, figure 6.1 shows the situation where a template with Out Gate 1 is being mapped to a controller with two out-gates, Out Gate 1 and Out Gate 2. When designing the mapping dialog, we assumed that the subjects would know what gates to map as gates should have reasonable names. The usability study pointed out that this would not always be the case, and it could therefore confuse users.

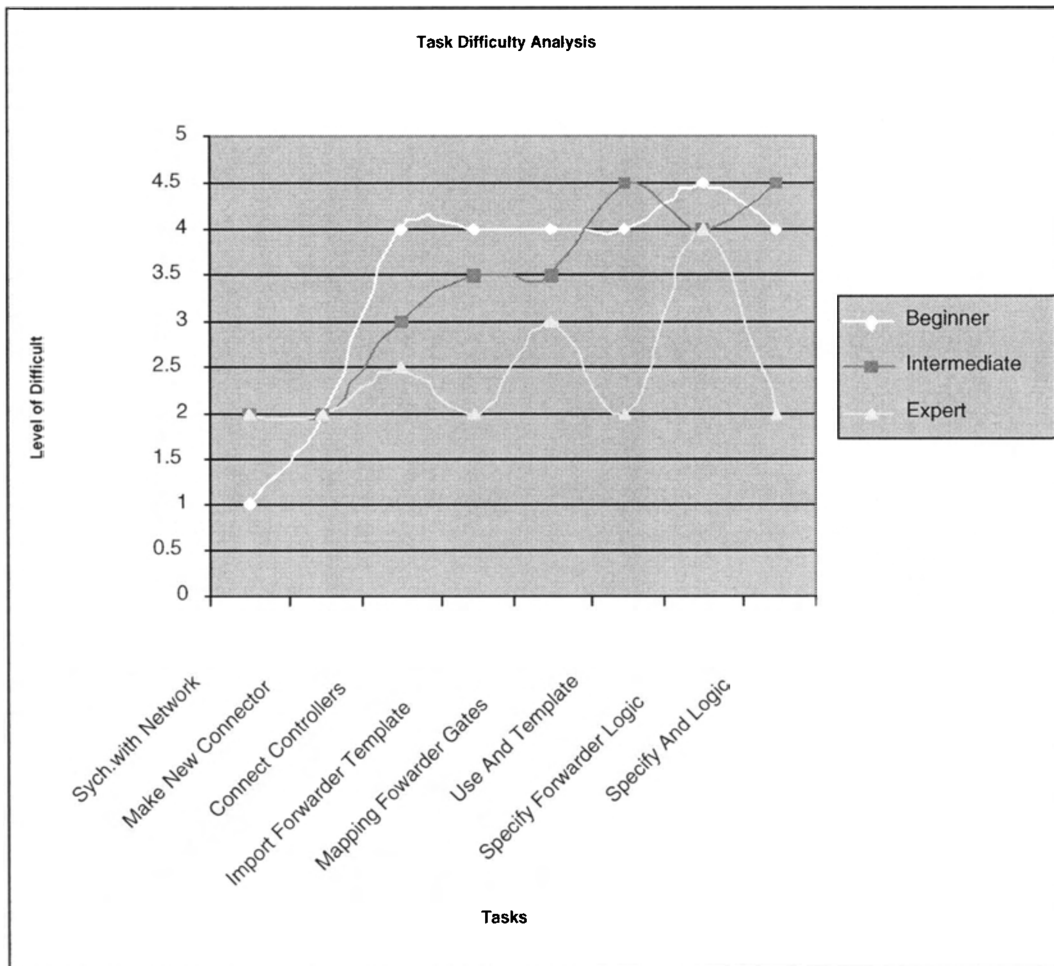

Figure 6.3. *Graph of perceived difficulties*

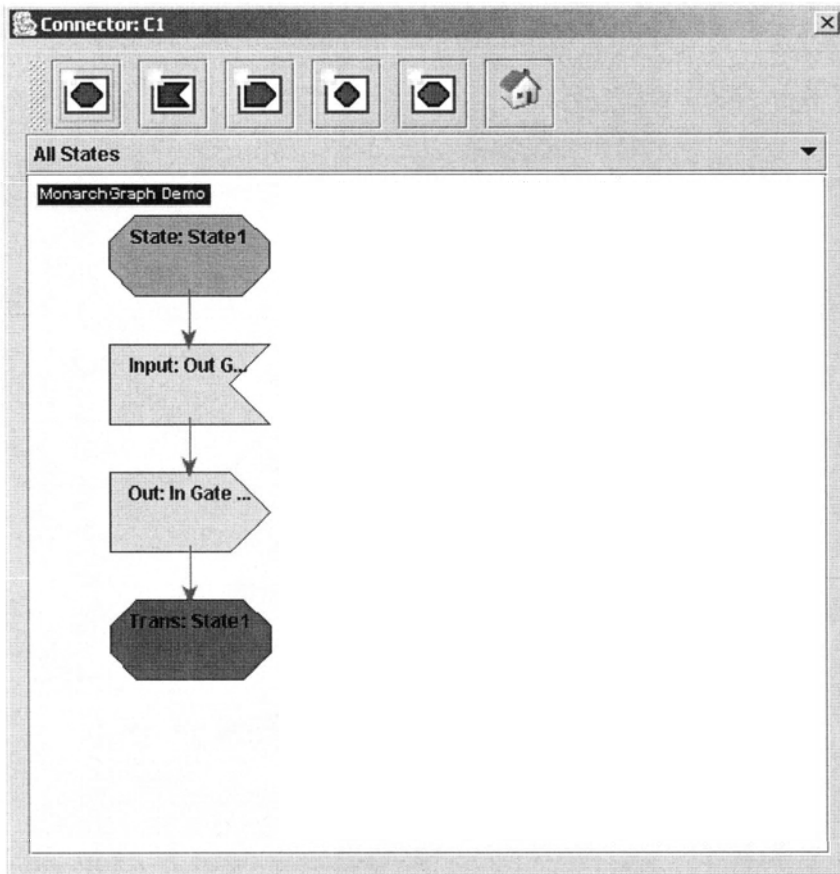

Figure 6.4. *Forwarder logic*

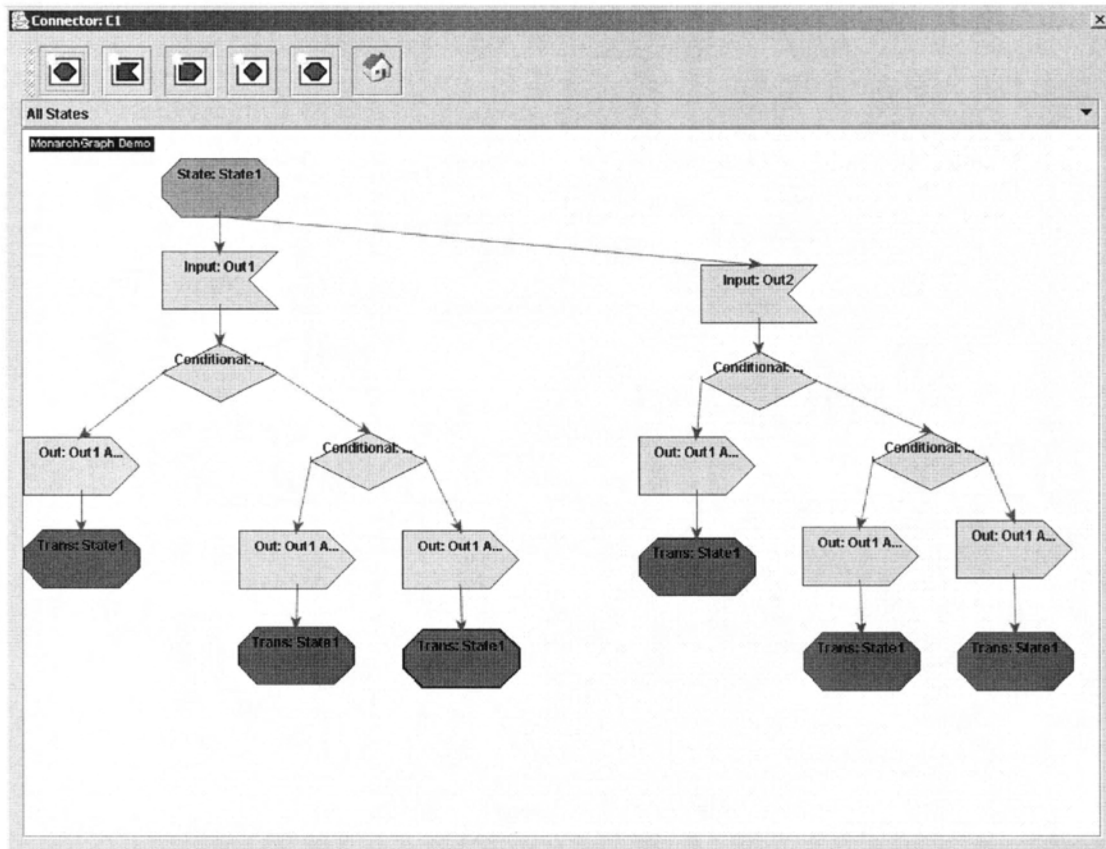

Figure 6.5. *And logic*

Specifying the Forwarder logic, shown in figure 6.4, which we assumed would be much easier than specifying the And logic, shown in figure 6.5, was seen as being just as difficult. This is probably because the subjects found the hardest thing about the logic was its finite state machine representation, not the complexity of the logic. Also, much of the knowledge required to complete the task was in the documentation they were to read. If the subjects did not read their introductory documentation very carefully or understand the material the task would have been difficult.

Our failure to properly emphasize the distributed nature of the programming environment also contributed to difficulties in the specifying logic. This led to confusion about why inputs and outputs were necessary and what role the connecter was to play in the communication.

6.7.3 Task Timings Analysis

While the subjects tried to complete the tasks, we timed how long it took them to attempt to complete the tasks. If the subject was getting too frustrated with the task, we stopped them, but ranked their time as 400 seconds.

Comparing how long tasks actually took subjects in comparison to how difficult the subjects felt the task was led to some interesting observations. For instance, the expert users felt that specifying the And Logic was "Easy", yet it took them twice as long as specifying the forwarder logic. This can partially be explained by the learning curve of the tool. When specifying the forwarder logic, the subject needed to learn our visual language paradigm. Although the forwarder logic was easy, grasping this paradigm was hard. The And logic was a much harder problem, but it felt easier, because the subjects had already learned how to use the tool.

Another interesting observation is that both beginner and intermediate users felt that importing the forwarder template was just as hard as mapping the forwarder template's gates. However, their timings show it was much easier. This can be explained in part because the subjects were still trying to learn about templates. Mapping is a harder task, but subjects were just as confused about what templates did as anything else.

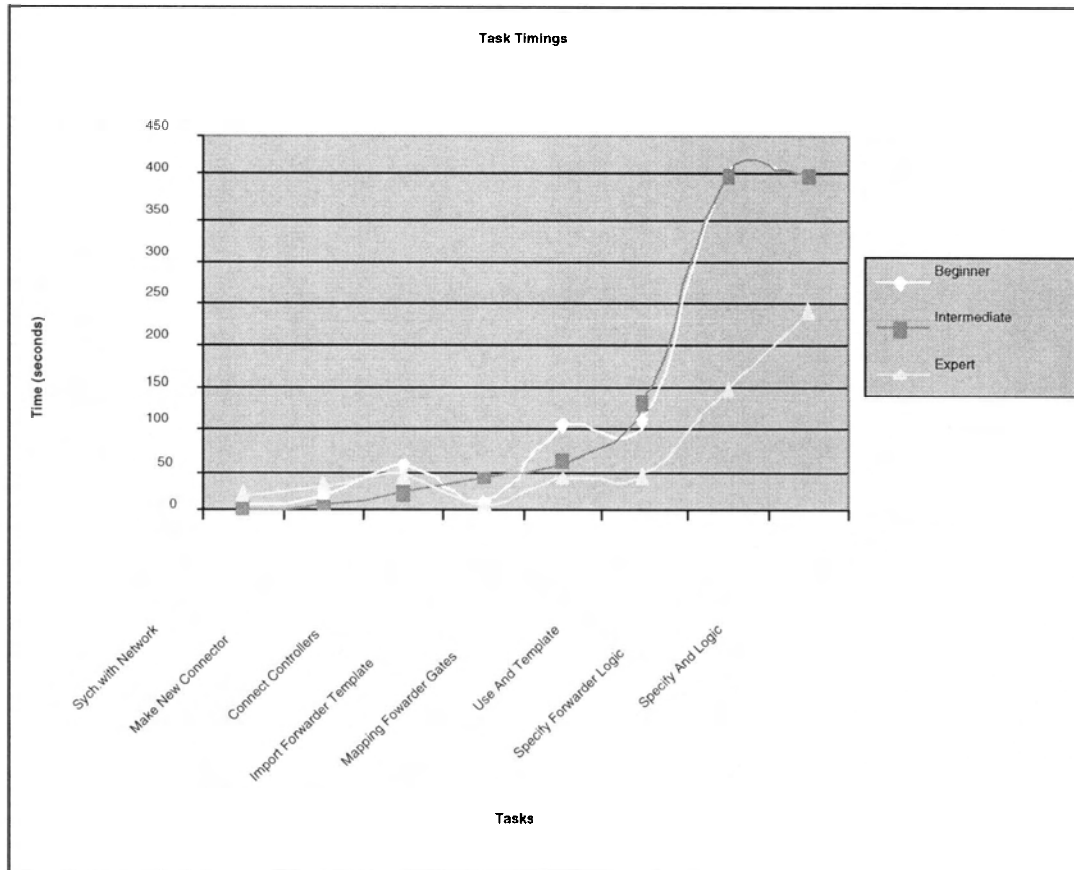

Figure 6.6. *Graph of the timings based upon user class*

6.7.4 Qualitative Observation

While the subject was completing the task, we asked them to describe what they were doing. After the subject was finished, we asked them to make any comments about the task. Below, we have summarized the findings.

6.7.5 General Tasks

6.7.5.1 Synchronize with the network

Subjects all found this task easy. Two observed that our "download" icon looked quite similar to our "synchronize with network" button.

6.7.5.2 Make a new connector

Subjects all found this task easy. However, 3 out of the 5 subjects first checked in the menu to see if they could accomplish the task there.

6.7.5.3 Connect two controllers

Subjects found this task hard. The Monarch Graph shift-drag mechanism for linking is hidden and non-intuitive. Two subjects also commented that they would like a link tool that they could select and then link things. This would be a big advantage, because it would make the mechanism for linking much more visible.

6.7.6 Logic Tasks

6.7.6.1 Specify the Forwarder Logic

Subjects found this task hard. There was confusion about why inputs would be related to out gates. When attempting to specify the logic, a number of subjects used double-clicking to try to reach the properties menu.

The greatest problem seemed to come from trying to understand the concept of finite state machines. With the exception of expert subjects, subjects were confused about why a thread of execution would begin with an input. Subjects were also confused about the purpose of states and transitions.

6.7.6.2 Specify the And Logic

Subjects found this task hard, though not much harder than specifying the forwarder logic. Expert subjects were now familiar with our concept of finite state machines. Other subjects were still confused.

6.7.7 Template Tasks

6.7.7.1 Import the Forwarder Template

Subjects completed this task fairly easily, but they were quite confused about what templates were for. Most subjects were surprised to find logic in the template's connector.

6.7.7.2 Mapping of Forwarder Logic

The mapping of template's gates to the actual controller's gates was viewed as a difficult task. They all complained that they were not sure what gates to map to which. We were expecting this to cause problems for subjects and were already planning how to fix this problem in the next release of microSynergy.

6.7.7.3 Use the And Template

To accomplish this task, users needed to import the template, and then properly map the gates. As with the forwarder template, subjects found this task easy to accomplish, but difficult to understand what they are doing.

6.8 General Comments

After the survey was completed, we asked the subjects if they had any general comments or suggestions on how microSynergy could be improved. The subjects discussed a number of usability problems that they had noticed. We predicted some of the usability problems, but were quite surprised at others. We have also gained a number of useful ideas about how to improve the program.

One usability problem that we predicted was the linking of nodes. Subjects had great difficulty with this. Monarch Graph, the component that we were using to

do the graphing is very limiting in this respect. We have since solved this usability problem by migrating to a more flexible graphing component called JGraph.

Another usability problem we predicted was that people would have difficulty with mapping gates. We are currently exploring some solutions to this problem.

The difficulty our subjects had with importing templates surprised us. We will need to make it clearer to subjects what templates represent. One way of accomplishing this might be to add a description of what each template does to the template and have a template preview tool. Subjects could view the description and preview the template diagrams when importing or browsing templates.

We were also extremely surprised by how difficult subjects found it to specify logic. We were predicting that subjects would find it difficult the first time, but the next time would be much easier. Only the expert subjects demonstrated this learning.

However, a lot of the confusion over specifying the logic stems not from our visual language but from distributed nature of the system and the finite state machine paradigm. Some subjects were confused about the purpose of states, why states needed inputs, and the purpose of transitions. It will be very difficult, much more difficult than expected, to introduce this paradigm to beginner and to intermediate subjects.

One suggestion that we had from a subject for improving the specification of logic was better and more numerous error messages. Currently, the program prevents specifying invalid links. We could improve this by adding messages that explain to the user why the link they have just attempted is invalid.

Another suggestion we had from a subject for improving specification logic was to have each connector be initialized with some trivial logic that would serve as an example. This would probably have value for beginner and intermediate users, but would annoy experts. A preferences menu, where we can let the user customize a number of preferences for the program.

6.9 Conclusions

The usability study, as expected, found a number of areas of weakness in microSynergy. As a research prototype, we did not expect subjects would find the product

very easy to use, but were pleased that the subjects could perform many of the tasks and the tool's concepts could be learned quickly.

The great value of the usability study has been in showing specific areas of weakness in microSynergy. After reviewing our results, we have come up with a number of suggestions for improvement:

- Add all toolbar options to the menu. We felt that users would find it more convenient to view the "New Connector" icon, rather than the "New Connector" menu item. However, this confused users.
- Add a help menu. Users all, at one time or another asked for help features.
- Migrate away from Monarch Graph. This could enable us to enable drag and drop for mapping templates, and create a more visible mechanism for linking nodes.
- Add a description of the template, and allow the user to browse and search by name and description.
- Improve the mapping mechanism.
- Add error messages if users try to make invalid mappings.
- Add a preferences menu, with some settings to help beginner users. Some suggestions are initializing a new connector with basic logic, showing the type of all the nodes, and adding labels to the icons.
- Make the download icon and the synchronize with network button less similar
- Look at creating a simpler view for beginner users that does not require knowledge of SDL. Perhaps they could draw lines between gates shown on the controllers.

Chapter 7

Reflections and Future Work

7.1 Contributions

microSynergy has examined the federating of controller-based devices into unified coordinated systems across a variety of protocols. The collaborative system which is created allows devices to maintain their own functionality and only minimally impacts the existing micro controller code base. Issues such as code reuse, network evolution and program understanding were address by the creation of a visual language. The multi-tiered design of the language allowed simple interrelationships among devices to be view in a very simple fashion, i.e. as links between boxes. The details about how the devices where coordinated was defined with an SDL like language. We believe that combination of modified finite state automata and connection-based programming allows programmers to rapidly understand and define programs that would normally take significantly longer to understand and build.

Via the development of a prototype, the project tackled some of the problems of simple device development such as:

- decreasing the time and complexity of creating simple interactions among many devices
- integration of multiple design and development elements into one unified tool, thereby providing a unified platform that assists the understanding of a program as well as its modification
- giving the designer/developer to ability to model the network, recognize the current configuration and evolve the configuration
- reuse of coordination logic was simplified by adding tool support and assisting

the user to make reasonable decisions

- connecting elements in a sensible fashion by highlighting only specific acceptable targets

7.2 Future Work

7.2.1 Arbitrary Data Types

Currently the contents of the messages are defined as 16 bit integers. Although this allows quite a bit of functionality, it is restrictive. Allowing messages of any type would simplify passing complex data which is required in many domains. Unfortunately, it would also complicate many aspects of microSynergy. Conditional statements currently examine the contents of messages. They would have to be redesigned to handle varying message types either at the design/deployment time of a roadmap or at runtime. This implies that programming would be more complex for the user as gates would have to have types associated with them in order to provide type safety.

7.2.2 Encapsulation of Logic

Currently, the state machines are built on a fairly flat model, i.e. connectors are not nested within, or connected to, other connectors. This limits the scalability of the VL to what the user can understand as programs become increasingly large. Although the division of the component logic from the coordination logic can dramatically simplify the visualization and conceptualization, as the networks become larger, i.e. hundreds or thousands of controllers, coordination logic could become quite complex and unmanageable [19].

The addition of nested connectors could provide greater abstraction of coordination logic and provide a more scalable modelling environment that matches the conceptual model of the developer. This could consolidate very complex logic into more understandable groupings.

The addition of gates to connectors could allow connecters to be pipelined. Allowing connectors to communicate with one another could enhance scalability of understanding as well as allow higher level service composition [20].

7.2.3 Atomization of Connector Logic

Currently, the connectors are monolithic entities. They can have pieces of logic that do not relate configuration, but are in the same connector. To better reuse connector logic the logic would be best distilled into separate connectors. The only existing mechanism to distill the logic when it is combined in such a manner is by hand via copy and paste. Although, this is functional and is the manner in which many other languages support reuse, it does not allow the tool to support mapping logic. If logical groupings could be separated and saved independently, they could be reused independently and have tool support that would remove problems and simplify development. This could leverage the concept of nesting connectors, as expressed in section 7.2.2

7.2.4 Dynamic Logic Loading

Currently the roadmap is defined in the editor and downloaded to the master controller as one unified entity. The roadmap is a monolithic entity, i.e. if a change is required, a new complete roadmap must be built and downloaded that the network be stopped, and a new roadmap downloaded and the network restarted. A system that allows part of a roadmap to be integrated with an already existing one would allow the system to dynamically acquire new logic, thus removing the need to restart the whole system.

This would require a new definition of the roadmap and a mechanism to integrate new pieces of logic and remove old pieces of logic. Runtime dependency checking of network resources would also seem important to having a stable system.

7.2.5 Roles and Auto-Mapping

Currently, the discovery and integration of embedded components is by the name associated with the gate. By associating roles with specified interfaces, many tasks done by both the developer and microSynergy could be simplified and more elegant. The association of roles to the interface of one or more embedded components could allow the dynamic mapping of template logic to a set of interfaces. This could in turn allow new devices to be added to the network and dynamically integrated with the remainder of the network.

New devices could have predefined roles, thereby implying that they have specific interfaces. Runtime could query a controller to determine its role. If another member of the network, which had the same role, dropped out of the network runtime could react by mapping the signal destined for the now absent controller to the newly integrated component. This would provide for facilities such as fail-over support. Runtime could also be easily modified to act upon the other parameters such as load and timeliness in forming routing decisions.

7.2.6 Model Checking and Verification

One of the primary reasons for having the translation from the editor's model to an XML-based file followed by the translation to a binary file was to have convenient access to the program for the purposes of analysis. The generation of well formed programs is crucial to the proper functioning of the system. Although the task of model checking an entire system as defined by microSynergy seems simple it can not be feasibly done at this time as it would also require that the embedded components be checked as well. Many embedded components are written in C/C++ and assembly code and are therefore very difficult to verify in an automated or manual fashion.

7.2.7 SOAP Services

Currently the system allows communication with proprietary message types. A PC-based proxy is used to allow runtime to communicate with SOAP- based services (.NET). In the future, runtime could communicate directly via SOAP with other computers on the Internet. This would require parsing XML messages, and mapping, and marshalling the messages to a form acceptable by runtime. The registration and discovery of new services would require our current registration system be modified to handle the reading and generation of Web Services Description Language (WSDL) and finding and choosing of Universal Description, Discovery and Integration (UDDI) services. Part of the adaptation has already started as runtime has been fitted with a web server allowing devices to directly interact with Runtime and thereby all controllers attached to Runtime.

7.2.8 Real-Time Support

Currently support for real-time systems integration has only gone as far as support for CANBus. There is no support for real-time, whether sporadic or periodic, in the operating system. Support would require selecting a real-time OS and building an analysis component that would dynamically determine acceptable load and latency for timely processing, integration and cleanup of services. For wide spread industrial adoption, as well as home audio/video support, real-time support is required.

7.2.9 Definition of a Global Component Network

The user currently has no mechanism to differentiate among devices with differing hardware or network performance profiles. This is only of concern when timeliness of data or communication becomes an issue. The components in the local network are assumed to not have substantial networking delays. If integration of real-time and non-real-time services into the same network is desired a distinction should be made. Currently, the definition of the network is defined by the introspection facility. The introspection facility scans the network for controllers all the microSynergy aware devices are registered and can take part in the collaborative network. There is no mechanism to select the most appropriate device across the Internet or across scatternets[1] as all registered devices are considered equal in resource and performance. Allowing the developer, user define how to handle traffic coming from a specific controller would be very useful when there is concern about network resource usage or timeliness of data. The system could be designed to profile the devices with which it interacts and thereby automatically adjusting which devices receive requests.

7.2.10 History State

The addition of a history state to the concept of hyper-state would provide mechanism to allow more functional nested state like logic.

[1]Scatternets are "(t)wo or more piconets co-located in the same area with interpiconet communication."[21]. Piconets are networks that allow communication between Bluetooth™devices. Devices in different piconets can communicate in what are referred to as scatternets [22]

7.2.11 Quantitative Evaluation

Quantitative empirical evaluations are subject to further work but are out of scope of this thesis.

References

[1] Mark Weiser, "Some computer science issues in ubiquitous computing," *Communications of the ACM*, 7 1993.

[2] M. Satyanarayanan, "Pervasive computing: Vision and challenges," *IEEE Personal Communications*, pp. 10–17, Aug. 2001.

[3] John Seely Brown Mark Weiser, "The coming age of calm technology," Internet, 8 1996.

[4] Veronica Falco Roy Want, Andy Hopper, "The active badge location system," *ACM Transactions on Information Systems (TOIS)*, vol. 10, no. 1, pp. 91 – 102, January 1992.

[5] Kevin Ashton, "The next computer revolution," Internet, 8 2001.

[6] Inc. RFID Journal, "Alien demos worlds first sub-10-cent rfid tag," Internet, 7 2002.

[7] Applied Digial Solutions, "Verichip corporation," Internet, 11 2002.

[8] Digital Angel Corporation, "Consumer," Internet, 11 2002.

[9] Digital Angel Corporation, "Medical," Internet, 11 2002.

[10] Applied Digial Solutions, "Thermo life," Internet, 11 2002.

[11] Front Edge Technologies Inc., "Technical information," Internet, 11 2002.

[12] Clemens Szyperski and Cun Pfister, "Component-oriented programming wcop'96," Workshop proceedings, Queensland University of Technology, Linz, Austria, 7 1996.

[13] Stefan Van Baelen, "Definition of components and notation for components (d.1.4.4)," White paper, Information Technology for European Advancement, December 2001.

[14] Lorrie Cranor and Ajay Apte, "Programs worth one thousand words: visual languages bring programming to the masses," *Crossroads*, vol. 1, no. 2, pp. 16–18, 1994.

[15] Amandeo Sarma Jan Ellsberger, Dieter Hogrefe, *SDL: Formal Object-oriented Language for Communicating System*, Prentrice Hall Europe, 1997.

[16] J. M. Alvarez, M. Diaz, L. Llopis, E. Pimentel, and J. M. Troyal, "Deriving hard real-time embedded systems implementations directly from sdl specifications,"

in *Proceedings of the ninth international symposium on Hardware/software codesign*. 2001, pp. 128–133, ACM Press.

[17] J. Rekers and Andy Schurr, "Defining and parsing visual languages with layered graph grammars," *Journal of Visual Languages and Computing*, vol. 8, no. 1, pp. 27–55, 1997.

[18] Dix Alan, *Human Computer Interaction*, Prentice Hall, Essex, 1998.

[19] Michael Beigl, Hans-W. Gellersen, and Albrecht Schmidt, "Mediacups: experience with design and use of computer-augmented everyday artifacts," *Computer Networks (Amsterdam, Netherlands: 1999)*, vol. 35, no. 4, pp. 401–409, 2001.

[20] D. Chakraborty, F. Perich, A. Joshi, T. Finin, and Y. Yesha, "A reactive service composition architecture for pervasive computing environment," 2002.

[21] Patrick Kinney, "Ieee std 802.15.1-2002," IEEE, June 2002.

[22] Ericson, "Specification of the bluetooth system version 1.1 core," Internet, February 2001, Bluetooth Core Specification.

Appendix A

An Example of a CDL File

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE NETWORK SYSTEM "CDL.dtd">
<NETWORK>
  <TARGETS>
     <TARGET compID = "1" name= "mSynergyHub">
        <INGATES>
        </INGATES>
        <OUTGATES>
           <DIGITALOUTGATE compID = "10" name = "Out Gate 1" />
           <DIGITALOUTGATE compID = "12" name = "Out Gate 2" />
           <DIGITALOUTGATE compID = "14" name = "Out Gate 3" />
        </OUTGATES>
      </TARGET>
      <TARGET compID = "10" name= "target2">
          <INGATES>
             <DIGITALINGATE compID = "10" name = "In Gate 1" />
          </INGATES>
          <OUTGATES>
          </OUTGATES>
      </TARGET>
  </TARGETS>
  <CONNECTOR compID = "1" name = "1">
    <STATE name ="1">
      <THREAD>
        <INPUT from = "" type = "" default = "true" />
          <STMT>
            <STATE name ="1">
            </STATE>
```

```
    </STMT>
</THREAD>
<THREAD>
  <INPUT from = "mSynergyHub" type = "Out Gate 1" default = "false" />
    <STMT>
      <STATE name ="4">
        <THREAD>
          <INPUT from = "mSynergyHub" type = "Out Gate 2" default = "false" />
            <STMT>
              <STATE name ="2">
                <THREAD>
                  <INPUT from = "mSynergyHub" type = "Out Gate 3" default = "false" />
                    <STMT>
                      <OUTPUT dest = "target2" type = "In Gate 1" >
                        <PARAM>
                          <VAR from = "mSynergyHub" type = "Out Gate 3"/>
                        </PARAM>
                      </OUTPUT>
                      <STATE name ="1">
                      </STATE>
                     </STMT>
                </THREAD>
                <THREAD>
                  <INPUT from = "" type = "" default = "true" />
                    <STMT>
                      <STATE name ="1">
                      </STATE>
                    </STMT>
                </THREAD>
              </STATE>
            </STMT>
          </THREAD>
          <THREAD>
           <INPUT from = "" type = "" default = "true" />
             <STMT>
               <STATE name ="1">
               </STATE>
             </STMT>
          </THREAD>
```

```
          </STATE>
        </STMT>
      </THREAD>
    </STATE>
    <HYPERSTATE name ="1">
      <DEFAULTSTATE  name ="4"/>
      <MEMBER  name ="1"/>
        <THREAD>
          <INPUT from = "" type = "" default = "true" />
            <STMT>
              <STATE name ="1">
              </STATE>
            </STMT>
        </THREAD>
     </HYPERSTATE>
  </CONNECTOR>
</NETWORK>
```

Appendix B

Annotated CEL Sample

```
1  1  14    Version, Number of Connectors, Number of instructions
1  0  0     Number of controllers, unused, unused
0  1  1     Connector number, Connector ID, Thread:1
9  1  0     Thread instruction, In instruction
1  1  10    In instruction, from target 1, out gate id 10
4  3  3     Conditional equals, 3 true instructions, 3 false instructions 1
1  1  10    Variable instruction, from target 1, component id 10
10 1  0     Constant instruction, value is 1
2  1  12    Output instruction, dest. controller id 1, component id 12
10 1  0     Constant, value 1
3  0  0     Transition, to thread 0
2  1  12    Output instruction, dest. controller 1, component id 12
10 0  0     Constant 0
3  0  0     Transition, to thread 0
```