A CONCURRENT OBJECT-ORIENTED APPROACH FOR REQUIREMENTS ANALYSIS AND DESIGN OF EMBEDDED SYSTEMS

by

Vikul Khosla

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Sciences and Applications

APPROVED:

D. G. Kafura, Chairman

J. D. Arthur

S. F. Midkiff

October, 1991

Blacksburg, Virginia
A CONCURRENT OBJECT-ORIENTED APPROACH FOR THE REQUIREMENTS ANALYSIS AND DESIGN OF EMBEDDED SYSTEMS

by

Vikul Khosla

Committee Chairman: Dennis G. Kafura
Computer Science

(ABSTRACT)

A requirements analysis approach for addressing the functional requirements of embedded systems has been proposed. Also proposed is a design approach based on the concurrent object-oriented programming paradigm. The design approach takes a specification created using the requirements analysis approach and transforms it into a detailed design. The detailed design is implemented using ACT++, a concurrent C++ that derives its concurrency semantics from the Actor model. The two approaches are illustrated by a simple but representative process control problem.

The requirements analysis approach in conjunction with the design approach provides a high level of traceability and promotes the reusability of specifications and design. Improved reliability and reduced development and maintenance costs also are potential benefits. Extensions of the work include an integrated software development environment for embedded systems.
ACKNOWLEDGEMENTS

I would like to thank my graduate advisor, Dr. Dennis Kafura, for his continued guidance during the course of my graduate study at Virginia Polytechnic Institute and State University. I have had the opportunity to take several graduate courses with Dr. Kafura as well as interact with him on a professional as well as a personal level. I have truly enjoyed working and learning from Dr. Kafura.

I would also like to thank my advisory committee members Dr. J. D. Arthur and Dr. S. F. Midkiff for their guidance and suggestions during the course of this research.

Finally, I would like to thank my family for their support and most of all my wife Mala.
# TABLE OF CONTENTS

## CHAPTER 1 - INTRODUCTION

1.1 Embedded Systems .................................................. 1
1.2 Concurrent Object-Oriented Programming ......................... 7
1.3 Organization of Thesis ............................................. 11

## CHAPTER 2 - STARE

2.1 The Process Control Model ........................................... 12
2.2 Embedded Control of a Steam Generator .......................... 15
2.3 The STARE Approach .................................................. 17
2.4 Internal Structure of a STARE state ............................... 27
   2.4.1 Environment Specification ................................... 27
   2.4.2 Controller Specification ...................................... 30
   2.4.2.1 Parameter Subsection ..................................... 30
2.5 State Transitions ..................................................... 35
2.6 Summary .................................................................... 36

## CHAPTER 3 - ACTOR BASED LANGUAGES

3.1 The ACTOR model ....................................................... 38
3.2 Summary .................................................................... 42

## CHAPTER 4 - EXTENDED ACT++ - DESIGN ISSUES

4.1 Adapting the Actor Model ............................................. 44
4.2 Design of ACT++ ....................................................... 45
4.3 ACT++ Class hierarchy ............................................... 47
4.4 ACT++ Operations ...................................................... 48
4.5 Interfacing Actors to a Non-Actor World .......................... 51
4.6 Summary .................................................................... 55

## CHAPTER 5 - DREAMS

5.1 The DREAMS architecture ............................................. 56
5.2 The Role of Extended ACT++ in DREAMS ......................... 61
5.3 A DREAMS based Actor Application ............................... 63
LIST OF FIGURES

Figure 1.1 - Correspondence between characteristics and requirements .................. 3
Figure 1.2 - Categorization of requirements specification techniques ................... 5
Figure 1.3 - A survey of concurrent programming languages .............................. 9
Figure 1.4 - A survey of Actor based languages .............................................. 10
Figure 2.2 - Embedded control system ............................................................ 15
Figure 2.3 - Steam generator ........................................................................... 16
Figure 2.4 - A state definition in STARE .......................................................... 20
Figure 2.5 - States reachable from external state startup .................................... 23
Figure 2.6 - States reachable from external state shutdown ............................... 24
Figure 2.7 - States reachable from external state normal ................................... 24
Figure 2.8 - Environment specification section of a state definition ................... 28
Figure 2.9 - An excerpt from the problem statement ......................................... 31
Figure 3.1 - An abstract representation of an actor ......................................... 39
Figure 3.2 - The tasks performed by an actor ................................................... 40
Figure 4.1 - ACT++ class hierarchy ................................................................. 47
Figure 4.2 - Summary of ACT++ operations .................................................... 48
Figure 4.3 - The two world views ................................................................... 52
Figure 4.4 - Design strategy for interface actors ............................................... 53
Figure 4.5 - Interface actors inheritance hierarchy .......................................... 54
Figure 5.1 - The DREAMS architecture ............................................................. 57
Figure 5.2 - Architecture of the controller specification ...................................... 59
Figure 5.3 - Architecture of the environment specification ............................... 60
Figure 5.4 - Detailed architecture of controller specification ............................ 69
Figure 5.5 - Topology and receptionist behavior classes .................................... 71
Figure 5.6 - Parameter subsection and collective failures classes ...................... 72
Figure 5.7 - Detailed architecture of the environment specification ................... 74
Figure 5.8 - Topology and failure classes for environment specification ............ 75
Figure 5.9 - Goal, policy, failure classes for environment specification .............. 76
CHAPTER 1 - INTRODUCTION

The problem statement for complex projects is usually a large document [Heninger 80] often plagued by ambiguity, incompleteness and inconsistencies with the consequences being realized only much later in the development life cycle at a considerable cost. Since an inadequate problem specification affects the subsequent stages of the development life cycle, errors resulting from omissions at this stage can cost up to 200 times more to correct than errors that could possibly arise later in the life cycle [Boehm 81].

The requirements analysis phase of the system development life cycle serves to transform a given problem statement into a precise, unambiguous and complete specification which then serves to guide the subsequent phases of the system life cycle. The requirements specification unambiguously states what system is to be developed and under what performance, environmental, functional and development constraints. Due to the evolutionary nature of large software projects, adequate techniques are needed to transform the problem definition into a specification that is also comprehensible and modifiable. The difficulty in analyzing the requirements for a given problem definition is common across all application domains. However, there are classes of computer systems where this problem is further exacerbated by the very nature of the problem domain.

1.1 Embedded Systems

The class of computer systems of special interest in this thesis are embedded
systems. Embedded refers to the fact that these systems are typically embedded in a larger system whose primary purpose is not necessarily computation [Zave 82]. Common examples of embedded systems are industrial process control systems, command and control systems, flight guidance systems, avionics and data acquisition and control systems. Typically embedded systems exhibit the following characteristics [Steusloff 84] [Schoeffler 84]:

- Asynchronous parallelism
- Complex system / environment interface
- Distributed
- Failure-proof reliability
- Graceful degradation
- Stringent performance.

As a consequence of these characteristics, the problem statements for embedded systems are not exempt from any of the pitfalls identified earlier, instead they are further exacerbated. Moreover, the fact that these systems are difficult to test due to the possibility of human and ecological disasters, further underscores the importance of a detailed requirements analysis phase in the system development life cycle for such systems.

The environment in which embedded systems are required to operate is highly complex not only due to the number of agents that are concurrently interacting with one another in an asynchronous fashion but also due to the distributed nature of the communication. Performance requirements, both task deadlines and arrival rates of stimuli, impose additional constraints on the communication and functional behavior of the system. Failure-proof reliability is an important safety criteria for embedded systems since the environment in which they operate is potentially unsafe. Desired levels of operational reliability and high availability of the system are typically achieved through redundancy in
both hardware and software [Schoeffler 84]. Graceful degradation refers to the non-impairment of the functional and performance objectives when the system is subjected to excessive loads from within its environment.

In addition to the complexity issues outlined above, the fact that embedded systems tend to be evolutionary with the software development process spanning several years creates maintenance costs that far outweigh the development cost. To address the evolutionary aspect of such systems, comprehensibility and modifiability of the requirements specification are important criteria for an analysis technique to satisfy.

![Diagram](image)

**Figure 1.1 -** Correspondence between characteristics and requirements

The requirements for embedded systems can be partitioned into structural requirements, functional requirements, and performance requirements [Steusloff 84], [Liskov 86]. Figure 1 categorizes the characteristics exhibited by embedded systems with
respect to their different requirements. Structural requirements address the architectural issues of how the system is to interface with its environment. Functional requirements are concerned with modelling the behavior of the system, which typically is a complex interaction based on asynchronous distributed communication within the environment. Performance requirements deal with temporal aspects of the specification of embedded systems. As shown in Figure 1, capturing the asynchronous parallelism and distributed computation in the complex system environment would render a purely functional system in the absence of any performance requirements. Further imposing task deadlines and timed responses from the environment on the functional behavior would render a system that not only performs the desired task but does so while satisfying the temporal constraints. Failure-proof reliability and graceful degradation are system-wide requirements and hence have been placed at the outermost nesting level.

Several techniques for specifying the functional behavior and performance requirements of embedded systems have been proposed. Table 1, which by no means is exhaustive, categorizes some of these specification techniques. Representational specification techniques have the advantage of simplicity at the cost of verifiability. The simplicity stems from the fact that they are informal specification techniques. Formal specification techniques on the other hand are either mathematically oriented or else have the structure and semantics of a procedural language. These formalisms facilitate the verification process, but are not easily comprehensible to all individuals involved across the different stages of the development life cycle. Procedural specifications further blur the distinction between requirements and high level design [Balzer 82]. The representational specification techniques that have been proposed are derivatives of Structured Design [Myers 78][Yourdon 78]. Real-Time Structured Analysis (RTSA) [Pirbhai 88][Ward 86] are purely representational. DARTS [Gomaa 84] is a further extension of RTSA, and it
includes guidelines for structuring a real-time system into concurrent tasks. ADARTS

<table>
<thead>
<tr>
<th>Representational</th>
<th>Procedural</th>
<th>Mathematical</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTSA, DARTS</td>
<td>PEARL, RT-ASLAN</td>
<td>RTL, RTRL</td>
</tr>
<tr>
<td>ADARTS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1.2 - Categorization of requirements specification techniques**

[Gomaa 89], an Ada-based Design Approach for Real-Time Systems is an Ada based extension of DARTS.

Mathematically oriented specification techniques that have been proposed for real-time systems are also relevant to embedded systems. RTRL [Dasarathy 85], a Real-Time Requirements Language is based on a Finite State Machine model. The verification process is facilitated by the automatic generation of test cases. RTL [Jahanian 86], a Real-Time Logic is a derivative of first order logic and is used primarily for expressing the requirements for time-critical systems. Other formal techniques based on petri-nets have been proposed for expressing the timing requirements [Coolahan 83] and for analyzing the safety properties [Leveson 87] of real-time systems.

Procedural specification languages tend to be specific to a certain application domain or else make assumptions about the environment in which they are to be used. PEARL [Steusloff 84] is geared toward distributed industrial process control whereas RT-ASLAN [Auernheimer 86] assumes that no two tasks will have their execution interleaved over one processor.

The above analysis techniques provide a rigorous foundation for representing a problem statement in a formal way. However, these modelling techniques provide very little or no guidance at all as to how the problem statement should be decomposed and
mapped into the requirements specification using the proposed formalism. Typically, embedded systems exhibit similar characteristics, making them amenable to such a decomposition.

The availability of a firm path for decomposing the behavior of the system in a structured fashion has several advantages, most notable of which are:

- Traceability
- Reusability

Traceability refers to the ability to relate a piece of code to the specification and vice versa. In addition to ensuring that the implementation conforms to the desired behavior of the system, traceability also helps reduce the development and maintenance costs for embedded systems. Having designers proceed along a firm path also promotes reusability of specifications.

Executable specification languages [Zave 82] that integrate the structural, functional and performance requirements for embedded systems often provide traceability from the specification to the implementation. However, they still lack guidelines for decomposing the behavior of the system in a structured fashion. Non-executable formal specification techniques as well as informal specification techniques certainly lack the traceability from the problem statement to the requirements specification and possibly to the implementation as well due to the lack of structuring criteria to achieve the transformation.

The purpose of this research is to suggest a requirements analysis approach for decomposing a problem statement for an embedded system in a structured and modular fashion by exploiting the characteristics inherent in the functional behavior of embedded systems. This approach could serve as a front end to some of the other work that has been done in the area of specification techniques for embedded systems. Also proposed are architectural and design guidelines for transforming a requirements specification based on
the proposed requirements analysis approach into a design document. Hence, by providing a technique for constructing a requirements specification from a problem statement and guidelines for transforming that specification into a detailed design, a high degree of traceability is achieved promoting automatability of code generation for embedded systems as well as promoting the reuse of specifications and thereby code.

The requirements analysis approach being proposed is independent of the implementation language. However, the architectural and design guidelines used to transform the specification into a detailed design use an Actor based concurrent object-oriented language prototype. The prototype used is an extended version of ACT++ [Kafura and Lee 89]. The extensions to the language were necessary in order to implement the proposed techniques for a representative process-control problem.

At this point, it would be prudent to investigate the efficacy of an Actor based concurrent object-oriented programming paradigm for embedded systems. We will investigate the suitability of the object-oriented message passing metaphor for the development of embedded systems, a discussion on how concurrency eases the software development task and explain why a model of computation is important in the context of embedded systems.

1.2 Concurrent Object-Oriented Programming

Software engineering principles such as abstraction and modularity have promoted the acceptance of an object-based programming paradigm to achieve highly cohesive and loosely coupled program components. Programming with objects offers the modularity necessary for developing large scale applications. The environment in which embedded systems operate is more readily modelled through the notion of objects. Object-oriented programming, couples the object-based approach with a resource sharing mechanism such
as inheritance, which promotes the reusability of code. Inheritance is a useful mechanism for capturing the commonality existing amongst the sensors and actuators that make up an embedded systems environment.

Concurrency refers to logical parallelism as opposed to physical parallelism. It provides the necessary flexibility to interleave the execution of components of a program on a single processor, or to distribute it among several processors. Concurrency abstracts away some of the details in an execution, allowing us to concentrate on conceptual issues without having to be concerned with a particular order of execution [Agha 90]. Modelling the asynchronous parallelism so evident in embedded systems through the notion of concurrency provides the necessary abstraction to analyze the problem at hand.

Broadly speaking, one of two models of concurrency could be imposed on a system of objects to render a concurrent object-oriented programming paradigm. First, there is the process based model of computation, in which process threads interleave their way through passive objects. These models use the more traditional approaches, such as shared variables and Hoare's monitors, for synchronization and concurrency control. Second, there is an actors based approach, in which the notion of a process is replaced by the concept of an active object (having its own thread of control) that communicates with other objects via message passing.

Combining the benefits of objects and concurrency should result in a programming paradigm powerful enough to handle the requirements of applications with distributed and real-time components. Several attempts have been made to combine these two technologies, and in fact, several languages exist today which combine one of the two "object" approaches with one of the two major approaches to concurrency. There also are languages that reflect variants of the two technologies. Figure 1.3 categorizes some of these languages.
The prototype ACT++ [Kafura 88], is a concurrent object-oriented language that combines the features of an object-oriented language, C++ [Stroustrup 86], with the concurrency semantics of the Actor model [Agha 86]. Other languages in this category are Actalk [Briot 89] and Rosette [Tomlinson and Scheveel 88]. Languages like Hybrid [Nierstrasz 87], POOL-T [America 87] and Concurrent Smalltalk [Yokote 86] have an object-orientation and are based more on the process-oriented approach to concurrency. Other languages that exploit the notion of active objects are ABCL [Yonezawa 87]

<table>
<thead>
<tr>
<th>Object-Based</th>
<th>Object-Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actors Based</strong></td>
<td><strong>Process-Based</strong></td>
</tr>
<tr>
<td>ABCL/1</td>
<td>Hybrid</td>
</tr>
<tr>
<td>Actra</td>
<td>POOL-T</td>
</tr>
<tr>
<td></td>
<td>Concurrent Smalltalk</td>
</tr>
<tr>
<td></td>
<td>Trellis/Owl</td>
</tr>
<tr>
<td></td>
<td>Actalk</td>
</tr>
<tr>
<td></td>
<td>Rosette</td>
</tr>
</tbody>
</table>

**Figure 1.3** - A survey of concurrent programming languages

and Actra. These languages, however, do not offer any resource sharing mechanism, such as inheritance, for promoting reusability of code. Languages based on the Actor model can be further categorized as shown in Figure 1.4.

As can be seen, several approaches have been taken to developing concurrent programming languages based on the notion of objects. Most of these language designs are motivated, in part, by the underlying architecture and application domain. The proposed language prototype ACT++ targets the embedded systems application domain.

The Actor model [Agha 86] is the selected model of computation for the language
prototype ACT++. Since an active object encapsulates both data and control, it more readily captures the parallelism inherent in distributed systems [Agha 90]. Message passing in the Actor model being asynchronous and buffered, more readily models the communication in embedded systems. It also helps establish common knowledge; if a

<table>
<thead>
<tr>
<th>Base Language</th>
<th>Sharing Mechanism</th>
<th>Object Interface</th>
<th>Application Domain</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACT++</td>
<td>C++</td>
<td>Inheritance</td>
<td>Replaceable</td>
<td>Embedded Systems</td>
</tr>
<tr>
<td>Actalk</td>
<td>Smalltalk</td>
<td>Inheritance</td>
<td>Replaceable</td>
<td>General Purpose</td>
</tr>
<tr>
<td>Rossette</td>
<td>Lisp</td>
<td>Inheritance</td>
<td>Replaceable</td>
<td>High Speed Computation</td>
</tr>
<tr>
<td>ABCL/1</td>
<td>New</td>
<td>None</td>
<td>Fixed</td>
<td>Distributed Programming</td>
</tr>
<tr>
<td>Actra</td>
<td>Smalltalk</td>
<td>Inheritance</td>
<td>Fixed</td>
<td>Real-Time</td>
</tr>
</tbody>
</table>

Figure 1.4 - A survey of Actor based languages

computation agent A communicates a message m synchronously with a computation agent B, A knows that B knows about m and B knows that A knows that B knows about m, etc., ad infinitum. In such a situation, A and B share common knowledge about the message m, a property fundamental to capturing the distributed nature of the communication in embedded systems. In the Actor model, actors can be dynamically bound, permitting actor systems to be reconfigurable and extensible, a property highly desirable for modelling the dynamism and redundancy inherent in embedded systems. Finally, guarantee of message delivery, nondeterminism in the arrival order of messages and interleaving of messages provide the necessary abstraction for reasoning about
concurrency, a conceptual basis for distributed systems.

1.3 Organization of Thesis

The remainder of this thesis is organized as follows. Chapter 2 discusses the requirements analysis approach for embedded systems. The requirements analysis technique is used to create a requirements specification for a representative process-control problem. This specification is included in Appendix B and is used to introduce and present the analysis technique. Chapter 3 discusses the Actor model, a model of concurrent computation for distributed systems as proposed by Agha [Agha 87]. Chapter 4 starts with a discussion on how the Actor model was adapted for a network of loosely coupled processors followed by a detailed discussion of ACT++. In the last section of this chapter we present the work done to make the first prototype more robust and functional followed by the extensions made to the ACT++ run-time system, rendering what this thesis calls the extended prototype. Chapter 5 contains the architectural and design guidelines for transforming the specification created using the proposed requirements analysis technique into an Actor’s based design. A part of the actual design that was created is included in Appendix D. Finally, Chapter 6 summarizes the work that was done and briefly talks about future research endeavors.
CHAPTER 2 - STARE

Embedded systems is a generic term that refers to a certain class of problems with similar attributes. Included in this class are command and control systems, flight guidance systems, industrial process control systems and data acquisition and control systems. Within embedded systems, the problem domain of interest in this thesis is industrial process control. This chapter focuses on the generic feedback control model that describes such systems and the proposed approach STARE, a requirements analysis technique for embedded systems that addresses the functional behavior of such systems.

2.1 The Process Control Model

Traditionally, process control has been defined as:

An arrangement of elements such as amplifiers, converters and human operators, interconnected in such a way as to maintain, or to affect in a prescribed manner, some physical quantity or condition of the process which forms a part of the system [Lowe 71].

Modern process control on the other hand is not restricted to just controlling the process, but is also concerned with the supervision and planning of the execution of the steps involved in controlling the process. Advancements in the areas of expert systems, distributed and reliable communications have led to computer controlled supervisory
systems that assist in the optimal and reliable functioning of modern process control systems.

![Diagram of a closed loop feedback control system](image)

**Figure 2.1** - Closed loop feedback control system

Despite these notable advances, still at the heart of any process control system is the control strategy based on classical feedback control system theory. This basic process control model is shown in Figure 2.1. The four components that are contained within this closed-loop are the process, the actuators, the sensors and the controller. The *functional* behavior of most process control systems can be modeled through this feedback control system.

The *process* (P) can be any mechanical, electrical or chemical system or combination thereof operating in a *controlled* environment. Typically, it is difficult to derive a mathematical model for the process due to it being highly nonlinear. The disturbances (D) that can affect the process over time render the process nonstationary as well. Due to the complex interactions within the process, system control is achieved by
controlling the individual subsystems as opposed to attempting to mathematical describe the process. An example of a process would be using boilers to generate steam for driving steam turbines.

*Sensors* (S) are used within the closed loop to monitor the condition of the process being controlled. For example, a temperature sensor within a steam generator would be needed to ensure that the generator was not subjected to excessive heat. The values sensed by the sensors are fed as inputs (I) to the controller.

*Actuators* (A) serve as the mechanism for realizing the changes to be made to the process. These would be any physical device that can affect the condition of the process being controlled, for example, water pumps and valves. The outputs (O) from the controller drive the actuators in order to control the process.

The *controller* (C) uses the inputs from the sensors and drives the actuators to control the process in the required manner. The disturbances that can affect the process cannot be manipulated and hence have no relevance to the controller. In addition to the inputs from the sensors, the controller's functionality is also influenced by the external interface (E), which typically is the operator console. Direct commands from the external interface can cause the controller to process the inputs from the sensors differently or to affect the actuators in a different manner.

The STARE approach being proposed provides a structured path for specifying the requirements for process control systems by exploiting the generic nature of this feedback control system. Before embarking on a detailed description of this approach it would be helpful to describe the representative process control problem that is being used to illustrate it.
2.2 Embedded Control of a Steam Generator

Figure 2.2 shows an embedded control system that monitors and regulates the steam generator under the control of a human operator. The control system must continuously display for the operator data reflecting the current operating condition of the generator. The operator may also issue commands to the control system. The control system must monitor the generator and regulate its condition to achieve safe, continuous operation. The steam generator itself is shown in Figure 2.3. Steam is produced by heating water in a large, closed tank. The resulting steam is vented from the top of the tank. The amount of water in the tank is measured by a water level indicator. Water may
be added to the tank by activating a water pump. The tank may be drained by opening a purge valve located at the bottom of the tank. The water is heated by burning natural gas.

![Diagram of steam generator]

**Figure 2.3 - Steam generator**

The amount of steam produced is controlled by regulating the flow of natural gas. At the top of the tank is a pressure relief valve which allows excess steam to be vented. The temperature and pressure inside the tank are measured by gauges attached to the tank. The
embedded control system is connected to the steam generator by sensors and actuators. The sensors allow the control system to determine the readings of the water level indicator, the temperature gauge and the pressure gauge. Two actuators allow the control system to open and close the pressure relief valve and the purge valve. Another actuator allows the control system to start and stop the water pump.

The embedded control system for controlling the steam generator conforms to the feedback control system described in the previous section. The inputs to the controller are the sensors measuring the temperature (T), the pressure (P), the water level (WL) and the gas level (GL) in the steam generator. The controller then processes these inputs based on the *state* of the generator and issues commands to drive the actuators, which in turn affect the process. The problem statement included in the appendix indicates that the Steam Generator can be in one of three defined states: Startup, Normal operation and Shutdown. How the actuators are to be manipulated and under what constraints, for each of the above states, is specified in the problem statement included in Appendix A. The next section provides a detailed description of the STARE requirements analysis approach using this representative process control problem for illustration purposes.

### 2.3 The STARE Approach

STARE, a requirements analysis approach for process control systems provides a structured path for transforming an informal definition of the functional behavior of process control systems into a precise and structured specification. Since the functional behavior is embodied in the controller component of the feedback control system, STARE provides guidelines for specifying this component in a precise manner.

The process being modelled, typically, is a continuous, nonlinear and nonstationary process for which it is difficult to derive a mathematical model. The external
disturbances D shown in Figure 2.1 arise from the environment in which the system is operating and could be in the form of a mechanical, electrical or chemical interaction. This interaction is a highly complex physical phenomenon that can be minimized but cannot be eliminated, with the consequence that the process is continually deviating from its required optimal behavior. The purpose of the actuators is to minimize this deviation. However, actuators being mechanical devices, tend to have a response time which is significant when compared to the continuous changes the process is going through. In addition, due to the thermodynamics of the process, the time it takes to see an appreciable change in the process after an actuator has been activated could also be substantial. Due to this time lag in the responsiveness of the system, the process as seen by the controller component no longer seems to be changing continuously, instead it seems to be changing in a discrete fashion. It is this discretization of the continuity of the process that brings about the concept of a state. This definition of a state, however, potentially has a very fine granularity of time.

In process control systems, the process is represented by a set of process variables. At any given point in time, the values of these process variables are sufficient to describe the process and the set of values together define the condition of the process. For example, in our representative problem the water level (WL), the temperature (T) and the pressure (P) are the three process variables. By imposing boundary values on the process variables a meaningful basis for defining a state is obtained. A state embodies the condition of a process for a certain period of time, with the process itself being viewed as a sequence of transitions from one state to another.

In STARE also, a process is viewed as a sequence of transitions from one state to another. However, the concept of a state as defined for process control systems in general, is somewhat extended. One can separate the events occurring in a state based on
whether they are dependent on the arrival of inputs from the sensors. As shown in Figure 2.1, the Controller (C) component of the feedback control system is certainly dependent on the arrival of data from the sensors. However, often the semantics of a state requires that a certain actuator's operational state remain invariant for the life of the state and is not required, in principle, to be dependent on the arrival of data from the sensors. The set of actuators whose operational state is to remain invariant for the life of the state constitutes the environment for that state. Hence, in addition to embodying the condition of the process, a state also embodies the environment for that state. The separation between the system state and the environment in which the system is operating also facilitates modelling exceptions, a feature that promotes completeness of the specification and thereby provides a greater degree of reliability. Therefore any definition of a state should account for the condition of the process as well as the condition of the actuators making up the environment for that state.

Figure 2.4 shows the structure of a state definition in STARE. Each state definition has the following two sections:

- Environment Specification
- Controller Specification

The Environment Specification section is used to specify the desired environment for the state whereas the Controller Specification section is used to describe the condition of the process and models the Controller component of the feedback loop shown in Figure 2.1. The Environment Specification section of the state definition embodies the invariant conditions of the actuators that make up the environment for that state. These invariants apply as soon as a transition is made into the state and remain in effect for the life of the state, i.e. until a transition occurs out of the state). For example, the state shown in
Figure 2.4 requires that the Pressure Relief Valve (PRV) and the Purge Valve (PV) remain closed for the life of the state.

<table>
<thead>
<tr>
<th>Environment Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Invariants</strong></td>
</tr>
<tr>
<td>PRV Closed, PV Closed</td>
</tr>
<tr>
<td><strong>Failure</strong></td>
</tr>
<tr>
<td>PRV opens =&gt; Report Failure</td>
</tr>
<tr>
<td>PV opens =&gt; State Transition (Startup_PV_Open)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Controller Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Water Level</strong></td>
</tr>
<tr>
<td><strong>Goal</strong></td>
</tr>
<tr>
<td>50 &lt; WL &lt; 85</td>
</tr>
<tr>
<td><strong>Policy</strong></td>
</tr>
<tr>
<td>50 &lt; WL &lt; 65 =&gt; Turn WP On</td>
</tr>
<tr>
<td>WL &gt; 65 =&gt; Turn WP Off</td>
</tr>
<tr>
<td><strong>Failure</strong></td>
</tr>
<tr>
<td>WL &gt; 85 =&gt; State Transition (Shutdown)</td>
</tr>
<tr>
<td>WL &lt; 50 =&gt; State Transition (Prime-WaterLevel)</td>
</tr>
</tbody>
</table>

| **Pressure**              |
| **Goal**                  |
| P < 450                   |
| **Failure**               |
| 450 < P < 500 => State Transition(StartUp_ExcessPressure) |
| P > 500 => State Transition (StartUpRelease_Steam) |

| **Temperature**           |
| **Goal**                  |
| 340 < T < 360             |
| **Policy**                |
| T < 350 => Apply gas at 80% level. |
| T > 350 => For every minute the temp is out of range adjust gas flow by 1% for each degree it is out of range. |
| **Failure**               |
| T > 375 => Alarm Message  |

Figure 2.4 - A state definition in STARE
The Controller Specification part of the state definition embodies the conditions of the process as opposed to the operational state of the actuators. It is used to specify the behavior of the Controller (C) component for the given state. Since the condition of the process is specified by assigning operating ranges for each of the process variables that are relevant to the current state, there is a separate subsection for each of the process variables. The subsection corresponding to a process variable is called a parameter subsection. Hence a Controller Specification is comprised solely of parameter subsections. For example, as shown in Figure 2.4, the Controller Specification has three parameter subsections corresponding to the Water Level, Pressure and Temperature process variables.

The structure of the Environment Specification section is shown in Figure 2.4 as well. The section consists of the Invariants block followed by an optional Constraints block and finally a required Failure block. This structure is motivated by the fact that when the system enters a certain state, the initial objective is to establish the desired environment. Thereafter the environment is continuously monitored for any changes. The Invariants block specifies the environment desired while the Constraints block, if any, describes the manner in which the desired environment is to be established and finally the Failure block monitors the environment for changes and specifies the action to be taken in the event that the environment does change.

As shown in Figure 2.4, the Controller Specification section has a Goal block, followed by an optional Policy block and finally a required Failure block. This partitioning is motivated by the generic feedback control system that is at the heart of any process control system. The Goal block is used to ascertain whether the process conditions for the state have been violated. In the event that the conditions for the current state are valid, the Policy block determines how the control is to be implemented. In the event that the conditions are not valid, the Failure block conveys what actions are to be taken.
subsequently. Included in the set of possible actions that can be taken from the Failure block is a state transition. Hence, only from the Failure block can a state transition be initiated.

So far we have provided only an overview of STARE. We have alluded to a process being represented through a sequence of transitions from one state to another. An attempt has been made to justify the chosen structure of a state and finally we have presented a high level view of the structure of a state itself. Before going any further, a review is presented of the states that resulted in applying STARE to the representative process control problem that is being used for illustrating this approach. A detailed analysis of these states is included in Appendix B in the form of the requirements specification document.

Figures 2.5 through 2.7 provide a synopsis of the possible states and the transitions between them. STARE distinguishes between two types of states, external states and internal states. External states are ones that are explicitly defined in the problem statement. For example, the problem statement specifies Startup, Normal and Shutdown modes of operation for the steam generator. Internal states on the other hand are states that have meaningful semantics associated with them and are created in order to adhere to the STARE approach. Figure 2.5 shows the internal states resulting from applying STARE to the external state Startup that had been specified in the problem statement. The external states have been highlighted in bold face. The very first state the system enters is the Startup state shown in Figure 2.5. In the event that there is insufficient water in the tank, the system enters the Prime_Water_Level state. Startup_Excess_Pressure is entered if there is an excess pressure build up in the steam generator. If the pressure build up is so severe that steam needs to be vented from the tank, the state Startup_Release_Steam is entered. The state Startup_PV_Open helps model an exception condition: the Purge Valve opening
during regular operation of the generator. On successful startup, the system moves into the Normal mode of operation. In the event any safety conditions are violated the system moves into the Shutdown state. Figure 2.7 shows the internal states that are reachable from the external state Normal Operation. These states are similar to the states shown for the Startup external state since the operating policies for Normal mode of operation also

Figure 2.5 - States reachable from external state startup
apply during Startup. Once the system enters Normal mode of operation there should be no need to prime the water tank. Figure 2.6 shows the internal states that are reachable

![Diagram of states](image1)

**Figure 2.6** - States reachable from external state shutdown

![Diagram of states](image2)

**Figure 2.7** - States reachable from external state normal
from the external state Shutdown. Shutdown requires that first the pressure drop to within acceptable limits and then the generator cool down to the desired temperature before it can be considered to be successfully shutdown. This sequence has been shown in Figure 2.6.

Before we delve into the internal details of a state definition in STARE, we review the guidelines used in identifying the states. These guidelines have been numbered G1 through G6 and are as follows:

**External State (G1)**

A prespecified mode of operation constitutes a valid state. Most process control systems define explicit modes of operation for the process, for example startup and shutdown. The requirements analysis phase commences by applying STARE to the initial external state the process is to be in. The semantics of the initial external state are first captured and additional internal states created in order to conform to the analysis technique. This process is repeated for all external states specified in the system. For example, we commenced our analysis by applying STARE to the startup mode of operation specified in the problem statement and created the internal states shown in Figure 2.5 in order to adhere to STARE.

**Process Condition State (G2)**

A state should be defined corresponding to an identifiable and significant process condition. Such states have to be identified by carefully reviewing the problem statement. Subsequent sections will provide detailed guidelines for identifying such states. For example the state Normal_Excess_Pressure shown in Figure 2.7 was defined to monitor the pressure between 450 and 500 psi.
**Sequence State (G3)**

Sequence states arise out of the need for implementing a sequence of operations. Each step in the sequence would be a valid state with a transition occurring to the next step in the sequence. The state Generator_Cool_Down in Figure 2.6 is a case in point.

**Hardware Exception State (G4)**

When an exception occurs, such as the failure of an actuator, it is usually accompanied by a change in the condition of the process as well. By introducing a new state, and having the Environment Specification section monitor the hardware exception and the Controller Specification section model the new process condition, structured partitioning of the problem statement is made possible. The state Normal_PV_Open shown in Figure 2.7 is entered if the purge valve were to suddenly open during regular operation of the generator.

**Conflict Resolution State (G5)**

In STARE a *conflict* refers to opposing actions being performed on an actuator by the different sections within a STARE state. Since the Environment Specification section of a state definition represents the environment for a state, it is to remain effective for the life of the state. It is possible that a desired operational state for an actuator in the Environment Specification section is conflicting with the action performed on the same actuator by the Controller Specification section. Such conflicts can be resolved by introducing additional states with a revised condition for the process. The details of such conflicts are explained more fully in the subsequent sections. For example, the states Normal_Excess_Pressure and Normal_Release_Steam shown in Figure 2.7 are a case in point.
External Interface State (G6)

For each command that is permissible from the external interface, there should be a corresponding state. In our representative problem the operator could either start the system or shut it down. Since these commands from the operator correspond to entering the StartUp and ShutDown states respectively, there was no need to define additional External Interface States.

In the next section we present the detailed structure of a STARE state.

2.4 Internal Structure of a STARE state

As pointed out earlier, a state definition has an Environment Specification section and a Controller Specification section. The internal details of the Environment Specification section are outlined below followed by a detailed description of the Controller Specification section. The state guidelines that were introduced earlier are also discussed in detail. The Hardware Exception state guideline being more relevant to the Environment Specification section is discussed there whereas Process Condition state, Sequence state and Conflict Resolution state guidelines are discussed in the Controller Specification section. The External state and External Interface state guidelines are not discussed further.

2.4.1 Environment Specification

The Environment Specification section of the state definition identifies the invariant conditions that hold within the state and has the structure shown in Figure 2.8. It is not, in principle, dependent on the arrival of stimuli from the environment in which the system is operating. This half of the state definition has an Invariants block followed by an optional Constraints block which in turn is followed by a required Failures block. Each of these blocks is discussed in detail below.
**Invariants**

The Invariants section is used to specify the invariant condition in terms of the desired operational state of the various actuators that are to remain invariant during the life of the state. The invariant operational state of an actuator that is desired is referred to as a sub-condition within the invariant condition. These actuators would be the ones that guarantee a consistent environment for that state. The definition of the invariant condition cannot contain any references to process variables since this section of the state definition is in principle independent of the arrival of stimuli from the environment, i.e. as soon as a transition is made into a state, the Environment section of the specification comes into effect and thereafter ensures that the invariant condition is always satisfied. *How* this condition is ensured is a design issue and one possible strategy will be presented when we
introduce the design approach used in transforming a STARE like requirements specification into a detailed design.

**Constraints**

It is through the Constraints block that constraints on the way the invariant condition is to be achieved can be specified. Typically, the constraints impose a sequential order on the sub-conditions within the invariant condition. Since this block is optional, in the absence of any sequencing constraints the sub-conditions could be achieved concurrently. As shown in Figure 2.8 the Constraints block ensures that the Gas Valve is turned off before the Pressure Relief Valve is opened. The Constraints block can affect only the actuators and no state transitions can be initiated from within this block.

**Failure**

The failure block is used to specify the actions to be taken in the event that the required invariants in the operational state of the actuators are not met. It is through failures that transitions can be made out of the state. For each sub-condition within the invariant condition, a corresponding failure should be specified. Requiring the designer to specify a failure for each sub-condition within the invariant condition ensures completeness of the specification since often the problem statement does not address what action should be taken in the event that an actuator fails while the process is in a certain state. Since the process moves from one state to another, by specifying the action to be taken in the event of an actuator failure for each state, we ensure the availability of a response irrespective of when the actuator fails. For example, in Figure 2.8, an action has been described for each possible sub-condition failure.
Hardware Exception State (Guideline G1)

When an exception occurs, such as the failure of an actuator, it is usually accompanied by a change in the condition of the process as well. By introducing a new state, with the Environment Specification section monitoring the hardware exception and the Controller Specification section modelling the new process conditions, structured partitioning of the problem statement is made possible. For example, the Purge Valve section of the excerpt shown in Figure 2.9 details what needs to be done in the event the Purge Valve were to open during regular operation of the generator. The fact that the pressure has to be checked \((P < 125 \text{ psi})\) before the generator can be shut down indicates the change in the condition of the process that has to be addressed within the Controller Specification of the new state.

2.4.2 Controller Specification

The Controller Specification part of the state definition models the semantics bound to the controller in the feedback control system model shown in Figure 2.1. The sensors provide inputs to the controller \((C)\) which are processed to produce the outputs that drive the actuators. Within the Controller Specification there is a section for each process variable being monitored by the sensors which is referred to as the parameter subsection. A typical state would have a parameter subsection for each process variable that can describe the condition of the process. Having a parameter subsection for each process variable ensures completeness of the specification as well.

2.4.2.1 Parameter Subsection

A Controller Specification is a collection of parameter subsections. The central idea behind a parameter section is to model a \textit{single process} variable. Since typically the process being controlled is nonlinear and nonstationary, to derive a mathematical model of
Startup:

To initiate the steam generator the water level must be at least 50% before the gas is turned on.

Normal Operation:

Pressure Gauge

If the pressure exceeds 500 psi the pressure relief valve must be opened, the gas turned off and an alarm message displayed for the operator. When the pressure falls below 480 psi the generator operation should return to normal. The water pump must not be operated if the pressure is above 450 psi.

Purge Valve

If at any time the gas is on or the boiler pressure is above 125 psi and this valve is sensed open, the pressure relief valve must be opened and the gas shut off.

Shutdown:

- when the pressure drops below 50 psi open the purge valve

When the pressure drops to 0 psi and temperature is below 100 degrees Fahrenheit the generator is considered to be in a safe condition and has been shut down.

Figure 2.9 - An excerpt from the problem statement

the process is a nontrivial task. Therefore the problem statement itself describes in detail how individual process variables are to be manipulated. Each parameter subsection requires a goal section, followed possibly by an optional policy section and finally a required failure section.

Goal

The goal associated with the parameter subsection specifies he goal predicate in terms of the relevant process variable. Since the inputs into the controller are only sensor
values, the predicate expression can only involve sensor values. It is important to have a well defined goal since most conflicts arise due to goals that have been defined in a loose fashion (see the details of the state guidelines below). When identifying the goal for a parameter subsection, the following should be kept in mind:

- Goal definition should involve only sensor values and should make no reference to the operational state of an actuator. Situations where it is absolutely imperative that actuator information be available, the introduction of a new state can alleviate the problem since the actuator information can be modelled through the Environmental Specification section of the state definition.

- Typically a goal would be the normal operating range for a parameter for a given state. Care should be taken when identifying the goal from its policy.

- Since goals affect the failures section, and since transitions can only be made in the event of a failure, goal definitions affect how the state transitions take place.

- A successful goal would result in the activation of the policy section for that goal whereas a goal failure would trigger the failure section.

**Policy**

Associated with each goal is an optional control policy that should be used to achieve the goal. It is not necessary to have a policy section. The control policy is the mechanism for realizing the goal. The separation of the goal from its control policy promotes reusability of the specification as well as helping in further structuring the specification. The following should be kept in mind when identifying the control policy:

- Only the control policy can manipulate the actuators.
• The control policy can have no affect on the occurrence of a failure.
• There should be no conflicts between the control policies of the different parameter subsections for that state.
• There should be no conflicts between the control policy of any parameter subsection and the Invariant block of the Environment Specification half of the state definition.

Failure

The failure part of the parameter subsection is activated when the corresponding goal cannot be met. There are two possible actions associated with a failure:
• Reporting
• State Transition

The action associated with a failure cannot change the operational state of an actuator. If such a response were desired, a new state could be introduced, the invariant section of which would ensure that the desired operational state of the actuator was realized. There should be a failure action identified for each possible reason why the goal could not be met. This ensures the completeness of the specification. In the representative process control problem used as a testbed for this approach, often the problem statement did not address the actions to be taken corresponding to the different failures that could occur. Identifying such omissions is an important part of the requirements analysis phase.

Process Condition States (Guideline G2)

When attempting to associate a state with a certain process condition, the condition selected should be minimal, i.e. no possible transition points should be hidden because of the overly broad manner in which the condition has been specified. An overly broad goal
definition could subsume several state transitions since there is a failure action associated with each reason for the failure and the fact that a state transition is a possible action in the event of a failure. For example, a preliminary examination of a part of the problem statement shown in Figure 2.9 (the Pressure section) reveals that the goal for the pressure process variable could have been \((P < 500 \text{ Psi})\). However, at 450 and 480 Psi the behavior of the process changes indicating the need for a tighter goal definition. This guideline typically works in conjunction with the guideline for Conflict Resolution states.

**Sequence States (Guideline G3)**

Process Control systems often require actions to be performed in sequence. Each individual action within the sequence would constitute a valid state with the transitions providing the desired sequentiality. These states would not have any reverse transitions that are characteristic of say the Process states. For example, as shown in Figure 2.9, the problem statement specifies a sequence of operations to be performed during Shutdown mode. This sequence has been shown in Figure 2.6.

**Conflict Resolution States (Guideline G5)**

These guidelines work in conjunction with the guidelines for identifying process states. A conflict refers to opposing actions being performed on an actuator either by a policy subsection of a parameter subsection and the Invariant block of the Environment Specification section or else between the policy subsections of two parameter subsections. For example, a preliminary examination of a part of the problem statement shown in Figure 2.9 (the Pressure section), reveals that if the goal had been specified as \((P < 500 \text{ psi})\) there would have been a *conflict*. The policy within the Water Level (WL) parameter subsection might be attempting to turn the pump on while the policy within the Pressure (P)
parameter subsection would be required to turn the pump off if the pressure were to exceed 450 psi. The conflict can be eliminated by defining the goal as \( P < 450 \) and introducing the state Normal_Excess_Pressure to handle the range \( 450 < P < 500 \) and the state Normal_Release_Steam to handle the range \( P > 500 \). Not only is the conflict eliminated but a more semantically meaningful structuring of the problem statement is created. Hence an appropriate goal definition is a key element in STARE for eliminating conflicts as well as imparting meaningful semantics to the states identified in the system.

2.5 State Transitions

An attempt has been made to describe the features of STARE in a fair amount of detail. However, an important issue has not been addressed, transitions between states. As has been pointed out earlier, the control process is viewed as a sequence of transitions between states. These transitions can be initiated from the failure block of the Environment Specification section or from the Failure block of a parameter subsection. Since STARE does not address how the different sections within a state definition are to be transformed into a design the following question is raised: In the event that the different sections are being realized concurrently, how do we arbitrate between transitions which occur simultaneously in the environment? For example, what if there were excessive water in the tank and the pressure had exceeded its safe operating limit?

STARE does not arbitrate between state transitions and instead achieves the desired consistency for the state being entered through appropriate parameter subsections and goal definitions therein. For example, in Figure 2.5 from the external state Startup, transitions can be caused from within the Failure block of the Environment Specification (PV Open) and from the Failure blocks of the Water Level and Pressure parameter subsections. One of the transitions from the Startup state (WL < 50) originates from the Failure block of the
Water Level parameter subsection and leads to the state Prime_Water_Level. Though the primary function being performed by the state Prime_Water_Level is to fill up the water tank, in order to avoid arbitrating between transitions that could occur simultaneously, the state Prime_Water_Level must also have parameter subsections to monitor the pressure and temperature for any exception conditions occurring since these were being monitored in the state Startup, thus ensuring the desired consistency.

Therefore when using STARE it is imperative that no assumptions be made about the order in which transitions could occur. Parameter subsections and goals therein should be defined in such a way that when a transition is made, the state being entered should not assume which section from within the departing state the transition originated from.

We have described a mechanism to obviate the need for arbitrating between transitions that could occur simultaneously. However, we have not provided any support for ensuring that the proposed mechanism is being realized. Heuristics are needed to verify that transitions are taking place in a consistent manner. Such heuristics are being investigated.

2.6 Summary

STARE is an approach for decomposing the requirements for process control systems into a structured specification. No attempt has been made to create another specification language. The work presented here could serve as a front end to several specification languages that have been developed and used for the specification of process control systems [Steusloff 84].

Additional work needs to be done on the issue of verification. Formal representation techniques need to be studied that would facilitate the representation of the functional behavior within the proposed structure so as to enable automatic verification and
facilitate running consistency checks. In addition, the manner in which the performance requirements of embedded systems can be integrated into such an approach also needs to be investigated.

A structured specification technique such as the one being proposed would certainly promote the reusability of specifications. The proposed specification technique would work well with reusability approaches like DRACO [Neighbors 90].

Although the STARE approach is independent of the underlying implementation environment, the most elegant way of transforming a given specification into the corresponding design would be to use the concurrent programming paradigm. The next chapter describes in detail the model of computation on which the implementation language is based, the chapter following that explains in detail the implementation language itself. STARE together with the design environment that has been proposed in Chapter 5 provides a high level of traceability and shows some promise of automating the software development process for process control systems.
CHAPTER 3 - ACTOR BASED LANGUAGES

This chapter introduces the Actor model, a message passing model of concurrent computation, originally proposed by Carl Hewitt and later extended by Agha [Agha 86].

3.1 The ACTOR model

The Actor model is a model of computation based on the concept of active objects in which the behavior of an object is a function of the incoming communication and its encapsulated state. The only active entity in the Actor model, an actor, is a computational agent that is identified by the address of the mail queue assigned to it at the time the actor is created. Figure 3.1 shows the abstract representation of an actor. The figure shows a mail queue with the buffered messages and the nth behavior processing the nth message. An actor can do one of three things as shown in Figure 3.2.

First, an actor can create a new actor. This is referred to as the "Create" primitive in the Actor model. The ability to dynamically create new actors promotes reconfigurability and extensibility. A reconfigurable system is one in which an actor is able to communicate with actors not known to it at the time it was created. Extensibility on the other hand, allows a system to dynamically allocate resources to a problem by generating computational agents in response to the magnitude of a computation required to solve a problem [Agha 86].
Second, an actor can communicate with another actor by sending a message to the receiving actor's mail queue address. This is referred to as the "Sendto" primitive in the Actor model. Communications in the Actor model are asynchronous and buffered, with a guarantee of message delivery. An asynchronous, buffered message passing model with a guarantee that messages will be delivered is critical to establishing common knowledge in a distributed system. The underlying mail system in the Actor model guarantees delivery of messages sent to an Actor but the arrival order of messages is non-deterministic. Such properties provide the necessary abstraction to reason about concurrent systems. They also mesh well with the notion of a lack of a global order in distributed systems.

Third, an actor can specify a replacement behavior, which is the mechanism for determining how and when the next communication is to be processed. In the Actor model this is referred to as the "Become" primitive. The replacement behavior affects only how the actor will process the next communication in its mail queue, the mail queue address of an actor, however, is invariant. The notion of a replacement behavior is the most novel concept in the Actor model. It not only serves as a synchronization mechanism but also serves to achieve an inherently more concurrent system. A behavior is defined by the

Figure 3.1 - An abstract representation of an actor
actors that make up the environment in which it executes, and a "behavior script" that describes how to process the next message from the actor's mail queue. The concept of a replacement behavior helps eliminate sequential bottlenecks caused by assignment to a store and also promotes concurrency within the actor itself since several behaviors of an actor could be executing concurrently. Note that the semantics of replacement is fundamentally different from changes to a local store, replacements can exist concurrently
The three main primitives in the Actor model are powerful enough to support more abstract operations and patterns of concurrent problem solving. An actor operates according to a simple execution cycle. An actor is created with an initial "current" behavior. A message when available, is removed from the mail queue and bound to the "current" behavior. Each behavior processes exactly one message and each message is bound to only one behavior. While the current behavior of an actor is executing, all communications that arrive at that actor's mail queue are buffered. When the currently executing behavior specifies a replacement through the become operation, the next message in the mail queue is bound to the replacement behavior. If there are no messages pending in the mail queue, the replacement behavior awaits the arrival of a message. In other words, an actor could have several behaviors executing at the same time. The cycle of binding an incoming communication to the current behavior of an actor is repeated until the actor voluntarily terminates by explicitly nominating a null replacement behavior or until the actor is garbage collected [Kafura 88]. Two different kinds of actors are distinguished based on where in its script it nominates a replacement behavior. A serialized actor is one that nominates a replacement behavior as the very last event in its script, hence the name serialized. An unserialized actor on the other hand, can process its messages concurrently by nominating a replacement behavior for processing the next message after the “effect” of the current communication has been realized.

The Actor model defines an actor's environment through the concept of acquaintances. The acquaintances of a behavior are those actors whose mail addresses are known to the behavior. Acquaintances play two roles in defining a behavior's execution environment. First, they function either as providers of a service or as recipients of a service. Second, they comprise the state information associated with a behavior's
execution environment. For example, an actor that models a bank account would have as an acquaintance an actor representing the account's current balance. Acquaintances are established either statically or dynamically. The naming mechanisms used to identify acquaintances statically, i.e. at compile time, will depend on the language definition. Actors whose mail queue addresses are passed as part of a mail message are used to establish acquaintances dynamically [Kafura 88]. Dynamically acquired acquaintances are used to form continuations and customers. A customer is an actor that is the recipient of the results of a computation. A continuation on the other hand is implemented using customers, and essentially specifies how the execution of the rest of the script should proceed after evaluation of some expression involving another actor whose response must be awaited. Another specialized actor is a receptionist, an actor to whom communications may be sent from outside the configuration to which it belongs. The set of receptionists evolves dynamically as the mail addresses of various actors may be communicated to actors outside the system.

Concurrency in the Actor model stems from three sources. First, there may be several actors executing concurrently. Second, in the case of an unserialized actor, multiple behaviors of the same actor could be executing concurrently. Finally, continuations using customer actors can produce a pipeline effect, where each step in the processing pipeline performs a part of the computation and passes the partial results on to the next step as a continuation [Kafura 88].

3.2 Summary

The Actor model is an asynchronous buffered message passing model of concurrent computation in distributed systems that attempts to maximize parallel execution by eliminating assignments to a local store through the concept of a replacement behavior. It
provides three powerful primitives, namely the "Create", "Sendto" and the "Become" operations. The invariant part of an actor is its mail queue, which has a unique address assigned to it when the actor is created and is the target in any communication sent to it.

The Actor model is ideally suited for highly to massively parallel architectures involving perhaps thousands of processors with their own local memories and connected by a very high speed message-passing based interconnection network. Such architectures are being studied. For example the Mosaic project at Caltech involves an experimental prototype consisting of 16,384 nodes and is expected to deliver 290,000 MIPS [Atas 88].
CHAPTER 4 - EXTENDED ACT++ - DESIGN ISSUES

As detailed in the previous chapter, the Actor model is a fine-grain model which maximizes concurrency by eliminating sequential bottlenecks resulting from assignments to a local store through the concept of a replacement behavior. The Actor model is ideally suited for multicomputer architectures. However, this level of granularity is not practical for a network of loosely coupled processors. In the next section we discuss how the level of granularity, message passing and message delivery concepts differ in the primitive Actor model and the model of computation on which ACT++ is based.

4.1 Adapting the Actor Model

The fine-grain concurrency inherent in the primitive Actor model is not practical for loosely coupled architectures, especially in satisfying performance requirements. Therefore, an approach similar to the one used in ABCL/1 [Yonezawa 87] was adopted for ACT++. Each actor is a procedure-sized object and is termed an ultralight process. The internal operation of an ultralight process is defined by traditional imperative code augmented by actor operations (Create, Sendto and Become) [Kafura 88].

A Continuation in the primitive Actor model is a mechanism for pipelining concurrency and is used extensively to model higher level abstractions. However, for our targeted architecture this would result in the dispersion of related objects, causing potential
performance problems and undermining the modular application structure. In our adaptation of the Actor model, a continuation is used to relegate less related code to another actor. In the primitive Actor model, the request-reply type blocking message passing style is simulated through the concept of an insensitive actor. An actor, A, becomes insensitive by assuming a behavior which forwards all incoming messages to an auxiliary buffer actor. This forwarding continues until the message arrives which contains the results of the initiated operation. At this time, A sends the buffer a message directing the buffer to retransmit all queued messages back to A. In the adapted model of computation, a more efficient mechanism for "waiting by necessity" [Caromel 88] is implemented, and is named after the Cbox structure of Concurrent Smalltalk [Yokote 86]. A Cbox is a special mailbox that is created to receive the response of a message. It is different from a future [Lieberman 86], as it can be reused for replies to more than one request.

4.2 Design of ACT++

Since the long-term objective of this research is to test the efficacy of concurrent object-oriented computing and reusability of code for developing embedded systems applications, the base language chosen for ACT++ was C++ [Stroustrup 86]. This was motivated in part, by the strong typing and a powerful inheritance mechanism offered by C++. Since a quick evaluation of this technology for the targeted application domain was our first objective, developing a rapid prototype was deemed the best alternative. The powerful inheritance mechanism offered by C++ facilitated the development of an inheritance hierarchy that captured the concurrency abstraction of the adapted Actor model, with no changes to the base language. Moreover, working within the framework of an inheritance hierarchy would facilitate making modifications to the design as they evolved. The next few sections detail the conceptual issues of how this abstraction was achieved in
the first prototype of the language.

Objects

There are two kinds of objects in ACT++, namely, active objects and private passive objects. Active objects have their own thread of control, which can weave its way through the passive objects. As explained in the introduction, this is a hybrid model of computation. However, ACT++ adopts the convention that the reference to the passive objects cannot be exported to other actors, thereby obviating the need for explicit synchronization control. Put differently, only one thread of control can potentially weave its way through a passive object.

Class

An object is an instance of some class. A class mechanism enables objects with similar attributes to be grouped together. Through the inheritance mechanism, a subclass can redefine, restrict or extend the definition of its superclass (the class it is derived from). A special class called ACTOR was provided for capturing the semantics associated with an active object. Hence, all active objects are defined within an inheritance hierarchy rooted in the ACTOR class. Passive objects on the other hand are instances of a C++ class not derived from ACTOR.

Mail System

In the adapted Actor model, on which ACT++ is based, the underlying mail system supports two types of messages and a corresponding mail box for each message. A request message targeted for a actor is delivered to the mail box that identifies that actor. The mail box to which request messages are sent is called a Mbox. A Mbox models the
actors mail queue and is invariant for the life of an actor. The mail box that receives reply messages on the other hand is called a Cbox and allows for programming in a manner similar to futures [Lieberman 87].

4.3 ACT++ Class hierarchy

Figure 4.1 depicts the inheritance hierarchy that provides the necessary Actor

![ACT++ Class Hierarchy Diagram]

Figure 4.1 - ACT++ class hierarchy
abstraction. Only the Mbox, Cbox and the Actor classes are visible to the ACT++ programmer. The dashed line in Figure 3.1 separates the classes comprising the ACT++ run-time system (RTS) from the classes that need to be visible to an ACT++ programmer.

### 4.4 ACT++ Operations

Figure 4.2 summarizes the operations that are provided in ACT++. Each of these operations is discussed below.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Class</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>none</td>
<td>Create a new actor</td>
</tr>
<tr>
<td>become</td>
<td>ACTOR</td>
<td>Specify a replacement</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Mbox, Cbox</td>
<td>Send a message</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Mbox, Cbox</td>
<td>Receive from a Cbox or Self</td>
</tr>
<tr>
<td>reply</td>
<td>Cbox</td>
<td>Send a reply message</td>
</tr>
<tr>
<td>in</td>
<td>Cbox</td>
<td>Check if a Cbox has a reply</td>
</tr>
</tbody>
</table>

**Figure 4.2 - Summary of ACT++ operations**

The **New** operation takes the name of an Actor class as an argument. It creates an Actor and returns the mail queue id or specifically, a reference to an Mbox. For example:

```c
Class Recursive_Factorial_Actor_Class;

Mbox Recursive_Factorial_Actor = New(Recursive_Factorial_Actor_Class);
```

can be used to create an actor for computing factorials that will be identified by the
“Recursive_Factorial_Actor” mail queue id.
The **become** operation is a member of the ACTOR class and is overloaded to allow an actor to either specify itself as the replacement behavior or nominate a different behavior. For instance:

```
become(self);
```

would be the way for an actor to nominate itself as the replacement behavior for processing the next message. In the event that a different behavior had to be nominated as a replacement the following would need to be done, where "Some_Other_Actor" would be a class derived from the Actor class:

```
become(New (Some_Other_Actor));
```

The **send** operation is implemented as an overloading of the stream operator, `<<`, in the class Mbox. This operator is also overloaded in the Cbox class, to be used by the **reply** operation in dispatching results to a Cbox. For example, sending a message to an Mbox “Factorial_Actor” asking it to “Compute” the factorial of 20 and dispatch the result to a Cbox “MyCbox” is accomplished by:

```
Cbox MyCbox;
Factorial_Actor << Factorial_Actor_Class::Compute << MyCbox << 20;
```

The **receive** primitive is implemented as the overloaded stream operator `>>` in both the Mbox and the Cbox classes since acquaintances can be received from a mail message and the result of a request message can be received from a Cbox. In the Mbox class this overloading permits the current behavior to extract the next message in the Actor’s Mbox. The method name passed in the mail message is not extracted. The method name argument
in the mail message is used by the ACT++ run-time system to select the method to invoke. Only the arguments for the invoked method are extracted through this operator. Hence, every method of a behavior script needs to extract its arguments from the mail message as shown below:

```c++
int factorial_to_compute;
self >> factorial_to_compute;
```

A Cbox that might have been part of the mail message is not extracted since the ACT++ RTS automatically stores it away as the reply point. This reply point is used by the reply primitive in dispatching the results to the requesting actor. This mechanism is referred to as `reply forwarding`.

An actor can block itself pending arrival of the results of an earlier request (in which the Cbox “MyCbox” had been specified as the reply point) through the following structure:

```c++
int Result;
myCbox >> Result;
```

The `reply` operation takes only the result to be dispatched as an argument, as shown below. It utilizes the overloading of the stream operator `<<` in the Cbox class to dispatch results `transparently` to the requesting actor. This transparency is achieved through the reply forwarding mechanism. When a Cbox is sent via a mail message to another actor, the ACT++ RTS stores the Cbox away as the reply point, which can then be used to dispatch results to the requesting actor in a transparent way.
4.5 Interfacing Actors to a Non-Actor World

As mentioned in the introduction, one of the objectives of this research was to evaluate the effectiveness of concurrent object-oriented programming for developing embedded systems applications. A preliminary structural evaluation of the language prototype revealed there was a problem interfacing an Actor application with the underlying operating system. We refer to this problem as one of interfacing an Actor world to a non-Actor world.

Traditional operating systems have a process oriented view of concurrency, with concurrency control mechanisms being incorporated into the operating system kernel. In such a world view, an actor application, even though comprised of numerous actors, is viewed as a single process by the underlying operating system. Figure 4.3 depicts the two world views.

Running actor applications under traditional operating systems, such as UNIX, would require interfacing the concurrency semantics of the Actor model, based on active objects, with the traditional process oriented concurrency semantics of the underlying operating system. To access system resources, a process has to request the services of the operating system. In the hybrid world of Actors and non-Actors, such a system request, though originating from an Actor, is viewed as coming from the process that subsumes the Actor application. The proposed concept of interface actors bridges the difference between the two world views. An interface actor is a specialized actor that permits the underlying operating system to view an actor requesting a service as an independent entity,
even though it is subsumed within a process.

In Figure 4.4 we show the design strategy on which the concept of interface actors has been based. In the absence of interface actors, when an actor in ACT++ would request an I/O service from the operating system, the operating system would view an I/O request as coming from the process instead. If the request could not be satisfied immediately, the underlying kernel would block the whole process, instead of just the actor making the request. In order to prevent the whole application from being blocked, when an actor requires I/O services from the underlying operating system, it communicates with an interface actor, instead of communicating directly with the operating system. As shown in Figure 4.4, the requesting actor sends a message to the interface actor requesting that the I/O be performed (1). The requesting actor passes a Cbox as part of the request.
message and then blocks on the Cbox awaiting the results. On receiving the message, the interface actor *repackages* the I/O request and invokes the Asynchronous I/O Manager in the extended ACT++ run-time system (RTS) (2). The Asynchronous I/O manager first attempts to satisfy the request immediately without blocking itself (3) (Unix Non-blocking I/O). In the event that the descriptor was *not ready* for I/O, the Asynchronous I/O manager sets up the I/O operation to be performed *asynchronously* using the asynchronous I/O facilities of UNIX (4). Finally, when the descriptor does become ready for I/O, UNIX interrupts the Actor application (5), at which point the Asynchronous I/O manager performs the I/O and dispatches the results to the Cbox on which the requesting actor was blocked (6). Interface actors have been provided for handling terminal and socket I/O.
The C++ inheritance mechanism was used to provide the necessary abstraction and the inheritance hierarchy is depicted in Figure 4.5.

Even though the current language prototype ACT++ realizes pseudo-concurrency, the problem of interfacing an Actor world to a non-Actor world was observed in the design of another prototype of ACT++ running on a shared memory multiprocessor and realizing physical concurrency. The underlying operating system still views system service requests as coming from the process that subsumes the different threads, as opposed to the thread that actually issued the request.

Given a complete actor machine, such interfaces to the underlying operating system will not be necessary. However, such machines will be based on highly to massively
parallel architectures, involving thousands of processors connected by a high speed message passing network, an architecture contrary to the objectives of this research. A distributed operating platform could obviate the need for such interface actors as well. However, these platforms themselves are prototypes, relying on the underlying operation system for I/O services. Hence, we feel such interfaces are necessary, at least while the effectiveness of the language for developing embedded systems is being established.

4.6 Summary

ACT++ combines the benefits of C++ with the concurrency semantics of the Actor model and is well suited for developing applications based on the concurrent object-oriented programming paradigm. Extended ACT++ is a more robust version of ACT++ and also provides a mechanism for interfacing the concurrency semantics associated with active objects with the more process oriented semantics of the underlying operating system.

In the next chapter we present the design approach that transforms a STARE-based requirements specification for process control systems into an actor-based design.
CHAPTER 5 - DREAMS

In Chapter 2 we presented STARE, a requirements analysis approach for process control systems. As pointed out earlier, STARE is independent of any underlying implementation strategy. A STARE state is highly structured and consists of the Environment Specification section and one or more parameter subsections making up the Controller Specification. An object-oriented approach not only readily models the actuators and sensors in any process control system but it also captures the structured nature of a STARE state in an elegant fashion. More importantly, since a STARE state is free of any conflicts between the various subsections and since the Environment Specification section constitutes an invariant environment for the state, an approach that exploits concurrency as well is very suitable for transforming a STARE based specification into a detailed design. This chapter focuses on DREAMS, a design approach for STARE based requirements analysis. DREAMS provides architectural guidelines for transforming a STARE specification into a high level design that exploits concurrency and uses the primitives of the concurrent object-oriented programming language ACT++ introduced in the previous chapter for transforming an architectural view of the requirements specification into a detailed design.
5.1 The DREAMS architecture

The DREAMS architecture is motivated by the structure of the feedback control loop shown in Figure 2.1 and the structure of a STARE state. Figure 5.1 depicts the DREAMS architecture with the arrows showing the direction of the communication between the connected blocks. Using the feedback control loop as a basis, the boundary of the architecture is defined in terms of the sensors (S), the actuators (A) and the external interface (EI). Since the Controller Specification half of a STARE state models the Controller component in the feedback control loop, the arrows depicting the communication of the Controller component with the boundaries of the system are similar to the ones shown in Figure 2.1 (1,2,4,6). Reverse communication from the Actuators stub to the Controller Specification stub (5) is motivated by the possibility of actuator

![Diagram of DREAMS architecture](image)

Figure 5.1 - The DREAMS architecture
malfucntions as is the communication arrow from the Actuators stub to the Environment Specification stub (7). A bidirectional communication path exists between the Controller Specification stub and the Environment Specification stub (8,9). Forward communication (9) is necessary since the External Interface can command the Controller to move into a different state, a fact that would have to be communicated to the Environment Specification stub as well. Since the Environment Specification stub constitutes the environment for a state, the reverse communication path (8) is necessary to communicate to the Controller Specification block any changes in its environment. Communication path (3) will be used to update the Operator Consoles with the current condition of the process.

Before going further, let us view the concurrency that exists in the architecture shown in Figure 5.1. The stubs shown in Figure 5.1 execute concurrently with each other. While the sensors are feeding data into the Controller Specification stub the Environment Specification stub is monitoring the environment associated with the state that the Controller Specification currently represents. Even while the sensors are sampling the next set of data values the Controller Specification stub assesses how the process variables should be manipulated and directs the Actuators accordingly. The External Interface can command the Controller Specification to change state any time, while the Operator console would be continuously updated with the condition of the process. In the overall architecture presented in Figure 5.1, DREAMS does not address the internal architectural details of the boundary of the system, which is made up of the Sensors, the Actuators and the External Interface stubs. DREAMS only requires "black box" functionality from the boundary of the system. We will now examine the architectural details of the Controller Specification stub and the Environment Specification stubs.

The architecture of the Controller Specification is shown in Figure 5.2. Two states S1 and S2 are shown in the figure to indicate the similar architecture across all Controller
Specifications. However, the discussion below applies to state S1 only. A parameter subsection is shown contained in a vertical dotted rectangle. In the sample figure there are two parameter subsections for state S1 of which only one is shown enclosed in a dotted vertical rectangle. The Failure sections of all the parameter subsections (S1F1, S1F2) are grouped together into one stub so as to provide one source from where the transitions can originate. A horizontal dotted rectangle encloses the Failure blocks (S1F1, S1F2) of all parameter subsections within a state and is referred to as the Collective Failures stub. The

\textbf{Figure 5.2} - Architecture of the controller specification
Goal (S1G1) of each parameter subsection communicates with the Collective Failures stub and the corresponding Policy stub (S1P1). Each Policy stub, in turn, communicates with the Actuators stub. Once again, let us view the concurrency that exists within the architecture of the Controller Specification. Each Parameter subsection operates concurrently with other Parameter subsections and also with the Collective Failures stub. Within a Parameter subsection the Goal and Policy stubs operate concurrently as well.

Figure 5.3 shows the architecture of the Environment Specification component. Two states S1 and S2 have been shown in the figure to indicate the similar architecture across all Environment Specification halves of a state definition. However, the discussion below again applies only to state S1. As explained in STARE, the goal predicate can be composed of several sub-goals. The multiple sub-goals shown in Figure 5.3 (S1G1, S1G2) communicate with the Failure stub (S1F) and the Policy stub (S1P).

**Figure 5.3 - Architecture of the environment specification**
There is only one Policy stub, as opposed to each sub-goal having a corresponding policy stub, since the Policy within the Environment Specification section plays the role of a serializer. Sub-goals that cannot be achieved concurrently are serialized through the policy section. There is only one Failure stub, once again to provide a single source from where the transitions can originate. Once again, the Goal, the Policy and the Failure stubs all operate concurrently.

Before presenting the detailed design guidelines for each stub shown in the architecture, let us view the efficacy of the extended ACT++ language prototype in transforming a high level design, based on the architecture described, into a detailed design. We will look at the correspondence between a stub in the architecture and an actor in extended ACT++, how a communication can be represented as a message to an actor, how a replacement behavior can elegantly model a state transition and how dynamic connectivity can be achieved through the concept of acquaintances.

5.2 The Role of Extended ACT++ in DREAMS

The extended ACT++ language prototype is a concurrent object-oriented language that derives its concurrency semantics from the ACTOR model, a model of concurrent computation for distributed systems. There is an interesting relationship between the key elements in the DREAMS approach and the language primitives provided within extended ACT++.

An actor, being an active object, readily models a stub in the DREAMS architecture. The independent thread of control activating each actor models the concurrent execution of stubs in the architecture. The object orientation imparted by the language to an actor results in loosely coupled and highly cohesive program components. In the Actor model, actors communicate with one another by sending asynchronous messages that are
buffered in the receiving actors mail queue. Asynchronous message passing is the underlying communication paradigm between the different hardware elements that make up a process control system.

In STARE, the continuity of the process was achieved through discrete transitions between states. The partitioning of a state into the Controller Specification and the Environment Specification is modelled by corresponding blocks within the DREAMS architecture. With the Controller Specification and the Environment Specification together representing a state, Figure 5.1 can be viewed as the connectivity between a state and the boundaries of the system. In order to elegantly capture the notion of a state transition, we need a mechanism that can achieve the necessary transitions without violating this connectivity. The Actor model provides such a mechanism in the replacement behavior. In the primitive Actor model the concept of a replacement behavior is used primarily as a synchronization mechanism to achieve greater intra-object concurrency and also as a substitute for assignment to a local store. In DREAMS the replacement behavior is viewed in a somewhat different light. If we view an actor as an agent that provides a service, with the mail queue serving as the access point for that service and the current behavior as the provider of that service, a behavior nominated as a replacement can be viewed as an alternative mechanism for providing the same or different service. It is through this dynamic relationship between the mail queue of an actor and its current behavior that we can achieve state transitions without violating the connectivity being proposed in DREAMS.

In addition to elegantly modelling the notion of a state transition, the semantics of a replacement behavior are also relevant to the strategy used in process control systems for achieving greater reliability. Typically, process control systems employ redundancy both in hardware and software to achieve a more reliable system. Such redundancies can easily
be modelled through additional 'Behaviors' with appropriate replacements being nominated in the event of a hardware failure. For example, if the primary water pump were to fail, the corresponding behavior could nominate a replacement that represented a backup water pump. Also, with expert system supervisory shells serving as front ends for process control systems, graceful degradation of the system is achieved through "replacement" of optimal goals with more achievable goals [Chandrasekaran 91]. Such goal replacements can also be elegantly captured through the concept of a replacement behavior with each goal being modelled through a different behavior.

An acquaintance in the Actor model is an actor whose mail queue id has been passed in a mail message to another actor. It is through the concept of acquaintances that the dynamic topology can be realized, a key element in achieving the desired functionality in DREAMS. Whenever a state transition occurs the Goal, Policy and Failure blocks within each parameter subsection and the Environment Specification section have to be set up dynamically in order to have them communicate with one another. Once a transition occurs, the actors corresponding to the state being exited can be incrementally garbage collected [Nelson 90], freeing up critically needed resources.

The language prototype ACT++ combines the benefits of C++ with the concurrency semantics of the Actor model. In no way have any of the language features of C++ been compromised. A powerful inheritance mechanism is available in ACT++ and can be used effectively in capturing the commonalities that exist amongst the different hardware elements that typically make up a process control system.

5.3 A DREAMS based Actor Application

In the previous section we showed the relationship between a language prototype like ACT++ and the DREAMS architecture. In this section we provide detailed guidelines
for applying ACT++ to the DREAMS architecture and transforming it into a detailed design. Before we provide detailed guidelines for the Controller Specification and the Environment Specification stubs in the architecture, we will present general guidelines for the boundaries of the system, i.e. modelling of sensors, actuators, the external interface and how the communication topology is to be set up. Once again, only general guidelines have been presented for the boundaries of the system since DREAMS requires only black box functionality from the sensors and actuators and the external interface stubs. This black box functionality has been addressed in the individual sections.

Sensors

A major function within any process control system is data acquisition. Analog and digital sensors are used to monitor the different process variables describing the system, such as temperature, water level and pressure. Often, the acquired data has to be processed. Processing of the acquired data could involve converting to engineering units, validating for sensor failures, filtering or smoothing and entering into a process database to be used for other tasks.

A possible scheme for realizing the black box functionality of the Sensors stub would involve modelling each sensor through an actor. These actors are referred to as data source actors (DSA's). The inheritance mechanism within ACT++ could be used effectively in creating a hierarchy that captures the commonalities that exist amongst the different sensors. The commonalities might involve the sensor type, the validation involved or even the manner in which the acquired data is being processed. Each DSA would have as its acquaintance the Controller Specification actor (CSA). After processing the data each DSA would be required to send the data via a mail message to the Controller Specification actor. The DSA's should be viewed as independent data acquisition agents.
responsible for feeding the Controller Specification Actor. It is this black box behavior of
the sensors that is of relevance to DREAMS.

The DREAMS architecture shown in Figure 5.1 does not have a communication
arrow from the Controller Specification to the Sensors. This is because it is being assumed
that through redundancy in the hardware, there will always be a backup sensor(s) to
activate in the event the primary sensor were to fail. This appears to be a reasonable
assumption since sensors are usually passive elements that do not directly affect the
dynamics of the process. In the event of a sensor failure, the corresponding DSA could
nominate a replacement that encapsulated the structural details of the backup sensor to be
used instead. So essentially, in DREAMS it is assumed that the Controller Specification
actor would be continually fed data reflecting the current conditions of the process.

**Actuators**

Actuators are the hardware elements that are used to manipulate the process
variables in order to minimize their deviation from a required optimal behavior. The black
box functionality of relevance to DREAMS is the ability to communicate the desired
operational state to the Actuators stub and the ability to receive from the Actuators stub a
response in the event that a required operational state had changed. We are suggesting a
possible scheme to achieve this black box functionality.

Each actuator could be modelled through an actor and is referred to as the Actuator
Mechanism Actor (AMA). These actors encapsulate the *mechanism* to be used in
activating the hardware element they are the software counterpart of. Also, encapsulated
within these actors could be the reverse transformations for processing the command data.
For example, the responsibility for converting the command data from engineering units to
electronic units would rest with the AMA's in the system. In addition, all hardware
dependencies should be encapsulated within these actors in order to make the system amenable to low cost maintenance. These actors will be driven via mail messages from the Controller Specification actor and the actor corresponding to the Environment Specification stub in the DREAMS architecture. The set of actuators that will be driven by the Environment Specification stub will constitute the environment of the state the Controller Specification is representing. The operational state of these actuators is to remain invariant for the life of the state. In the event the operational state were to change, the change would be communicated back to the actor representing the Environment Specification stub, as has been shown in the architecture. The communication arrow from the Actuators stub to the Controller Specification is also for informing the latter of actuator failures, this time the set of actuators that are being manipulated by the Controller Specification actor.

**External Interface**

The External Interface for process control systems refers to the devices available to the operator for controlling the regular operation as well as the displays available for monitoring the condition of the process. In STARE we introduced the concept of an External Interface state. An External Interface state is a STARE state that embodies the actions to be taken corresponding to a command received from the External Interface. For each command that can be received from the External Interface, there should be a corresponding External Interface state defined. This structure is necessary since the controlling software enters the defined state when the command is executed.

Once again DREAMS only requires the ability to send and receive messages from the External Interface layer. However, we are presenting a scheme that would help organize the internal details of this layer. The actors used to model the External Interface are referred to as External Interface actors (EIA’s). An actor should be defined to
separately model each input and output hardware device. The actor(s) corresponding to the output device(s) will be responsible for updating the operator displays whereas the actor(s) responsible for monitoring the input device(s) will await the arrival of a command from the operator. The asynchronous I/O capability available in the extended ACT++ prototype is used to asynchronously monitor receipt of commands from the operator. The EIA's communicate with the Controller Specification Actor as has been shown in the DREAMS architecture.

**Topology**

The DREAMS architecture requires only black box functionality from the boundaries of the system. The core of this functionality is the communication between the different stubs. As long as there is a mechanism that enables a communication to be sent and received by a stub in the architecture, DREAMS does not require any specific manner in which the black box functionality is to be achieved, though a possible organizational scheme has been proposed in the previous sections.

If the organizational scheme proposed in the previous sections is to be realized it is imperative that each actor's acquaintances adhere to the communication structure in the architecture of Figure 5.1. The DSA's and the EIA's require the Controller Specification actor as an acquaintance. The Controller Specification actor and the actor corresponding to the Environment Specification stub require the other as an acquaintance as well as the relevant AMA's and the EIA's. The subset of AMA's that are ensuring the environment for the state need the Environment Specification actor as an acquaintance while the remaining AMA's require the Controller Specification actor as an acquaintance. The communication structure can then be implemented as messages sent from the currently executing behavior of an actor to the mail queue of the target actor.
5.4 Controller Specification

Before we provide detailed guidelines for transforming the Controller Specification stub in the DREAMS architecture into a detailed Actors design we would once again like to underscore an important feature of the ACT++ language prototype. It is through the Become operation that we are able to achieve an invariant relationship between the mail queue of an actor and its behavior while implementing state transitions. We have already alluded to the Controller Specification being implemented through an actor. However, at a given point in time the Controller Specification is part of a STARE state. Therefore, if we were to model each state as a behavior and use the become operation to nominate a replacement behavior whenever a transition takes place, the overall connectivity being proposed in DREAMS would not change and yet the system would be changing states as required.

In Figure 5.2 we presented the architecture of the Controller Specification. In this architecture we showed that for each state the Controller Specification half of it is modelled as a Collective Failures stub, executing concurrently with the Goal and Policy stubs for each parameter subsection. In DREAMS each of these concurrently executing stubs is modelled as an independent actor. In Figure 5.4 we provide a detailed architecture for designing the Controller Specification. The actors in the figure have been shown in rectangles whereas the behaviors of an actor have been shown in circles. If an actor has more than one behavior, the set of behaviors have been shown enclosed in a solid ellipse. However, if an actor has only one behavior, only the actor has been shown. The architecture shown would correspond to a hypothetical Controller Specification with two parameter subsections. In DREAMS the CSA, the Controller Specification actor, has a topology behavior T that is responsible for setting up the topology for that state and a receptionist behavior R. The desired topology has been shown enclosed in a dotted ellipse.
and the two behaviors of the CSA have been shown in a solid ellipse. The topology behavior would create the Collective Failures actor (CFA) with the CSA as its acquaintance. It would create the Policy actors with the appropriate AMA's as acquaintances. The Goal Actors (G) would be created with the CFA and the corresponding Policy actors (P) as acquaintances. On having set up this actor topology, the topology behavior would then nominate a receptionist R as a replacement and communicate to it the

*Figure 5.4 - Detailed architecture of controller specification*
topology it had set up. The Receptionist R would take the sensor values being fed to the Controller Specification from the Sensors stub and dispatch them to the Goal actors as appropriate. Once the Receptionist R has been nominated, it would remain the current behavior for the Controller Specification actor (CSA). It would keep nominating itself as a replacement until a state change is involved. Since a state change involves replacing the current behavior of the CSA, the Receptionist R would be responsible for nominating the topology behavior T corresponding to the new state being entered once a transition is to be initiated. The Receptionist R could then be made responsible for initiating the steps involved in incrementally garbage collecting the actors corresponding to the state being exited. The actors to be garbage collected have been enclosed in a dotted ellipse in Figure 5.4. Alternatively, a built-in garbage collection mechanism might exist which would assume this responsibility. Algorithms for such a garbage collector have been proposed in [Nelson 90].

Let us now apply the detailed architecture guidelines to the Controller Specification half of the STARE state shown in Figure 2.4 and create the ACT++ classes for it. This class structure has been shown in Figure 5.5 and Figure 5.6 with the method definitions for these classes included in Appendix D. Due to space constraints we have only shown the classes corresponding to the water level parameter subsection for the state definition in Figure 2.4. However, each parameter subsection within the Controller Specification would have a similar set of classes. There would be a class for the topology behavior and the receptionist behavior. A class would also be required for the Collective Failures actor and finally classes would be needed for the Goal and Policy stubs within each parameter subsection. The current prototype of the language provides the same base class, ACTOR, for both Actors and their behaviors. Hence the topology and receptionist behaviors have been shown being derived from the ACTOR class. Using Figure 5.5
as a reference, we will now provide a detailed description of these classes. The topology
behavior would set up the topology and nominate the Receptionist behavior as a

typedef struct list_of_actors {
    Mbox *head;
    struct list_of_actors *tail;
} LIST_OF_ACTORS;

class Startup_Topology_T : public ACTOR {
public :
    void set_up_topology();
};

class Startup_Receptionist_R : public ACTOR {
    LIST_OF_ACTORS topology;
    Mbox esa;
    int water_level;
    int pressure;
    int temperature;
    int wp_status;
    int prv_status;
    int gv_status;
    int time;
    int dsa_count;
    void update_topology_list();
    void garbage_collect_state();
public :
    void dispatch();
    void recv_sensor_value();
    void recv_time();
    void recv_actuator_state();
    void nominate_shutdown();
    void nominate_startup_PV_open();
    void nominate_prime_water_level();
    void nominate_startup_excess_pressure();
    void nominate_startup_release_steam();
    void nominate_normal();
};

Figure 5.5 - Topology and receptionist behavior classes
replacement. It would then communicate the topology it had set up to the Receptionist by sending the recv_acquaintances message. When the Receptionist becomes the current behavior for the CSA, there are two types of messages that can be interleaved in the CSA’s mail queue. First, there are the messages that are coming from the DSA. These messages would communicate the sensor values (recv_sensor_values), actuator states (recv_actuator_states) and the current simulation time (recv_time). The second set of messages would correspond to the transitions that should be initiated in the event of failures (nominate prime_water_level, nominate startup_excess_pressure, nominate startup release

```java
class Startup_Water_Level_Goal : public ACTOR {
    Mbox cfa;
public:
    void recv_acquaintances();
    void goal();
};

class Startup_Collective_Failures : public ACTOR {
    Mbox csa;
public:
    void recv_acquaintances();
    void water_level_failures();
    void pressure_failures();
    void temperature_failures();
};
```

Figure 5.6 - Parameter subsection and collective failures classes

steam, nominate normal, nominate shutdown and nominate_startup_PV_open).

In Figure 5.6, due to space constraints we show the classes for the water level parameter subsection only. Also shown is the class for the Collective Failures actor. Each parameter subsection will have classes corresponding to the goals and the policies. The classes have a similar structure. The acquaintances are represented as private data members.
and they have two methods: one to receive the acquaintances and one corresponding to the goal or policy as appropriate. The Collective Failures class, Startup_Collective_Failures has the CSA as an acquaintance and has a method for receiving this acquaintance as well as methods for the different failures that can occur from the Controller Specification part of a STARE state.

5.5 Environment Specification

In Figure 5.7 we show the detailed architecture of the Environment Specification stub. The structuring is similar to what was shown for the Controller Specification component. However, there is no receptionist needed to feed the Goals since this half of a state specification is not dependent on the arrival of stimuli (sensor data) from the environment. The topology behavior T, after establishing the actor topology shown enclosed in a dotted ellipse in Figure 5.7, nominates the Failure behavior F whose sole purpose is to communicate to the Controller Specification Actor that the environment for the state it was representing has changed. In Figure 5.8 we show the classes that would be required for implementing the topology and failure behaviors that have been shown in Figure 5.7. As can be seen in Figure 5.8, since the detailed architecture for the Environment Specification is similar to the one described for the Controller Specification, even the class structure for it is very similar to the classes that were created for the Controller Specification. Figure 5.9 shows the remaining classes for the Environment Specification. Since the Environment Specification half of a state definition is initially ensuring, and thereafter monitoring, the desired environment for a state, each sub goal within the goal predicate is being modelled by a separate actor. These actors block themselves (on Cboxes) after they have communicated the operational state to the single Policy actor. The single Policy actor resolves any sequencing constraints and uses reply
forwarding to request the relevant AMA to raise an exception by replying to the Cbox on

![Diagram](image)

**Legend**

- Replacement
- Mail Message
- State Topology
- Multiple Behaviors
- Actor with one Behavior
- Actor with multiple Behaviors

*Figure 5.7 - Detailed architecture of the environment specification*

which the sub goal actor was blocked if the required operational state were to unexpectedly change at any time during the life of the state. Details of the method implementations have been included in Appendix D.
5.6 Summary

typedef struct list_of_actors {
    Mbox *head;
    struct list_of_actors *tail;
} LIST_OF_ACTORS;

class Startup_Environment_Spec_T : public ACTOR {
public:
    void set_up_topology();
};

class Startup_Environment_Spec_F : public ACTOR {
    LIST_OF_ACTORS *topology;
    Mbox csa;
    void update_topology_list();
    void garbage_collect_state();
public:
    void go_into_shutdown();
    void nominate_startup_pv_open();
    void report_PRV_failure();
};

Figure 5.8 - Topology and failure classes for environment specification

This chapter presented DREAMS, a design approach that takes a STARE-based requirements specification document and transforms it into an architectural design using the principles of concurrency. Also proposed in DREAMS was an organization that realized the concurrency within the DREAMS architecture using extended ACT++, a concurrent object-oriented language prototype that derives its concurrency semantics from the Actor model.

By using active objects to model the different sections of a STARE state, DREAMS provides a clean transformation path to move from the specification to the
design. The concurrency semantics available in the ACT++ language prototype provide a

```
class Startup_PRV_Goal : public ACTOR {
  Mbox *cfa;
public:
  void recv_acquaintances();
  void goal();
};

class Startup_PV_Goal : public ACTOR {
  Mbox *cfa;
public:
  void recv_acquaintances();
  void goal();
};

class Startup_Collective_Failures : public ACTOR {
  Mbox esa;
public:
  void recv_acquaintances();
  void PRV_failures();
  void PV_failures();
};
```

*Figure 5.9* - Goal, policy, failure classes for environment specification

natural way of structuring the communication amongst the active objects. Finally the fresh
perspective that DREAMS imparts to some of the language features within ACT++
provides an elegant way of capturing the dynamic manner in which process control systems
respond to their environment.
CHAPTER 6 - CONCLUSIONS

This thesis presented STARE, a requirements analysis approach for process control systems. The proposed approach provides a structured path for transforming a prose like problem statement into a structured requirements specification. The approach, however, addresses only the functional behavior of process control systems. The approach does not address the specification of performance, reliability, safety or other non-functional aspects of embedded systems.

Also proposed was a design approach, DREAMS, based on the principle of concurrent active objects for transforming a STARE-based requirements specification into a detailed design. Detailed guidelines were presented for designing the states. However, only general guidelines were presented for the boundaries of the system. In addition, the efficacy of an active objects based programming paradigm for developing process control applications was demonstrated. The natural manner in which active objects can model the hardware elements that make up a process control system and the communication between them was emphasized.

The requirements analysis approach in conjunction with the design approach provides a high level of traceability; a piece of code can easily be related to the problem statement. This high level of traceability can be attributed to the following factors. First, through a requirements analysis approach we were able to transform the problem statement
into a structured specification. We then proposed a design architecture to exploit the structure of the specification. Each section of the specification was mapped to a stub in the architecture. Finally, we presented a detailed implementation strategy for organizing the internal details of the objects resulting from the architecture. Guidelines were provided for organizing ACT++ classes around each stub. Hence, a path was provided for mapping a section of a STARE state to an ACT++ class. A approach that provides such a level of traceability promotes the reasoning and analysis of the problem at a much higher level of abstraction, specifically at the level of the requirements. The analysis of a problem at the requirements level improves the reliability of the end product in addition to reducing the development as well as maintenance costs, especially for highly evolutionary systems.

The proposed software development approach was illustrated by a representative process control problem. While the scale of this problem is certainly not representative of industrial process control applications, it does have the basic elements that would be of interest to a process control engineer. This problem, we believe, was a good preliminary test. Even using this minimal problem as a testbed for the proposed software development approach, we discovered several interesting issues.

The work presented in this thesis is the first step toward a long term goal of providing an integrated software development environment for process control systems. The focus of this chapter is to discuss the manner in which the proposed software development approach can be enhanced keeping in mind the long term goal. These enhancements have been presented in the form of informal discussions on the requirements analysis approach, the design approach as well as the concurrent object-oriented language presented in this thesis.
6.1 A Discussion on STARE

STARE, in conjunction with a design approach like DREAMS, provides a high level of traceability and thereby paves the way for a computer-aided software engineering tool that could support the software development process for embedded systems. However, STARE in its present state falls considerably short of providing even a partially automated software development environment. There are several issues that need to be addressed before it can be declared as a robust tool in developing software for embedded systems. We address each of these issues individually.

State Transitions

While presenting STARE we briefly alluded to the consistency that is desired when a transition takes place. STARE does not provide any mechanism for arbitrating between simultaneous transitions. However, a heuristic, based on definitions of appropriate parameter subsections and goal definitions therein, was proposed to ensure the consistency of the state being entered. The validity of this heuristic as well as the need for additional heuristics is being investigated.

Representation Scheme

In order to use STARE to create executable specifications it is essential to have a formal representation scheme that would enable consistency checks to be run for validation purposes. In its present form, STARE lacks a formal representational scheme. Any representation scheme must satisfy several criteria. The specification of performance requirements for embedded systems is a challenging problem in itself. Any representation scheme should address the specification of performance requirements and also their verifiability without, however, comprising on the comprehensibility of the resulting
document. Ease of use and modifiability are important criteria as well. Most important, the chosen representation scheme should facilitate the process of creating executable specifications. As noted earlier, STARE could serve as a front-end to existing specification techniques. However, as part of the effort to create an integrated software development environment for embedded systems, an appropriate representation scheme is under investigation.

**Non-Functional Requirements**

As noted earlier, STARE only addresses the functional behavior of process control systems. This behavior is encapsulated within the Controller component of the feedback loop that is at the heart of any process control system. For some applications, addressing the functional behavior is sufficient. However, for embedded real-time systems it is not only important to describe what needs to be done but also express *when* or *how soon* it needs to be done. In order to provide an integrated software development environment we need to be able to specify the performance requirements as well. In addition, we also need to address the reliability aspect as well as the graceful degradation issues affecting the specification of embedded systems.

**Consistency**

Consistency within the requirements is an important criterion for any specification technique to satisfy. The representative real-time process control problem that we used for illustrating STARE, though minimal when compared to industrial process control systems, was sufficient to indicate the importance of ensuring a consistent specification. Even in our sample problem we observed the need for heuristics to ensure consistent state transitions. Such heuristics are being investigated.
Domain Knowledge

An issue that has a considerable impact on the effectiveness of a requirements analysis approach like STARE is field knowledge about the application domain. The embedded systems domain is highly complex. In presenting STARE we were motivated by the generic feedback control system that we believe is common across many embedded systems applications. Our lack of detailed engineering knowledge did not preclude an analysis of this feedback control system. The STARE approach was a product of this analysis. However, we do realize that certain critical engineering issues might not have been addressed as part of this analysis. As we gain more experience with the engineering aspects of embedded systems we will adapt STARE to provide the desired functionality.

Reliability

An important motivational factor in presenting a approach like STARE was how to improve the reliability of embedded systems. Reusability of specifications and in conjunction with DREAMS, reusability of design as well, can play an important role in realizing improved reliability as well as reduced development costs. Since the sections within a STARE state are related to the operating characteristics of a hardware element, they facilitate the reuse of specifications. Reusability approaches like DRACO [Neighbors 90] could very well be used in conjunction with a approach like STARE.

6.2 A Discussion on DREAMS

Architecture

The effectiveness of the DREAMS architecture, specifically the communication between the different stubs, needs further investigation. We believe that the architecture is generally appropriate for embedded systems. A more comprehensive evaluation at this
stage is precluded by the lack of field experience in the embedded systems application domain. Specifically of concern is the issue whether a general principle can be advocated for detecting actuator failures.

The efficacy of a message-passing based communication paradigm for developing embedded applications also needs to be investigated.

**Boundaries of the System**

As pointed out earlier, due to the lack of detailed knowledge about how embedded systems are engineered in practice, the DREAMS approach was restricted to the Controller Specification and the Environment Specification stubs in the architecture. Only general guidelines addressing the required black box behavior were presented for the boundaries of the system. Eventually, detailed guidelines for the boundaries of the system should be presented as well.

Embedded systems achieve greater reliability and high availability of the system through dynamic redundancy in the hardware. Since the redundant hardware is contained within the boundaries of the system, the importance of providing a approach that addresses these issues is only further emphasized. Graceful degradation of the system is another factor that underscores the importance of a cohesive approach. These issues will be dealt with as we gain more experience with the embedded systems application domain.

**State Transitions**

One of the key elements in the DREAMS approach was the manner in which a state transition was realized. The approach provides a fresh perspective on the concept of a replacement behavior by using it to implement a state transition. A state in STARE consisted of a Controller Specification and an Environment Specification. In DREAMS the
two halves of a state definition are modelled as separate stubs. Whenever a state transition takes place these two halves independently transition into the corresponding half of the new state. It is not known if there is a need to impart the notion of an atomic transaction to the two independent transitions that are taking place. Also, presently it is assumed that a state transition does not affect the boundaries of the system. The validity of this assumption needs to be verified especially in the context of redundant hardware being used to improve the reliability of the system.

6.3 A Discussion on ACT++

We described a concurrent object-oriented language prototype, ACT++, that combines the concurrency semantics of the ACTOR model with the object-oriented language C++. The language primitives were described along with how the primitives provide an elegant mechanism for implementing the DREAMS approach. However, there are several language issues of concern to us, especially in the context of the DREAMS approach.

Express Messages

The present language prototype does not support priority messages. In DREAMS a state transition is achieved through a mail message. For instance, the Collective Failures actor would send a message to the receptionist for that state which would then initiate the state transition. Since the receptionist is also being fed by the DSA's with the sensor values, we need a mechanism that would ensure that the message corresponding to the state transition is processed first. This can be easily achieved by communicating the need
for a state transition through an express message. Providing support for such a prioritized message passing scheme is being investigated in context of the current prototype.

**Cboxes**

In the current ACT++ prototype it is not possible to extract a Cbox from a message. Instead the current behavior is required to use the reply forwarding mechanism in order to send a reply to that Cbox. This scheme does not seem to work very well for embedded applications. Often, before the currently executing behavior can send a reply on a Cbox, it has to nominate several replacements. Since these nominations are in a sense continuations, it does not satisfy the initial design objectives of the Cbox abstraction.

**Behaviors**

The ACT++ prototype implementation does not differentiate between an actor and a behavior. Both actors and behaviors are required to be derived from the ACTOR base class. In the proposed software development approach, behaviors play a critical role in achieving state transitions as well as encapsulating state related information. With the current prototype not differentiating between an actor and its associated behavior there is an excessive run-time overhead. Whenever only a behavior is needed, instantiating an instance of the ACTOR class instead results in a Mbox being created as well. These additional unused Mboxes present an excessive run-time overhead.

**Additional Abstractions**

An important issue that is being investigated is how to make embedded systems resynchronize with their environment. This is especially important for highly responsive systems. For example, a overly fast data acquisition component (DSA) may overwhelm
the component that processes the data (the Controller Specification), in which case at some point the decisions being taken by the Controller Specification would not reflect the current condition of the process. Abstractions that address this flow control problem without constraining the throughput of the system need to be investigated further.

We would also like to investigate the need for special purpose actors and behaviors and how they could ease the software development process. For example, we create Goal actors, Policy actors and Failure actors. Special purpose behaviors could be created corresponding to the different states the Controlling software could be in. Once again, these issues will be dealt with as more experience is gained with the engineering aspects of embedded systems.
APPENDIX A - STEAM GENERATOR CONTROL

Problem Description

As shown in Figure 1, the embedded control system monitors and regulates the steam generator under the control of a human operator. The control system must continuously display for the operator data reflecting the current operating condition of the generator. The operator may also issue commands to the control system. The control system must monitor the generator and regulate its condition to achieve safe, continuous operation. The steam generator itself is shown in Figure 2. Steam is produced by heating water in a large, closed tank. The resulting steam is vented from the top of the tank. The amount of water in the tank is measured by a water level indicator. Water may be added to the tank by activating a water pump. The tank may be drained by opening a purge valve located at the bottom of the tank. The water is heated by burning natural gas. The amount of steam produced is controlled by regulating the flow of natural gas. At the top of the tank is a pressure relief valve which allows excess steam to be vented. The temperature and pressure inside the tank are measured by gauges attached to the tank. The embedded control system is connected to the steam generator by sensors and actuators. The sensors allow the control system to determine the readings of the water level indicator, the temperature gauge and the pressure gauge. Two actuators allow the control system to open and close the pressure relief valve and the purge valve. Another actuator allows the control system to start and stop the water pump. The operation of the steam generator is described in three parts: the procedure to start up the generator, the rules for continuous, safe operation and the procedure to shut down the generator.
Figure A.1 - Embedded control system

Startup:
The start up sequence is initiated by an operator command to the control system. To initiate the steam generator the water level must be at least 50% before the gas is turned on. If the water level is greater than 85% the start up sequence should not continue until field operators have corrected the problem. An appropriate message must be displayed periodically describing why the generator is not being brought on line. When the water level is within bounds the start up may resume. The pressure relief valve must be closed before the gas is applied. If the initial temperature is below 350 degrees Fahrenheit gas should be applied at an 80% level. The control strategy for normal operation (described below) applies during the start up sequence.

Normal Operation:
The rules governing the normal operation of the steam generator are described in terms of the sensor values that must be maintained and the controls that must be applied to the actuators. The sensors (water level indicator, pressure gauge, temperature gauge) are
read-only. The actuators (water pump, pressure relief valve, purge valve) can be read and written to. Each actuator maintains the last value written to it. When the actuator is read the current value is returned. Reading the actuators allows the initial status of the system to be determined. Also, a malfunctioning actuator can be determined when the current value read is not the same as the previous value written. For digitally controlled actuators, a 0 value means to stop / close while a 1 value means to start / open.
Water Level Indicator
This indicator is an analog value (0 - 100%) measuring the amount of water in the tank. Whenever the gas burner is being used this level must not exceed 85% or go below 15%. Should either of these limits be violated, the pressure relief valve must be opened, the gas turned off and the water pump stopped. An alarm message must also be displayed for the operator. In normal operation the water level is controlled by adding water when the level drops below 35% and by turning off the pump when the level exceeds 65%.

Pressure Gauge
This gauge is an analog value (0 - 600) psi measuring the steam pressure in the tank. If the pressure exceeds 500 psi the pressure relief valve must be opened, the gas turned off and an alarm message displayed for the operator. This alarm message must remain displayed as long as the pressure is above 500 psi. The alarm message must display the number of minutes since the alarm condition arose. When the pressure falls below 480 psi the generator operation should return to normal. The water pump must not be operated if the pressure is above 450 psi.

Temperature Gauge
This gauge is an analog value (100 - 500 degrees Fahrenheit) measuring the steam temperature at the top of the tank. The temperature should be controlled to 350 degrees +/- 10 degrees. The control strategy to maintain the operating range is as follows: for every 1 minute that the temperature is out of the desired range adjust the gas flow by 1% for each 1 degree that the temperature is out of range. An alarm message must be displayed if the current temperature exceeds 375 degrees. This alarm message must remain displayed as long as the temperature is above 375 degrees. The alarm message must display the number of minutes since the alarm condition arose.

Water Pump
This is a digital actuator which controls the power relay for the water pump.

Pressure Relief Valve
This is a digital actuator which controls the pressure relief valve at the top of the tank.
Gas Supply Valve
This is an analog actuator (0 - 100%) which controls the amount of natural gas supplied to the steam generator.

Purge Valve
This valve is located on the bottom of the tank and is used to empty all water from the tank. It must not be opened while the gas is being applied. If at any time the gas is on or the boiler pressure is above 125 psi and this valve is sensed open, the pressure relief valve must be opened and the gas shut off. This is a safety critical condition and appropriate alarm messages must be displayed for the operator. This alarm message must remain displayed as long as the alarm condition persists. The alarm message must display the number of minutes since the alarm condition arose.

Shutdown:
The generator must be shut down if any of the extreme limits are exceeded. A shutdown may also be requested by the operator. The shutdown requires that the gas be shut off completely. If the gas supply valve fails to respond, an appropriate message must be displayed for the operator. Throughout the shutdown sequence messages should be displayed keeping the operator informed of the progress of the shutdown operation. The normal shutdown sequence is as follows:

The normal shutdown sequence is as follows:
• stop all gas flow
• open the pressure relief valve
• when the pressure drops below 50 psi open the purge valve

When the pressure drops to 0 psi and temperature is below 100 degrees Fahrenheit the generator is considered to be in a safe condition and has been shut down. A message indicating the end of the shut down sequence must be displayed for the operator.

Communication with the Simulator

Messages between the simulator and the control program are communicated over a single stream socket. The format and purpose of each message is described below. Low-level
details about the socket and several other messages which are provided for testing purposes will be described in a later document.

There are five (5) message types each containing simple text (ASCII) strings. The message types differ in length but all messages begin with a 7 byte header containing the name of the message. The message name is left-justified in the header field. The message names are REQUEST, DRIVE, STATE, FAILURE, CLOCK. Each message ends with a single byte called the end-of-text character ("/").

Two of the messages are sent spontaneously by the simulator. Two of the messages are sent from the control program to the simulator. The final message is sent by the simulator in direct response to a control program message.

**Delayed Response**

Since the simulator will be attempting to model the real world, output commands to the simulator do not take effect immediately. Time must be allowed for such things as valve travel and motor response. In general, if an output command has not taken effect within 30 seconds the actuator should be considered as faulty and some other action may be necessary to prevent undesirable results.

**Simulator Messages**

The two messages sent spontaneously by the simulator are the CLOCK and FAILURE messages.

**CLOCK Message**

This message is sent at regular intervals and is the control program's only concept of "real" time. A clock message contains the current clock tick count. The tick rate will be approximately 3 ticks every 10 seconds.
**FAILURE Message**

The FAILURE message is sent when the control program has allowed the simulator to exceed one or more of its extreme operating limits. For example, if the water pump is started and not stopped the water level will rise until it begins filling the steam outlet. Such situations are considered hazardous. The FAILURE message contains a 30 byte "reason" field indicating the nature of the failure. In response to this message, the control program should display the reason for failure and terminate.

```
FAILURE  <REASON>>  EOT
```

**Control Program Messages**

The two messages sent by the control program are the REQUEST and DRIVE messages.

**REQUEST message**

This message is the means by which the control program interrogates the condition of the simulator. In response to this message, the simulator sends the STATE message described below.

```
REQUEST  EOT
```
**DRIVE Message**

This message is sent to the simulator to control its operation. As shown in the next figure, the "c" fields are single bytes ("0" or "1") indicating whether you want the corresponding digital actuator closed ("0") or opened ("1"). The "xxx" field is the percent open you want the gas supply valve.

![Diagram of DRIVE Message]

The last message is sent by the simulator in response to the control program's REQUEST message. It is the STATE message.

**STATE Message**

As shown below, the first four fields are the actual states of the four devices: pressure relief valve, water pump, drain valve and the gas supply valve. The last three fields indicate the state of the read-only analog sensors (temperature, pressure, water level). The values for the four analog values are 0-100%, 100-500 degrees Fahrenheit, 0-600 Psi and 0-100% respectively.
APPENDIX B - REQUIREMENTS SPECIFICATION

This appendix contains the requirements Specification obtained by applying SPARC to the process control problem described in Appendix A. The detailed states described here were summarized in Chapter 2 when we presented the SPARC methodology.

<table>
<thead>
<tr>
<th>Environment Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Invariant</strong></td>
</tr>
<tr>
<td>PRV Closed, PV Closed</td>
</tr>
<tr>
<td><strong>Failure</strong></td>
</tr>
<tr>
<td>PRV Open =&gt; Report Failure</td>
</tr>
<tr>
<td>PV Open =&gt; State Transition (Startup_PV_Open)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Controller Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Water Level (WL)</strong></td>
</tr>
<tr>
<td>• Goal</td>
</tr>
<tr>
<td>WL == 50</td>
</tr>
<tr>
<td>• Failure</td>
</tr>
<tr>
<td>50 &lt; WL &lt; 85 =&gt; State Transition (Normal)</td>
</tr>
<tr>
<td>WL &gt; 85% =&gt; State Transition (Shutdown)</td>
</tr>
<tr>
<td>WL &lt; 50% =&gt; State Transition (Prime_Water_Level)</td>
</tr>
<tr>
<td><strong>Pressure (P)</strong></td>
</tr>
<tr>
<td>• Goal</td>
</tr>
<tr>
<td>(P &lt; 450 Psi)</td>
</tr>
<tr>
<td>• Failure</td>
</tr>
<tr>
<td>450 &lt; P &lt; 500 =&gt; State Transition (StartUp_Excess_Pressure)</td>
</tr>
<tr>
<td>P &gt; 500 Psi =&gt; State Transition (StartUp_Release_Steam)</td>
</tr>
<tr>
<td><strong>Temperature (T)</strong></td>
</tr>
<tr>
<td>• Goal</td>
</tr>
<tr>
<td>(340 &lt; T &lt; 360)</td>
</tr>
<tr>
<td>• Policy</td>
</tr>
<tr>
<td>T &lt; 350 =&gt; Apply gas at 80% level.</td>
</tr>
<tr>
<td>T &gt; 350 =&gt; For every minute the temp is out of range adjust gas flow by 1% for each degree it is out of range.</td>
</tr>
<tr>
<td>• Failure</td>
</tr>
<tr>
<td>T &gt; 375 F =&gt; Alarm Message</td>
</tr>
</tbody>
</table>

**Figure B.1 - Startup state definition**

95
**Environment Specification**

- **Invariant**
  
  PV Closed, WP On, GV Off, PRV Closed

- **Constraints**
  
  Close PV before turning WP on

- **Failure**
  
  PV Open => State Transition (PWL_PV_Open)
  WP Off => Report Failure
  GV On => Report Failure
  PRV open => Report Failure

**Controller Specification**

**Water Level (WL)**

- **Goal**
  
  WL < 50%

- **Failure**
  
  WL > 50% => State Transition (Startup)

**Pressure (P)**

- **Goal**
  
  P < 450 Psi

- **Failure**
  
  450 < P < 500 => State Transition (PWL_Excess_Pressure)
  P > 500 Psi => State Transition (PWL_Release_Steam)

**Temperature (T)**

- **Goal**
  
  340 < T < 360

- **Failure**
  
  T > 375 F => Alarm Message

*Figure B.2 - Prime_Water_Level state definition*
### Environment Specification

- **Invariant**
  - PRV Closed, WP Off, PV Closed
- **Failure**
  - PRV open $\Rightarrow$ Report Failure
  - WP on $\Rightarrow$ Report Failure
  - PV open $\Rightarrow$ State Transition (Shutdown)

### Controller Specification

#### Water Level (WL)
- **Goal**
  - WL $< 85\%$
- **Failure**
  - WL $> 85\%$ $\Rightarrow$ State Transition (Shutdown)

#### Pressure (P)
- **Goal**
  - 450 < P < 500 Psi
- **Failure**
  - P $< 450$ $\Rightarrow$ State Transition (StartUp)
  - P $> 500$ Psi $\Rightarrow$ State Transition (StartUp_Release_Steam)

#### Temperature (T)
- **Goal**
  - 340 < T < 360
- **T Policy**
  - T $< 350$ $\Rightarrow$ Apply gas at 80% level.
  - T $> 350$ $\Rightarrow$ For every minute the temp is out of range adjust gas low by 1% for each degree it is out of range.
- **Failure**
  - T $> 375$ F $\Rightarrow$ Alarm Message

---

**Figure B.3 - Startup_Excess_Pressure state definition**
Environment Specification

- **Invariant**
  PRV Open, WP Off, PV Closed, Gas Off
- **Constraints**
  Turn Gas Off before Opening PRV
- **Failure**
  PRV Closed $\Rightarrow$ Report Failure
  WP On $\Rightarrow$ Report Failure
  PV Open $\Rightarrow$ State Transition (Shutdown)
  Gas On $\Rightarrow$ Report Failure

Controller Specification

**Water Level (WL)**
- **Goal**
  WL < 85%
- **Failure**
  WL > 85% $\Rightarrow$ State Transition (Shutdown)

**Pressure (P)**
- **Goal**
  P > 500 Psi
- **Failure**
  50 < P < 500 Psi $\Rightarrow$ State Transition (StartUp_Excess_Pressure)
  P < 450 Psi $\Rightarrow$ State Transition (StartUp)

**Temperature (T)**
- **Goal**
  (340 < T < 360)
- **Failure**

**Figure B.4 - Startup_Release_Steam state definition**
Environment Specification

- **Invariant**
  Gas Off, WP Off, PV Open

- **Failure**
  - Gas On $\Rightarrow$ State Transition (Shutdown)
  - PV closed $\Rightarrow$ State Transition (Startup)

Controller Specification

**Water Level (WL)**
- **Goal**
  WL $< 85\%$
- **Failure**
  WL $> 85\%$ $\Rightarrow$ State Transition (Shutdown)

**Pressure (P)**
- **Goal**
  (P $< 125$Psi)
- **Failure**
  P $> 125$ $\Rightarrow$ State Transition (Shutdown)

**Temperature (T)**
- **Goal**
  (340 $< T < 360$)
- **Policy**
  - T $< 350$ $\Rightarrow$ Apply gas at 80% level.
  - T $> 350$ $\Rightarrow$ For every minute the temp is out of range adjust gas flow by 1% for each degree it is out of range.
- **Failure**
  T $> 375$ F $\Rightarrow$ Alarm Message

**Figure B.5** - Startup_PV_Open state definition
Environment Specification

- Invariant
  PRV Closed, WP Off, PV Closed

- Failure
  PRV open => Report Failure
  WP on => Report Failure
  PV open => State Transition (Shutdown)

Controller Specification

Water Level (WL)
- Goal
  WL < 85%

- Failure
  WL > 85% => State Transition (Shutdown)

Pressure (P)
- Goal
  450 < P < 500 Psi

- Failure
  P < 450 => State Transition (Prime_Water_Level)
  P > 500 Psi => State Transition (PWL_Release_Steam)

Temperature (T)
- Goal
  340 < T < 360

- T Policy
  T < 350 => Apply gas at 80% level.
  T > 350 => For every minute the temp is out of range adjust gas low by 1% for each degree it is out of range.

- Failure
  T > 375 F => Alarm Message

Figure B.6 - PWL_Excess_Pressure state definition
Environment Specification

- **Invariant**
  PRV Open, WP Off, PV Closed, Gas Off
- **Constraints**
  Turn Gas Off before Opening PRV
- **Failure**
  PRV Closed => Report Failure
  WP On => Report Failure
  PV Open => State Transition (Shutdown)
  Gas On => Report Failure

Controller Specification

**Water Level (WL)**
- **Goal**
  WL < 85%
- **Failure**
  WL > 85% => State Transition (Shutdown)

**Pressure (P)**
- **Goal**
  P > 500 Psi
- **Failure**
  450 < P < 500 Psi => State Transition (PWL_Excess_Pressure)
  P < 450 Psi => State Transition (PWL)

**Temperature (T)**
- **Goal**
  (340 < T < 360)
- **Failure**
  T > 375 => Alarm Message

Figure B.7 - PWL_Release_Steam state definition
Environment Specification

- Invariant
  Gas Off, WP Off, PV Open
- Failure
  Gas On => State Transition (Shutdown)
PV closed => State Transition (Prime_Water_Level)

Controller Specification

Water Level (WL)
- Goal
  WL < 85%
- Failure
  WL > 85% => State Transition (Shutdown)

Pressure (P)
- Goal
  (P < 125 Psi)
- Failure
  P > 125 => State Transition (Shutdown)

Temperature (T)
- Goal
  (340 < T < 360)
- Policy
  T < 350 => Apply gas at 80% level.
  T > 350 => For every minute the temp is out of range adjust
gas flow by 1% for each degree it is out of range.
- Failure
  T > 375 F => Alarm Message

Figure B.8 - PWL_PV_Open state definition
Environment Specification

- **Invariant**
  PRV Closed, PV Closed

- **Failure**
  - PRV Open => Report Failure
  - PV Open => State Transition (Normal_PV_Open)

Controller Specification

**Water Level (WL)**
- **Goal**
  15% < WL < 85%
- **Policy**
  - WL < 35% => Turn WP On
  - WL > 65% => Turn WP Off
- **Failure**
  - WL > 85% => State Transition (Shutdown)
  - WL < 15% => State Transition (Shutdown)

**Pressure (P)**
- **Goal**
  (P < 450 Psi)
- **Failure**
  - 450 < P < 500 => State Transition (Normal_Excess_Pressure)
  - P > 500 Psi => State Transition (Normal_Release_Steam)

**Temperature (T)**
- **Goal**
  (340 < T < 360)
- **Policy**
  - T < 350 => Apply gas at 80% level.
  - T > 350 => For every minute the temp is out of range adjust gas flow by 1% for each degree it is out of range.
- **Failure**
  - T > 375 F => Alarm Message

**Figure B.9 - Normal state definition**
Environment Specification

- **Invariant**
  PRV Closed, WP Off, PV Closed

- **Failure**
  PRV Open => Report Failure
  WP On => Report Failure
  PV Open => State Transition (Shutdown)

Controller Specification

**Water Level (WL)**

- **Goal**
  15% < WL < 85%

- **Failure**
  WL > 85% => State Transition (Shutdown)
  WL < 15% => State Transition (Shutdown)

**Pressure (P)**

- **Goal**
  450 < P < 500 Psi

- **Failure**
  P < 450 => State Transition (Normal)
  P > 500 Psi => State Transition (Normal_Release_Steam)

**Temperature (T)**

- **Goal**
  340 < T < 360

- **Policy**
  T < 350 => Apply gas at 80% level.
  T > 350 => For every minute the temp is out of range adjust gas flow by 1% for each degree it is out of range.

- **Failure**
  T > 375 F => Alarm Message

**Figure B.10** - Normal_Excess_Pressure state definition
Environment Specification

- **Invariant**
  PRV Open, WP Off, PV Closed, Gas Off
- **Constraints**
  Turn Gas Off before Opening PRV
- **Failure**
  PRV Closed $\Rightarrow$ Report Failure
  WP On $\Rightarrow$ Report Failure
  PV Open $\Rightarrow$ State Transition (Shutdown)
  Gas On $\Rightarrow$ Report Failure

Controller Specification

**Water Level (WL)**

- **Goal**
  $15 < WL < 85\%$
- **Failure**
  WL $> 85\%$ $\Rightarrow$ State Transition (Shutdown)
  WL $< 15\%$ $\Rightarrow$ State Transition (Shutdown)

**Pressure (P)**

- **Goal**
  P $> 500$ Psi
- **Failure**
  $450 < P < 500$ Psi $\Rightarrow$ State Transition (Normal_Excess_Pressure)
  P $< 450$ Psi $\Rightarrow$ State Transition (Normal)

**Temperature (T)**

- **Goal**
  $340 < T < 360$
- **Failure**
  T $> 375$ F $\Rightarrow$ Alarm Message

Figure B.11 - Normal_Release_Steam state definition
Environment Specification

- **Invariant**
  Gas Off, WP Off, PV Open

- **Failure**
  Gas On => State Transition (Shutdown)
  PV closed => State Transition (Normal)

Controller Specification

**Water Level (WL)**
- **Goal**
  15 % < WL < 85%
- **Failure**
  WL > 85% => State Transition (Shutdown)
  WL < 15% => State Transition (Shutdown)

**Pressure (P)**
- **Goal**
  P < 125 Psi
- **Failure**
  P > 125 => State Transition (Shutdown)

**Temperature (T)**
- **Goal**
  340 < T < 360
- **Policy**
  T < 350 => Apply gas at 80% level.
  T > 350 => For every minute the temp is out of range adjust gas flow by 1% for each degree it is out of range.
- **Failure**
  T > 375 F => Alarm Message

Figure B.12 - Normal_PV_Open state definition
Environment Specification

- Invariant
  PRV Open, GV Off, PV Open, WP Off
- Constraints
  First close GV, then open PRV, then close WP and finally open PV
- Failure
  PRV closed => Report Failure
  PV closed => Report Failure
  GV On => Report Failure
  WP Or: => Report Failure

Controller Specification

Pressure (P)
- Goal
  P == 50 Psi
- Failure
  P > 50 Psi => State Transition (Shutdown)
  P < 50 Psi => State Transition (Generator_Cool_Down)

Figure B.13 - Shutdown state definition

Environment Specification

- Invariant
  PRV Open, GV Off, PV Open, WP Off
- Constraints
  First close GV, then open PRV, then close WP and finally open PV
- Failure
  PRV closed => Report Failure
  PV closed => Report Failure
  GV open => Report Failure
  WP open => Report Failure

Controller Specification

Temperature(T)
- Goal
  T == 100 F
- Failure
  T > 100 F => State Transition (Generator_Cooldown)
  T < 100 F => Exit

Figure B.14 - Generator_Cool_Down state definition
APPENDIX C - HEADER FILE FOR STARTUP STATE

This appendix contains the header file for the Normal state definition in SPARC. Part of
this header file was presented when we introduced SPARC in chapter 2.

```c
#include "act.h"       // has to be included for all ACT++ applications
#include "types.h"
#include "ama.h"       // header file the Actuator Mechanism actors

extern ama system_ama; // AMA's defined globally

class Startup_Topology_T : public ACTOR {
public :
    void set_up_topology();
};

class Startup_Receptionist_R : public ACTOR {
    LIST_OF_ACTORS *topology;
    Mbox esa;   // environment specification actor
    int water_level;
    int pressure;
    int temperature;
    int wp_status;
    int prv_status;
    int pv_status;
    int gv_status;
    int time;
    int dsa_count;  // used by the receptionist behavior
    void update_topology_list();
    void garbage_collect_state(); // free up resources

public :
    void dispatch();
    void recv_sensor_values(); // receive sensor values from DSA's
    void recv_time();          // receive simulation time from DSA's
    void recv_actuator_states(); // receive actuator states from DSA's
```
void nominate_shutdown(); // the different replacements that can be
void nominate_prime_water_level(); nominated from this state
void nominate_startup_excess_pressure();
void nominate_startup_release_steam();
void nominate_startup_PV_open();
void nominate_normal();
}

class st_cs_wl_sga : public Actor { // goal actor for the water level parameter subsection
    Mbox cs_cfa; // collective failure acquaintance
public:
    void recv_acquaintances();
    void goal();
};

class st_cs_press_sga : public Actor { // goal actor for the pressure parameter subsection
    Mbox cs_cfa; // collective failure acquaintance
public:
    void recv_acquaintances();
    void goal();
};

class st_cs_temp_sga : public Actor { // goal actor for the temperature parameter subsection
    Mbox cs_cfa; // collective failure and policy acquaintances
    Mbox spa;
public:
    void recv_acquaintances();
    void goal();
};

class st_cs_temp_spa : public Actor { // policy actor for the temperature parameter subsection
    int lastok_time; // ama acquaintance is defined globally
    int curr_time;
public:
    void policy();
};

class st_cs_cfa : public Actor { // collective failures actor
    Mbox csa; // Controller Specification actor
public:
    void recv_acquaintances();
    void wl_failures();
    void press_failures();
};
void temp_failures();

};

class Startup_Environment_Spec_T : public ACTOR {
public:
    void set_up_topology();
};

class Startup_Environment_Spec_F : public ACTOR {
    LIST_OF_ACTORS *topology;
    Mbox csa; // controller specification actor
    void update_topology_list();
    void garbage_collect_state();
public:
    void go_into_shutdown();
    void nominate_startup_PV_open();
    void report_PRV_failure();
};

class Startup_es_pv_sga : public ACTOR {
    Mbox es_cfa;
public:
    void recv_acquaintances();
    void goal();
};

class Startup_es_prv_sga : public ACTOR {
    Mbox es_cfa;
public:
    void recv_acquaintances();
    void goal();
};

class Startup_es_cfa : public ACTOR {
    Mbox esa;
public:
    void recv_acquaintances();
    void PV_failure();
    void PRV_failure();
};
APPENDIX D - SOURCE FILE FOR STARTUP STATE

#include "normal.h"
#include "types.h"

void Startup_T::set_up_topology()
{
    Mbox cs_cfa = New(st_cs_cfa); // collective failures actor for this state;
    *cs_cfa << (METHOD)&st_cs_cfa::recv_acquaintances << MYMBOX;

    Mbox wl_spa = New (st_cs_wl_spa); // create the policy actors
    Mbox press_spa = New (st_cs_press_spa);
    Mbox temp_spa = New (st_cs_temp_spa);

    // Now start creating the goal actors
    Mbox wl_sga = New(st_cs_wl_sga);
    *wl_sga << (METHOD)&st_cs_wl_sga::recv_acquaintances << cs_cfa << wl_spa;

    Mbox press_sga = New(st_cs_press_sga);
    *press_sga << (METHOD)&st_cs_press_sga::recv_acquaintances << cs_fa << press_spa;

    Mbox temp_sga = New(st_cs_temp_sga);
    *temp_sga << (METHOD)&st_cs_temp_sga::recv_acquaintances << cs_cfa << temp_spa;

    become(New (Startup_Receptionsist_R)); // nominate the receptionist

    // communicate the topology
    self << (METHOD)&Startup_T::update_topology_list << wl_sga;
    self << (METHOD)&Startup_T::update_topology_list << press_sga;
    self << (METHOD)&Startup_T::update_topology_list << temp_sga;
    self << (METHOD)&Startup_T::update_topology_list << wl_spa;
    self << (METHOD)&Startup_T::update_topology_list << press_spa;
    self << (METHOD)&Startup_T::update_topology_list << temp_spa;
void Startup_Receptionist_R::update_topology_list(Mbox elem) {
  LIST_OF_ACTORS *curr_state_elem;

  curr_state_elem = state;
  while (curr_state_elem->next_elem != (LIST_OF_ACTORS *)NULL) {
    curr_state_elem = curr_state_elem->next_elem;
    curr_state_elem->next_elem = (LIST_OF_ACTORS *)new(LIST_OF_ACTORS);
    curr_state_elem = curr_state_elem->next_elem;
    curr_state_elem->elem = elem;
    curr_state_elem->next_elem = (LIST_OF_ACTORS *)NULL;
  }
}

void Startup_Receptionist_R::garbage_collect_state()
{
  LIST_OF_ACTORS *curr_state_elem;

  curr_state_elem = state;
  while (curr_state_elem != (LIST_OF_ACTORS *)NULL) {
    Free(curr_state_elem->elem); //Free frees the Mbox and the Actor
    curr_state_elem = curr_state_elem->elem;
  }
}

void Startup_Receptionist_R::dispatch()
{
  LIST_OF_ACTORS *curr_state_elem;

  if (dsa_count == 3) {       // check if the necessary values have been received
    curr_state_elem = state;
    *(curr_state_elem->elem) << (METHOD)&st_cs_wl_sga::goal << wl;
    curr_state_elem = curr_state_elem->next_elem;
    *(curr_state_elem->elem) << (METHOD)&st_cs_press_sga::goal << press
        << pv_stat;
    curr_state_elem = curr_state_elem->next_elem;
    *(curr_state_elem->elem) << (METHOD)&st_cs_temp_sga::goal << temp << pv_stat;
    dsa_count = 0;
  }
}
void Startup_Receptionist_R::recv_sensor_values()
{
    self >> temp >> press >> wl;
    ++dsa_count
    dispatch();      // can the values received from the DSA's be sent to the goal actors ??
    become(self);
}

void Startup_Receptionist_R::recv_time()
{
    self >> time;
    ++dsa_count
    dispatch();
    become(self);
}

void Startup_Receptionist_R::recv_actuator_states()
{
    self >> prv_stat >> wp_stat >> pv_stat >> gv_stat;
    ++dsa_count
    dispatch();
    become(self);
}

void Startup_Receptionist_R::nominate_shutdown()
{
    garbage_collect_state();
    become (New (Shutdown_Topology_T));
}

void Startup_Receptionist_R::nominate_prime_water_level()
{
    garbage_collect_state();
    become (New (PWL_Topology_T));
}

void Startup_Receptionist_R::nominate_startup_excess_pressure()
{
    garbage_collect_state();
    become (New (SEP_Topology_T));
}
void Startup_Receptionist_R::nominate_startup_release_steam()
{
garbage_collect_state();
become (New (SRS_Topology_T));
}

void Startup_Receptionist_R::nominate_normal()
{
garbage_collect_state();
become (New (Normal_Topology_T));
}

void Startup_Receptionist_R::nominate_startup_PV_open()
{
garbage_collect_state();
become (New (Startup_PVO_Topology_T));
}

void st_cs_wl_sga::recv_acquaintances()
{
self >> cs_cfa;
become(self);
}

void st_cs_wl_sga::goal()
{
int wl;

self >> wl;
if (wl != STARTUP_MIN_WL)
    *cs_cfa <<= (METHOD)&st_cs_cfa::wl_failures << wl;
become(self);
}

void st_cs_press_sga::recv_acquaintances()
{
self >> cs_cfa;
become(self);
}

void st_cs_press_sga::goal()
{
int press, pv_stat;
self >> press >> pv_stat; // receive from the Receptionist

if (press > 450)
    *cs_cfa << (METHOD)&st_cs_sfa::press_failures << press;
become(self);

void st_cs_temp_sga::recv_acquaintances()
{
    self >> cs_cfa >> spa;
become(self);
}

void st_cs_temp_sga::goal()
{
    int temp, gv_stat, pv_stat, time;
    self >> temp >> gv_stat >> pv_stat >> time; // receive from the Receptionist R

    if (((temp > 340) && (temp < 360))
        *spa << (METHOD)&st_cs_spa::policy << temp << gv_stat << time;
    else
        *sfa << (METHOD)&st_cs_cfa::temp_failures << temp << gv_stat
            << pv_stat << time;
become(self);
}

void st_cs_temp_spa::policy()
{
    int temp, gv_stat, time, excess_temp, out_of_range_time, new_gv_stat;
    self >> temp >> gv_stat >> time;

    if (temp < 350) {
        *gas_ama << (METHOD)&ama::recv_temp_policy << 80;
        lastok_time = time;
    }
    else
    {
        curr_time = time;
        excess_temp = temp - 350;
        out_of_range_time = curr_time - lastok_time;
        new_gv_stat = gv_stat - (excess_temp * out_of_range_time);
        new_gv_stat = (new_gv_stat < 0 ? 0 : new_gv_stat);
        new_gv_stat = (new_gv_stat > 100 ? 100 : new_gv_stat);
        *gas_ama << (METHOD)&ama::recv_temp_policy << new_gv_stat;
become(self);
}

void st_cs_cfa::recv_acquaintances()
{
    self >> csa;
    become(self);
}

void st_cs_cfa::wl_failures()
{
    int wl;
    self >> wl;

    if (wl > 85)
        *csa << (METHOD)&Startup_Receptionist_R::nominate_shutdown;
    else if (w > 50)
        *csa << (METHOD)&Startup_Receptionist_R::nominate_normal;
    else
        *csa << (METHOD)&st_s_seb::nominate_pwl;
    become(self);  // no more message will be processed - this state will be garbage collected
}

void st_cs_cfa::press_failures()
{
    int press;

    self >> press >> pv_stat;

    if (press > 500)
        *csa << (METHOD)&st_s_seb::nominate_sep;
    else if (press > 450)
        *csa << (METHOD)&st_s_seb::nominate_srs;
    become(self);  // no more message will be processed - this state will be garbage collected
}

void st_cs_cfa::temp_failures()
{
    int temp, gv_stat, pv_stat;

    self >> temp >> gv_stat >> pv_stat;
}
if (temp > 375) {
    printf("TEMPERATURE ALARM : Temperature is %d \n", temp);
}
become(self); // still within the same state
}

void Startup_Environment_Spec_T::set_up_topology() {

    Mbox es_cfa = New(Startup_es_cfa); // collective failures actor for this state;
    *es_cfa << (METHOD)&Startup_es_cfa::recv_acquaintances << MYMBOX;

    Mbox es_spa = New(Startup_es_spa); // create the policy actor

    // Now start creating the goal actors

    Mbox prv_sga = New(Startup_es_prv_sga);
    *prv_sga << (METHOD)&Startup_es_prv_sga::recv_acquaintances << es_cfa << es_spa;

    Mbox pv_sga = New(Startup_es_pv_sga);
    *pv_sga << (METHOD)&Startup_es_pv_sga::recv_acquaintances << es_cfa << es_spa;

    become(New(Startup_Environment_Spec_F)); // nominate the Failure behavior

    // communicate the topology

    self << (METHOD)&Startup_Environment_Spec_F::update_topology_list << prv_sga;
    self << (METHOD)&Startup_Environment_Spec_F::update_topology_list << pv_sga;
    self << (METHOD)&Startup_Environment_Spec_F::update_topology_list << es_spa;
    self << (METHOD)&Startup_Environment_Spec_F::update_topology_list << es_cfa;
}

void Startup_Environment_Spec_F::nominate_Startup_PV_Open() {

    *csa << (METHOD)&Startup_Receptionist_R::nominate_startup_pv_open;
    become(self);
}

void Startup_Environment_Spec_F::go_into_shutdown() {

    garbage_collect_state();
    become(New(Shutdown_Environment_Spec_T));
}

void Startup_Environment_Spec_F::report_PRV_failure() {

    printf ("\n\n PRV has failed !!! \n");
}
become (self);
}

void Startup_es_pv_sga::recv_acquaintances() {
    self >> es_cfa;
    become(self);
}

void Startup_es_pv_sga::goal() {
    Cbox pv_open_cbox;
    int status;

    *pv_ama << (METHOD)&ama::recv_pv_status << PV_CLOSED << pv_open_cbox;
    pv_open_cbox >> status;
    *es_cfa << (METHOD)&Startup_es_cfa::PV_failure;
    become(self);
}

void Startup_es_prv_sga::recv_acquaintances() {
    self >> es_cfa;
    become(self);
}

void Startup_es_prv_sga::goal() {
    Cbox prv_open_cbox;
    int status;

    *prv_ama << (METHOD)&ama::recv_prv_status << PRV_CLOSED << prv_open_cbox;
    prv_open_cbox >> status;
    *es_cfa << (METHOD)&Startup_es_cfa::PRV_failure;
    become(self);
}

void Startup_es_cfa::recv_acquaintances() {
    self >> esa;
    become(self);
}
void Startup_es_cfa::PV_failure() {
    *esa << (METHOD)&Startup_Environment_Spec_F::nominate_startup_PV_open;
become(New (SPVO_Environment_Spec_T));
}

void Startup_es_cfa::PRV_failure() {
    *esa << (METHOD)&Startup_Environment_Spec_F::report_PRV_failure;
become(self);
}
REFERENCES


VITAE

Vikul Khosla is currently a Master's degree candidate in Computer Science at Virginia Polytechnic Institute and State University. He received his Bachelor's degree in Mechanical Engineering from Birla Institute of Science and Technology, Pilani, India. His research interests are in the area of distributed systems, concurrent object-oriented systems and embedded applications.

He is affiliated with the student chapter of the Association of Computing Machinery.