

A SIMULATION-BASED SOFTWARE DEVELOPMENT
METHODOLOGY FOR DISTRIBUTED REAL-TIME SYSTEMS

By

Xiaolin Hu

Copyright © Xiaolin Hu 2004

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2004

THE UNIVERSITY OF ARIZONA®
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Xiaolin Hu

entitled A SIMULATION-BASED SOFTWARE DEVELOPMENT
METHODOLOGY FOR DISTRIBUTED REAL-TIME SYSTEMS

and recommend that it is accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Bernard P. Zeigler, Ph.D. _____
Date

Jerzy W. Rozenblit, Ph.D. _____
Date

Salim A. Hariri, Ph.D. _____
Date

Feiyue Wang, Ph.D. _____
Date

Young Jun Son, Ph.D. _____
Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director Bernard P. Zeigler, Ph.D. _____
Date

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: _____

ACKNOWLEDGEMENT

It hardly seems possible to reach the end of this long process without the support from many others, who have helped me in so many ways along the way.

First of all, I thank my advisor, Bernard P. Zeigler, for his mentoring and support on my research in the Ph.D. program. His insight to scientific research and the way to carry it out have greatly inspired me and will continue to guide me through my career path.

I thank the rest of my dissertation committee: Dr. Jerzy W. Rozenblit, Dr. Salim A. Hariri, Dr. Feiyue Wang, and Dr. Young Jun Son for their helpful comments, and suggestions during the course of my Ph.D. study. I especially thank Dr. Feiyue Wang, for his years of help and encouragement to my study and research. I also thank Dr. Hessam Sarjoughian, and Dr. Jerry Couretas. They are always available to discuss my research ideas and to help me to think through the problems.

I thank my colleagues in the ACIMS Lab: James Nutaro, Saehoon Cheon, Saurabh Mittal, Narayanaswami Ganapathy, Rajanikanth Jammalamadaka, Mahesh Veena, and Chungman Seo; and previous Lab members: Dr. Young Kwan Cho, and Dr. Sunwoo Park. We had a good time together.

Thank you, Yingying, for all the time spent with me.

–Xiaolin

To My Parents: Zejun Hu, and Qiuying Duan

TABLE OF CONTENTS

LIST OF FIGURES	10
LIST OF TABLES	12
ABSTRACT	13
CHAPTER 1 INTRODUCTION	15
1.1 Real-Time Systems.....	15
1.2 Research In Real-Time Systems.....	18
1.3 Computation Models For Real-Time Systems	23
1.4 Real-Time Software Development Methods.....	39
1.5 Simulation In System Development.....	33
1.6 Challenges Of Real-Time Software Development	35
1.7 Summary Of Contribution.....	39
1.8 Dissertation Organization	41
CHAPTER 2 DEVS FOR REAL-TIME SYSTEM DEVELOPMENT	43
2.1 DEVS As A Simulation-Based Design Framework	43
2.2 A Brief Review of DEVS Concepts	46
2.3 Modeling A System’s Behavior, Structure, And Timeliness Using DEVS	55
2.4 Connect to The Real Environment Through DEVS <i>activity</i>	60
2.3 Simulation and Execution of DEVS Models.....	61
CHAPTER 3 A MODEL CONTINUITY SOFTWARE DEVELOPMENT METHODOLOGY	67
3.1 Model Continuity in Software Development.....	67
3.2 Modeling, Simulation, Execution and Model Continuity.....	70
3.2.1 Model Continuity for Non-Distributed Real-Time Systems.....	70
3.2.2 Model Continuity for Distributed Real-Time Systems.....	74

TABLE OF CONTENTS – *Continued*

3.3	Simulation-Based Test for Real-Time Systems.....	77
3.3.1	A Virtual Test Environment.....	77
3.3.2	Incremental Simulaiton and Test for Non-Distributed Real-Time Systems.	78
3.3.3	Incremental Simulaiton and Test for Distributed Real-Time Systems	81
3.4	The Development Process of the Methodology	85
3.5	<i>abstractactivity</i>	88
3.5.1	<i>activity</i> and <i>abstractactivity</i>	88
3.5.2	Interface Between an Atomic Model and Activity	91
3.5.3	Interface Between <i>abstractactivity</i> and Environment Model	93
3.5.4	Impelment <i>activity</i> 's User-Defined Functions in <i>abstractactivity</i>	97
3.5	Network Delay Model and <i>addcouplingwithdelay()</i>	99
CHAPTER 4 VARIABLE STRUCTURE MODELING		102
4.1	System Evolution and Adaptive Computing.....	102
4.2	Conceptual Development for Variable Structure In DEVS.....	104
4.2.1	Variable Structure Modeling.....	105
4.2.2	Operation Boundaries	106
4.2.3	Changing Port Interface	108
4.3	Examples of Variable Structure.....	109
4.3.1	Dynamically Emulate the System Entity Structure (SES).....	110
4.3.2	A Refonfigurabe Workflow System	111
4.4	Implementation of Variable Structure In DEVS	114
4.4.1	Hierarchical Structure of DEVS Models and Their Simulators.....	115
4.4.2	Add/Remove Coupling Dynamically.....	118
4.4.3	Add/Remove Model Dynamically	119
4.4.4	Add/Remove Coupling In Distributed Environment	124
4.4.5	Add/Remove Ports	127

TABLE OF CONTENTS – *Continued*

CHAPTER 5	DISTRIBUTED AUTONOMOUS ROBOTIC SYSTEM – A DYNAMIC TEAM FORMATION EXAMPLE	129
5.1	Distributed Autonomous Robotic Systems	129
5.2	Hardware Description Of The Acims Robot.....	131
5.3	The Dynamic Team Formation Process	134
5.4	Developing Models of the System.....	135
5.4.1	System Model and Dynamic Coupling Change.....	136
5.4.2	Robot Model	138
5.4.3	Hardware Interface <i>activity</i>	141
5.4.4	Environment Model and <i>abstractactivity</i>	144
5.5	Stepwise Simulation, Deployment, and Execution	147
5.5.1	Step 1: Central Simulation.....	148
5.5.2	Step 2: Distributed Simulation.....	149
5.5.3	Step 3: Robot-In-The-Loop Simulation.....	150
5.5.4	Step 4: Real System Test	153
5.5.4	Deployment and Execution.....	154
5.6	Results and Discussion	155
CHAPTER 6	A SCALABLE ROBOTIC CONVOY SYSTEM	159
6.1	A Description of the Robot Convoy System	159
6.2	Models of the Robot Convoy System.....	161
6.2.1	System Model	161
6.2.2	Robot Model	162
6.3	Test the System In a Virtual Test Environment	168
6.4	A Study of Formation Coherence.....	171
CHAPTER 7	A HIGH PERFORMANCE SIMULATION ENGINE FOR LARGE-SCALE SYSTEMS.....	177

TABLE OF CONTENTS – *Continued*

7.1	Simulation of Large-Scale Systems.....	177
7.2	The Standard DEVS Simulation Protocol.....	179
7.3	An Proposed Improved Simulation Engine.....	182
7.4	The New Simulation Engine and its Data Structure.....	183
7.4.1	The Simulation Protocol.....	183
7.4.2	The <i>minSelTree</i> Data Structure.....	184
7.4.3	Building <i>minSelTree</i> for Two-Dimension Cellular Space Models.....	189
7.5	Performance Analysis.....	190
7.6	Examples and Test Data.....	192
7.6.1	Simulation of a Diffusion Model.....	192
7.6.2	Simulation of a Fire Spreading Model.....	194
CHAPTER 8 CONCLUSION AND FUTURE WORKS.....		196
8.1	Conclusions.....	196
8.2	Future Work.....	199
REFERENCES.....		202

LIST OF FIGURES

Figure 1.1	Research in real-time systems	19
Figure 1.2	A Timed Automata Example.....	28
Figure 1.3	Capsule, port, and protocol in UML-RT	31
Figure 1.4	Connector in UML-RT	32
Figure 2.1	Basic Entities and Relations	47
Figure 2.2	Discrete event time segments	48
Figure 2.3	Interpretation of the DEVS structure.....	51
Figure 2.4	A “leader-follower” system modeled by a DEVS coupled model	56
Figure 2.5	Timed state diagram of a screen saver program.....	57
Figure 2.6	Time patterns modeled in DEVS.....	59
Figure 2.7	DEVS model, activity, and the external environment.....	61
Figure 2.8	Simulate/Execute DEVS model in centralized and distributed environment. 62	
Figure 2.9	Hierarchical distributed simulation or execution topology	64
Figure 2.10	Model mapping.....	65
Figure 3.1	Modeling, Simulation and Execution of Non-distributed Real-time System. 71	
Figure 3.2	Modeling, Simulation and Execution of Distributed Real-time System	75
Figure 3.3	Step-wise Simulations of Non- distributed Real-time System.....	79
Figure 3.4	Simulation-based test of Distributed Real-time System.....	82
Figure 3.5	Development Process of the Methodology.....	86
Figure 3.6	Environment, models, activity, and abstractActivity	89
Figure 3.7	“Hierarchical” coupling between abstractActivity and environment model..	94
Figure 3.8	The addCouplingWithDelay() method	99
Figure 4.1	A variable structure process	106
Figure 4.2	Dynamically Emulate the System Entity Structure (SES)	110
Figure 4.3	Stages of the Reconfigurable GPT System	112
Figure 4.4	Relationship between models and their simulators (fast-mode simulation case)	
	116

LIST OF FIGURES - *Continued*

Figure 4.5	Methods and data structures used in variable structure implementation.....	116
Figure 4.6	Models and their simulators in distributed simulation	125
Figure 5.1	The ACIMS Mobile Robot.....	131
Figure 5.2	Model of Multi-Robot System.....	136
Figure 5.3	Model of Robot.....	139
Figure 5.4	Environment Model.....	144
Figure 5.5	Simulation-based test of the “team formation” system	147
Figure 5.6	Simulation of robots	156
Figure 5.7	Execution of robots.....	157
Figure 5.8	A Scalable Dynamic Team Formation Example	158
Figure 6.1	System Model of the Scalable Robotic System.....	161
Figure 6.2	Robot Model.....	163
Figure 6.3	State charts of robots’ Convoy models.....	165
Figure 6.4	Environment Model.....	168
Figure 6.5	Using network delay model in central simulation.....	170
Figure 6.6	10 robots in central real time simulation	170
Figure 6.7	Real robots for demonstration	171
Figure 6.8	Moving trails of robots during a simulation.....	173
Figure 6.9	Position errors of the robotic convoy system	175
Figure 6.10	Formation coherence as criteria for sensor capabilities obtained via simulation.....	176
Figure 7.1	The Standard DEVS Simulation Protocol	180
Figure 7.2	Model, simulator, and the minSelTree	185
Figure 7.3	A two-dimension cellular space model	189
Figure 7.4	Simulation time of coordinator and oneDcoord.....	193

LIST OF TABLES

Table 7.1 Comparison of the two simulation engines when simulating a one-dimensional diffusion model	193
Table 7.2 Comparison of the two simulation engines when simulating a two-dimensional fire spread model	195

ABSTRACT

Powered by the rapid advance of computer, network, and sensor/actuator technologies, distributed real-time systems that continually and autonomously control and react to the environment have been widely used. The combination of temporal requirements, concurrent environmental entities, and high reliability requirements, together with distributed processing make the software to control these systems extremely hard to design and difficult to verify.

In this work, we developed a simulation-based software development methodology to manage the complexity of distributed real-time software. This methodology, based on discrete event system specification (DEVS), overcomes the “incoherence problem” between different design stages by emphasizing “model continuity” through the development process. Specifically, techniques have been developed so that the same control models that are designed can be tested and analyzed by simulation methods and then easily deployed to the distributed target system for execution. To improve the traditional software testing process where real-time embedded software needs to be hooked up with real sensor/actuators and placed in a physical environment for meaningful test and analysis, we developed a virtual test environment that allows software to be effectively tested and analyzed in a virtual environment, using virtual sensor/actuators. Within this environment, stepwise simulation methods have been developed so that different aspects, such as logic and temporal behaviors, of a real-time system can be tested and analyzed incrementally.

Based on this methodology, a simulation and testing environment for distributed autonomous robotic systems is developed. This environment has successfully supported the development and investigation of several distributed autonomous robotic systems. One of them is a “dynamic team formation” system in which mobile robots search for each other, and then form a team dynamically through self-organization. Another system is a scalable robot convoy system in which robots convoy and maintain a line formation in a coordinated way.

CHAPTER 1

INTRODUCTION

1.1 Real-Time Systems

Fueled by Moore's law of exponentially expanding computational and networking infrastructure, and the rapid advance of sensor and actuator technologies, real-time systems that continually and autonomously control and react to the environments have been widely used. These systems are most frequently found in telecommunications, aerospace, defense, and automatic control applications. Because of their wide applications as well as their distinct properties such as timeliness and dependability¹, real-time systems begin to draw more and more research attention from both industrial developers and academic researchers.

Many definitions exist for real-time systems. Different definitions may emphasize different aspects of the nature of a real-time system. The most common definitions emphasize the *timeliness* of a real-time system. For example, as defined in [Phi97], [Kri97]: *a real-time system is one whose logical correctness is based on both the correctness of the outputs and their timeliness*. Other definitions emphasize both timeliness and safety [Phi97]: *A real-time system is a system that must satisfy explicit (bounded) response-time constraints or risk severe consequences, including failure*. In the context of this dissertation, the definition provided by [Sha01] is used which emphasizes the interaction between a system and its environment: *Real-time systems are*

¹ The term *dependability* is defined as the trustworthiness of a system such that reliance can justifiably be placed on the service it provides [Lap92].

computer systems that monitor, respond to, or control, an external environment. This environment is connected to the computer system through sensors, actuators, and other input-output interfaces.

Real time systems have existed ever since the inception of computers. While these systems in the past were mainly applied to process control and restricted by primitive hardware devices, today's real-time systems exhibit complexity and scalability hundreds, if not thousands, times higher than before. These changes are mostly driven by the increasingly complex tasks that need to be accomplished and the increasingly challenging environments that need to be dealt with. Among them the most salient change would be that today's real-time systems are more and more networked together to finish system-wide tasks. As a result, distributed real-time systems are formed in which subcomponents of the systems, though are physically distributed, coordinate and cooperate together to finish common tasks.

Generally speaking, characteristics of a typical modern real-time system include:

- They are often geographically distributed;
- They may contain very large and complex software components;
- They must interact with concurrent real-world entities;
- They may contain processing elements that are subject to the constraints of computation resources (such as memory, CPU, network speed), cost, size, etc.

The one common feature of all real-time software systems is timeliness, that is, the requirement to respond correctly to inputs within acceptable time intervals. This property characterizes a vast spectrum of different types of systems ranging from purely time-driven to purely event-driven systems, from soft real-time systems to hard real-time systems, and so on. For these systems, a systematic time handling and time modeling approach will benefit both systems' design and verification. It also follows from the very nature of most real-time applications that there is a stringent requirement for high reliability. This can be formulated as a need for dependability and safety. To give high levels of reliability requires fault-tolerant hardware and software, and demands effective test methods and techniques during systems' development. Besides timeliness and reliability, real-time systems also have a concurrent event-driven nature, and are characterized by their continuous interaction with the environment; therefore they are sometimes called "reactive systems" [Ben91] [Hal93]. The combination of temporal requirements, limited resources, concurrent environmental entities and high reliability requirements (together with distributed processing) presents the system engineer with unique problems. Real-time systems are now recognized as a distinct discipline. It has its own body of knowledge and theoretical foundation.

Real-time systems are often embedded, meaning that the computational system exists inside a larger system, with the purpose of helping that system to achieve its overall responsibilities. For this reason, real-time systems and embedded systems are usually referred together as real-time embedded systems. However, in the context of this dissertation, more emphasis is put on the "real-time" aspects of a system, while less

emphasis being put on the “embedded” aspects such as limited CPU, memory, and power resources. Specifically, the work of this dissertation focuses on a major category of real-time systems that are characterized as complex, event-driven, and distributed. Such systems are most frequently encountered in telecommunications, aerospace, defense, and automatic control applications. The size and complexity of these systems demand a considerable development effort, typically involving large development teams, that is followed by an extended period of evolutionary growth [Sel98]. It is also important to point out that although the development of real-time systems involves work on both software and hardware, this dissertation mainly focuses on the software development of real-time systems.

1.2 Research in Real-time Systems

There has been tremendous research work under the umbrella of real-time systems. These research works cover different aspects of real-time systems, from low level hardware interfaces to system level design methodologies, from theoretic computation models to practical code generation and execution, etc. Figure 1.1 categorizes these research works into three layers (groups). From the bottom to the top, they are marked as *Technologies*, *Models*, and *Methods* respectively. This section gives a brief description of each of these research works. As the goal of this section is to provide a general picture for the research works of real-time systems, readers who are interested in any specific subjects are recommended to check the referred papers or books for more details.

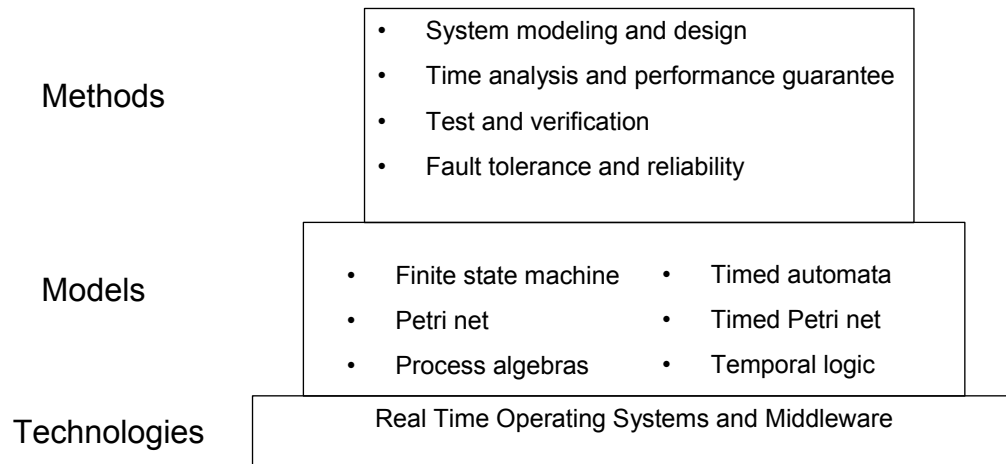


Figure 1.1: Research in real-time systems

Included in the bottom layer are real-time operating systems and middleware. They represent the infrastructure technology, most driven by commercial industry developers, which has been developed to facilitate the execution and communication of real-time systems. The main goal of any real-time operating system is to provide fast, predictable and concurrent services such as fast response to interrupts and predictable scheduling algorithms to the programs running above it. These specialized operation systems are often stripped-down versions of traditional timesharing operating systems which are made appropriate for the real-time domain [Gho94]. An essential difference, due to the distinguished nature of the real-time applications, is that the external events and activities which must be delivered have a hard deadline. As such, fairness, guaranteed by the time-slicing, is replaced by event-triggered (ET) or timer-triggered (TT) scheduling policies, which are better suited for coping with real-time [Tho00]. Other important features of real-time OS include scalable kernel structure and fault tolerance, etc. One of the main research efforts in real-time OS and middleware are real-time scheduling algorithms.

Some of the most well known real-time scheduling algorithms are *Rate Monotonic (RM)* scheduling [Lei80], *Earliest Deadline First (EDF)* scheduling [Zha87], *Minimum-Laxity-First (MLF)* scheduling [Der89] and *Maximum-Urgency-First (MUF)* scheduling [Ste91].

Included in the middle layer are computation models that are widely used in the design, analysis, and implementation of real-time software. Formal computation models for real-time systems have received growing attentions in the recent years. A formal model is an essential ingredient of a sound system-level design methodology because it makes it possible to capture the required functionality, verify the correctness of the functional specification and synthesize the specification tool-independently [Sgr00], [Mar01]. As timeliness is an important feature in real-time embedded systems, computation models can be characterized into two categories: model not considering time such as finite state machine, petri net, process algebra; model considering time such as timed automata, timed petri net, temporal logic. These computation models provide the basis to capture the behavior and structure of a system under development. Those models considering time also capture the timeliness feature of the system. They support time modeling explicitly so are naturally fitted into the real-time domain. The next section will discuss some of these computation models in more detail.

Included in the top layer are methods (methodologies) proposed for real-time system development. These methods are further categorized into four groups: *system modeling and design*, *time analysis and performance guarantee*, *test and verification*, *fault tolerance and reliability*. The research work of *system modeling and design* focuses on providing systematic modeling methodologies and design tools (based on one or several

computation models) to facilitate modelers/designers to capture the properties of a system under development. For example, the real-time UML of Object Management Group (OMG), although still in its request for proposals (RFP) stage, seeks a standard way of modeling real-time systems using Unified Modeling Language (UML) [OMG1], [OMG2], [OMG3], [OMG4]. Besides the capabilities to model a system's behavior and structure, most modeling languages for real-time systems also provide the capability for (explicitly or implicitly) time modeling.

The research work of *time analysis and performance guarantee* focuses on the development of techniques to analyze and guarantee scheduled tasks to meet their deadlines. Specifically, time analysis refers to timing and schedulability analysis, meaning how to analyze the system to check if the timeliness and deadline requirement has been met. There has been a lot of research work in this area, especially for the hard real-time systems. Most of them are focus on the time driven systems. The timeliness requirements of such time-driven software then correspond to ensuring that given a set of tasks, on a given platform, all tasks can be executed at the specified rates, while meeting the deadlines. Some frequently used notions when conducting time analysis are *worst-case response time*, *end-to-end timing constraints*. For event driven systems, not much work has been done because of the inherit unpredictability of the system state. However, [Sak99][Sak00] suggest that it is possible to do schedulability analysis for event-based system by defining concepts such as *transaction*, *time critical path* and *priority event*. While time analysis techniques analyze the schedulability of a system, performance guarantee focuses on the development of techniques to guarantee scheduled tasks to meet

their deadlines. Generally speaking, the variety of the nature of systems and their execution environments ask for various scheduling techniques that are most suitable to a specific system. For example, missing a deadline is unacceptable for hard real-time systems, but acceptable for soft real-time systems. Among the techniques for performance guarantee, QoS and imprecise computation [Nat95] are two techniques that gained a lot of research interest recently.

The research work of *test and verification* focuses on the techniques and methods to test or verify the system under development. Typically, there are two main disciplines: formal methods and simulation based methods. Each of them has unique contributions to the verification effort. Formal methods, such as model checking, are mathematically based languages and techniques for specification and verification of complex software and hardware systems. The simulation-based methods verify the design by generating test input and running simulations to check if desired results are reached.

The research work of *fault tolerance and reliability* focuses on the techniques for a system to handle abnormal situations. These techniques include the techniques to enable the system to continue its function, maybe in a degraded mode, under abnormal conditions; the techniques to enable a system to recover itself when part of the system has malfunctions, and the techniques to analyze or evaluate the reliability, availability, or safety of the system under development [avi75, avi76].

While the purpose of Figure 1.1 is to characterize the related work into different groups, it doesn't mean to separate the relationship between them, nor does it represent all the research work in real-time systems. In fact, the research work of real-time system

is so broad that it almost covers every aspect of the general theories of computing systems. These research efforts are interwoven together and complementary to each other. For example, a real-time operating system may apply concepts from finite state machine theory; a system modeling methodology might be heavily dependent on the underline RTOS support.

1.3 Computation Models for Real-time Systems

Computation models form the basis for any formal methodologies or processes. This section gives an informal description to several computation models. They are *finite state machine*, *Petri nets*, *timed automate*, and *temporal logic*. These models are chosen here because they are most relevant to the research work of this dissertation, which is based on the DEVS (Discrete Event System Specification) formalism.

Finite State Machine

Finite State Machines (FSM) [Gil62], also known as Finite State Automation (FSA), at their simplest, are models of the behaviors of a system or a complex object, with a limited number of defined conditions or modes, where mode transitions change with circumstance. Finite state machines consist of four main elements:

- *States* which define behavior and may produce actions
- *State transitions* which are movement from one state to another
- *Rules or conditions* which must be met to allow a state transition

- *Input events* which are either externally or internally generated, which may possibly trigger rules and lead to state transitions

FSM is typically used as a type of control system where knowledge is represented in the states, and actions are constrained by rules. Like any rule-based systems, if all the antecedent(s) of a rule are true, then the rule is triggered. It is possible for multiple rules to be triggered, and in the area of reasoning systems, this is called a conflict set. There can only be one transition from the current state, so a consistent conflict resolution strategy is required to select only one of the triggered rules to fire and thus performing a state transition. This brings us to two main types of FSM. The original simple FSM is what's known as deterministic, meaning that given an input and the current state, the state transition can be predicted. An extension on the concept at the opposite end is a non-deterministic finite state machine. This is where given the current state; the state transition is not predictable. It may be the case that multiple inputs are received at various times, means the transition from the current state to another state cannot be known until the inputs are received (event driven).

There are two main methods for handling where to generate the outputs for a finite state machine. They are called a Moore Machine and a Mealy Machine, named after their respective authors. A Moore Machine is a type of finite state machine where the outputs are generated as products of the states. A Mealy Machine, unlike a Moore Machine is a type of finite state machine where the outputs are generated as products of the transition between states.

Finite state machines is not a new technique, it has been around for a long time. There are a number of abstract modeling techniques that may help or spark understanding in the definition and design of a finite state machine, most come from the area of design or mathematics.

- State Transition Diagram: also called a bubble diagram, shows the relationships between states and inputs that cause state transitions.
- State-Action-Decision Diagram: simply a flow diagram with the addition of bubbles that show waiting for external inputs.
- Statechart Diagrams: a form of UML notation used to show behavior of an individual object as a number of states, and transitions between those states [Bru00].
- Hierarchical Task Analysis (HTA): though it does not look at states, HTA is a task decomposition technique that looks at the way a task can be split into subtasks, and the order in which they are performed [Dix98]

Petri Nets

Petri nets, or place-transition nets, are classical models of concurrency, non-determinism, and control flow, first proposed by Carl Adam Petri in 1962. A Petri net is a graphical and mathematical modeling tool. It consists of *places*, *transitions*, and *arcs* that connect them. *Input arcs* connect places with transitions, while *output arcs* start at a transition and end at a place. There are other types of arcs, e.g. *inhibitor arcs*. Places can contain *tokens*. The current state of the modeled system (the *marking*) is given by the

number (and type if the tokens are distinguishable) of tokens in each place. Transitions are active components. They model activities that can occur (the transition *fires*), thus changing the state of the system (the marking of the Petri net). Transitions are only allowed to fire if they are *enabled*, which means that all the preconditions for the activity must be fulfilled (there are enough tokens available in the input places). When the transition fires, it removes tokens from its input places and adds some at all of its output places. The number of tokens removed / added depends on the cardinality of each arc.

Petri nets are a promising tool for describing and studying systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. As a graphical tool, Petri nets can be used as a visual-communication aid similar to flow charts, block diagrams, and networks. In addition, tokens are used in these nets to simulate the dynamic and concurrent activities of systems. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behavior of systems.

The major weaknesses of Petri nets are: (1) to model the notion of time, it is not straightforward [Rem93]; (2) as the system size and complexity evolve, the state-space of the Petri net grows exponentially, which could become too difficult to manage both graphically and analytically; (3) control logic is hard-wired, i.e. inflexible to cope with system change. A lot of research has been carried out in order to tackle, in particular the first two weaknesses. For example, Timed Petri net was proposed to enrich the modeling power of ordinary Petri nets by incorporating the notion of time. Dealing with time in PN is accomplished by assigning time elapses to the transition, to the places, or even to the

tokens and arcs. An important research area to manage the complexity of system modeling using Petri nets is Petri net synthesis. The major idea behind the method is to build a complex model through systematically synthesizing some well-defined Petri-net modules. Researchers in this area endeavor to provide the theories and methodologies for preserving the system properties during net synthesis [Zho93].

Timed Automata

A timed automaton is a finite automaton with a finite set of real-valued clocks. The clocks can be reset to 0 (independently of each other) with the transitions of the automaton, and keep track of the time elapsed since the last reset. The transitions of the automaton put certain constraints on the clock values: a transition may be taken only if the current values of the clocks satisfy the associated constraints. With this mechanism we can model timing properties such as “the channel delivers every message within 3 to 5 time units of its receipt.” Timed automata can capture several interesting aspects of real-time systems: qualitative features such as liveness, fairness, and nondeterminism; and quantitative features such as periodicity, bounded response, and timing delays [Alu94].

Figure 1.2 gives an example of timed automaton. It has two states and two clocks x and y . Suppose it starts operating the configuration $(p, 0, 0)$ (the two last coordinates denote the values of the clocks). Below we give a brief description of how this timed automaton works based on a run over a timed word $(a, 3.2) \rightarrow (c, 5.1) \rightarrow (b, 8.2) \dots$ as shown in the diagram. When the automaton stays at p , the values of the clocks grow. At

time 3.2 ($x=3.2$; $y=3.2$), the condition $y < 4$ (the guard of the transition from p to q) is satisfied and the automaton can move to q while firing action a and resetting x to 0. Thus the configuration of the automaton becomes $(q, 0, 3.2)$. As time increases, at time 5.1, the automaton fires action c and reset the clock y to 0 (there is no guard, or the guard is always true, for this transition). As the automaton returns to state q , its new configuration after this transition is $(q, 1.9, 0)$. Then at time 8.2, the condition $x=5$ (the guard of the transition from q to p) is satisfied and the automaton moves to p and fires action b . This makes the configuration of the automaton become $(p, 5, 3.1)$.

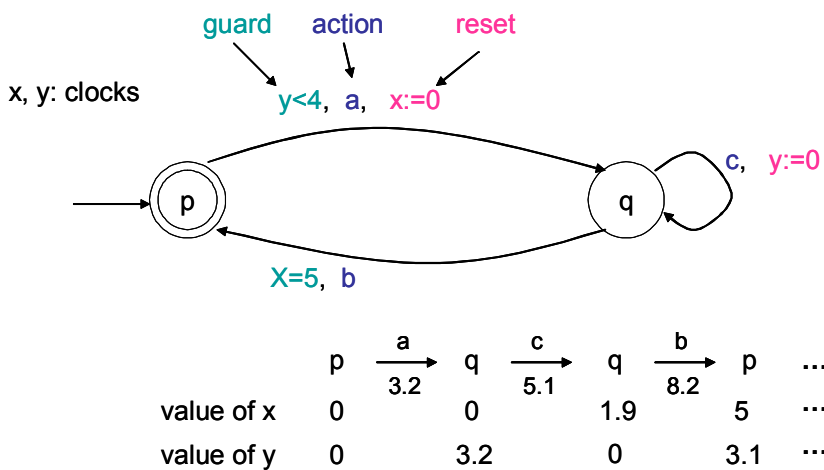


Figure 1.2. A Timed Automata Example

Temporal Logic

The term Temporal Logic [Pri57], [Pri67], [Pri69] has been broadly used to cover all approaches to the representation of temporal information within a logical framework. It is also more narrowly to refer specifically to the modal-logic type of approach introduced around 1960 by Arthur Prior under the name of Tense Logic and subsequently developed

further by logicians and computer scientists. Temporal logic is an extension of conventional (propositional) logic which incorporates special operators that cater for time. With Temporal Logic one can specify and verify how components, protocols, objects, modules, procedures and functions behave as time progresses. The specification is done with (Temporal) logic statements that make assertions about properties and relationships in the past, present, and the future.

Classical temporal logics deal with time in a qualitative way, which makes them not suitable to deal with real-time systems. Qualitative properties like safeness, liveness and fairness can be investigated. To overcome this drawback, extensions were proposed where time is treated in a quantitative way [Fro95]. The different approaches are: (1) considering time as a derived concept, on the basis of the “next”-operator, which allows specifications to deal with a concrete time domain; (2) introduce an explicit clock variable to function; and (3) bounding the temporal operators. In these “real-time temporal logics”, quantitative properties like periodicity, real-time response (deadlines) and delays can be defined.

Temporal Logics often find their application in the specification of real-time properties in model-checking verification. Often, they are also used as an engine to formulate and make deductive proofs in the formal verification of (timing) properties. As Temporal Logic represents temporal information within a logical framework, it allows formal and automated verification and checking. For example, temporal logic has been used to supplement Java Assertions for the purpose of testing [Eri].

1.4 Real-time Software Development Methods

Various real-time software development methodologies have been developed. Typically at the heart of a methodology lies the system modeling specification, which is based on one or more computation models. For example, the HW/SW co-design methodology described in [Tho00] uses the Multi-Thread Graph (MTG) model for system modeling; the Ptolemy II [Lee01], [Pto] method supports heterogeneous models such as discrete event, data flow, finite state machines and so on for system modeling; the Real-Time Object-Oriented TMO method [Kim97] uses the Time-triggered Message-triggered Object (TMO) model for system modeling. Several other models and design methods for real-time systems have been surveyed by Gomaa [Gom93], [Gom00]. Based on the modeling specification, techniques to analyze, design, test, and synthesize the software are developed. Some methods, especially those that are commercially supported by industry companies, also provide highly integrated developing environment (IDE) and CAD tools. These environments and tools aim to automate the development process, thus greatly speeding up the development time.

Ideally, a methodology should span across different abstraction levels, supporting the full path from system-level specification down to code implementation. For real-time systems, it is also desirable for the specification to support timeliness modeling explicitly and systematically. Other desired features of a specification include supporting modularity for model reuse; allowing implementation independent specification to enhance portability; incorporating formal model-based modeling to enable automated model checking and synthesis; etc. The current state of art is that most methods only

support some, not all, of these features. Below we use UML-RT² as a case study example [Bia03] to see what kinds of features UML-RT supports.

A Brief Introduction to UML-RT

It is well known that UML [OMG5] has achieved a great popularity in software development. This is because UML is a semi-formal notation relatively easy to use and well supported by tools. Recently, the deficiencies of standard UML as a vehicle for complete specification and implementation of real-time embedded systems has led to a variety of competing and complementary proposals [Mar01]. One of the major attempts is the UML-RT profile, which derives from ObjectTime's ROOM methodology [Sel94]. It is likely that OMG will include UML-RT features in the definition of UML 2.0.

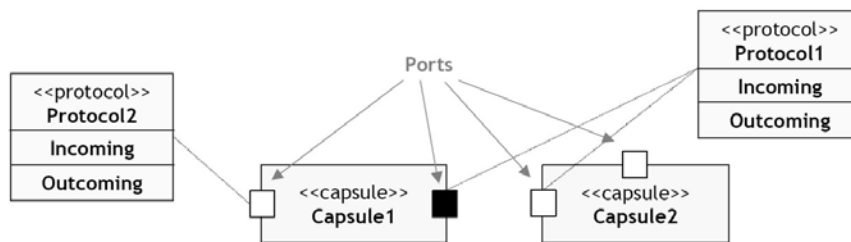


Figure 1.3: Capsule, port, and protocol in UML-RT

UML-RT is an extension of UML that addresses real-time issues. It provides a formalism to handle active objects. An active object is called a *Capsule* in UML-RT and it communicates with other capsules through asynchronous messages, which are sent and received through *Ports*. A *Port* is defined by a *Protocol* that defines which messages can

² The UML-RT is chosen here because it is directly related to the DEVS modeling approach which is the basis of this dissertation. As a matter of fact, the concepts in UML-RT: Capsule, Port, Connector directly correspond to the Model, Port, and Coupling in DEVS respectively.

be sent through a port (Out messages) and which messages a port accepts (In messages). Given a *Protocol*, its conjugate is always defined, by simply inverting the In and Out messages. This is shown in Figure 1.3.

The *State Diagrams* associated to each *capsule* have the usual syntax and semantic as in plain UML, including the “run to completion” behavior [OMG5]. The only additional constraint is that any message (except internal messages, which remain in the boundaries of the *State Diagram*) has always to be referred to a *Port*. As shown in Figure 1.4, *capsules* are connected through *Connectors*. A *Connector* binds two different *Ports* with compatible *Protocols*. A protocol is always compatible with its conjugate; a protocol to be compatible with another one has to accept as In messages a superset of the other protocol’s Out messages and has to send a subset of the messages accepted by the other protocol.

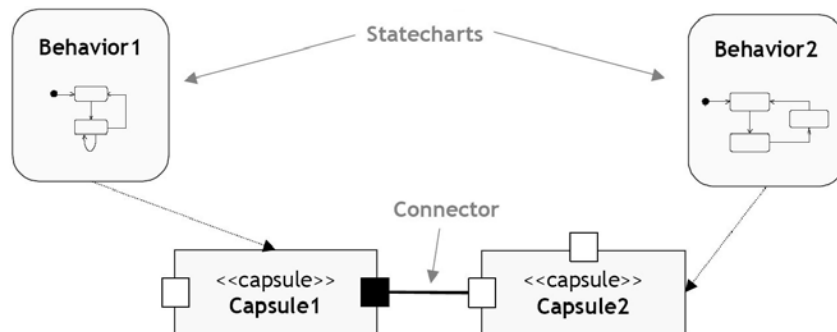


Figure 1.4: Connector in UML-RT

UML-RT mainly focuses on the concept of active component (the *Capsule*) and it doesn’t directly address real-time constraints. The concept of time can be found in standard UML-RT libraries –mainly thanks to the *Timer* class stereotype– but not directly in the modeling language.

UML-RT is implementation oriented: it is conceived to be used with a complete library in a language of choice, usually abstracting from the underneath platform. By embedding code fragments in transitions and states (as defined in plain UML) a UML-RT model can be directly translated to code (Rational Rose RealTime being the reference tool for generating working embedded distributed systems from UML-RT models).

UML-RT is an effective notation for the design and implementation of systems. However, the application of UML-RT to the real-time domain is still suffering from several problems [Bia03]. First, UML-RT is not formally well defined. Very often real-time applications are also safety-critical, and thus call for activities like the verification of properties (such as safety, utility, liveness, ...), the simulation of the system, the generation of test cases, etc. It is very hard (if at all possible) to carry out such activities when the specifications are written in semi-formal notations like UML or UML-RT. Secondly, time issues (i.e., the representation of time and time constraints) are not treated at a native level: ad-hoc components (like timers) have to provide time-related information to the system.

1.5 Simulation in System Development

Simulation technologies have been widely used in industry to assist system development. It can offer significant savings in time and resources over system development; revolutionize traditional design evaluation methods and overcome a number of safety, environmental and security constraints.

Typically, simulation tools are used at two stages of system development: the analysis stage to support concept development, virtual prototyping, etc.; the test stage to provide test environments and test cases for system verification and evaluation. By applying simulation technology in the analysis stage, simulation-based design can highlight problems early enough in the product development process, where they may be addressed more cost-effectively. Engineers have also achieved a measure of success with simulation-based virtual prototyping. Many leading manufacturers, among them Boeing, Chrysler, and General Dynamics, have saved millions of dollars on fighter planes, automobiles, and submarines by replacing physical prototypes with computer mock-ups [Mic98]. Simulation-based test and verification enable automated test program and test cases generation, functional coverage and checking, etc. It has been widely used, although still in an ad-hoc way, by both hardware and software developers. For example, test generation techniques, tools, and solutions are widely recognized as the main means for hardware verification of complex designs. The approach of using simulation-based software design and implementation combined with hardware-in-the-loop simulation techniques greatly accelerate the embedded software development and integration processes. Effective use of these techniques results in a faster product development cycle, lower development costs, and higher overall product quality.

Simulation Based Acquisition

Simulation Based Acquisition (SBA) [SBA] is an acquisition process in which DoD and industry are enabled by robust, collaborative use of simulation technology that is

integrated across acquisition phases and programs. SBA is defined as the integrator of simulation tools and technology across acquisition functions and program phases. It advocates the usage of simulation across all the phases of a system's development. Furthermore, it advocates using M&S as early as possible such as to start the development process with the training simulation/simulator as the virtual prototype and defer formal system operational requirements until the virtual prototype is built. This might be a new trend for large-scale complex system development.

SBA envisions an integrated approach to the use of M&S in systems acquisition. The application of the simulation tools has been stove-piped in many cases as the tools available were designed for the immediate intended purpose and did not address other life cycle phases. An integrated set of simulation tools, across the phases, permits analyses to address the full implications of competing concepts including initial effectiveness analyses, man/machine interfaces, tactics, techniques and procedures on a combined arms battlefield, and analysis of concept implications to manufacturing, reliability and supportability. The philosophy of an early use of M&S, designed with the entire life cycle in mind, provides the benefit of reusing an expanding, validated, consistent, and up to date, and common M&S infrastructure.

1.6 Challenges of Real-time Software Development

Although the hardware capabilities for real-time systems have been improved greatly, the implementation of their functionalities has steadily shifted to the software. This is driven by the fact that software has much more flexibility to cope with system varieties

and requirement changes. Recent studies indicate that up to 60% of the development time of an embedded real-time system is spent in software coding [Mor96], [Pau96], [Pau97]. Clearly this is a sign showing the importance of real-time software development. However, it is also a sign indicating that the existing software development methods are insufficient to develop real-time systems. As a matter of fact, the lack of good design methods and support tools has made the software development for real-time systems a bottleneck, especially when a large number of subsystems and task synchronization are involved.

The real-time software developer faces several unique challenges beyond those of classical software development. First, as mentioned before, real-time systems need to meet both timeliness and reliability requirements. These requirements add extra complexity to the software design and test. For example, for hard real-time systems, special test and analysis techniques have to be developed to test the correctness of the control model and to guarantee the system can meet deadlines under all conditions. Secondly, real-time systems usually operate in a real environment, which may be unknown during the design time or continuously evolve itself as time proceeds. Thus the software that controls these real-time systems should be able to deal with uncertainties, such as to dynamically configure itself to adapt to a changing environment. This also poses great challenges to effectively and thoroughly test the software under development. Besides these unique challenges, the rapid growth of real-time embedded systems brings two other factors into real-time software's complexity. First, real-time systems are more and more networked together. It will not be unusual for hundreds of embedded

controllers, smart sensor and actuators work together to finish a common task. Thus, scalability is becoming a more and more important issue to deal with. Second, with their rapid growth, real-time systems are expected to carry out more and more complex functionalities. It has been foreseen that the new breed of real-time embedded systems, which have enough computational power and memory to carry out complex functionalities, will become dominant [Rob00]. In order to handle the complexity of these systems, much effort has to be put on system modeling, design, analysis, and verification techniques. These high level techniques are becoming essential, as distributed real-time systems become more and more pervasive and complex.

To address the importance and complexity of real-time software development, various models and development methods have been proposed. However, so far none of them fits very well in supporting the design, test, and execution of real-time software from a systematic way. In studying the literature of current real-time software development methods, we notice the following common deficiencies:

- In the software development lifecycle, different stages are disconnected to each other, thus resulting in inherent inconsistency among analysis, design, test, and implementation artifacts. For example, in the analysis stage for large-scale complex systems, mathematical models are usually built to analyze the control algorithms. However, these mathematical models are rarely effectively utilized by the design stage which uses modeling languages such as UML, and the implementation stage which uses programming languages such as C or Java. Because of this kind of model discontinuity, transformation from one model to

another is needed between different stages. This transformation is an error prone process. Furthermore, it makes it very difficult, if not impossible, to maintain a consistent view of the artifacts from different development stages.

- Software test for distributed real-time systems is largely ad hoc and at a low level. Although control algorithms can be developed and tested in the analysis stage, once they are transformed into implementation codes, extensive test is still needed because of the discontinuity problem mentioned above. For this reason, a lot of tests are meaningful only after the actual code is generated, and in some cases, has to be conducted with the real hardware. This kind of low-level activities results in later detection of inconsistency with the system specification.
- Despite there is continuous need for software to dynamically reconfigure itself in order to adapt to new situations or new environments, there is no effective and systematic way to design and analyze these kinds of self-adaptive software [Rob00]. As real-time systems usually operate in dynamic real environments, they tend to exhibit dynamic reconfiguration to change their structures and operation modes according to different situations. Thus it is desirable if a real-time software development method provides a systematic way to model dynamic reconfiguration of systems.
- Scalability becomes a more and more important issue as real-time embedded systems increasingly networked together. To ensure scalability, component based technology [Ras01] and suitable software structures and physical topologies are

needed. Meanwhile, computer-based modeling and simulation (M&S) methodology is required since the scale of systems is well beyond what analytical tools alone can handle and there is limited ability to do controlled experiments.

1.7 Summary of Contributions

Overall, the main contribution of this research is the development of a simulation-based software development methodology [Hu02, Hu03a] to manage the complexity of distributed real-time software. This methodology, based on discrete event system specification (DEVS), overcomes the “incoherence problem” between different design stages by emphasizing “model continuity” through the development process. Specifically, techniques have been developed so that the same control models that are designed can be tested and analyzed by simulation methods and then easily deployed to the distributed target system for execution. To improve the traditional software testing process where real-time embedded software needs to be hooked up with real sensor/actuators and tested in a physical environment, a virtual test environment is developed that allows software to be effectively tested and analyzed in a virtual environment, using virtual sensor/actuators. Within this environment, stepwise simulation-based test methods have been developed so that different aspects, such as logic and temporal behaviors, of a real-time system can be tested and analyzed incrementally.

Based on this methodology, a simulation and testing environment for distributed autonomous robotic systems is developed. This environment applies stepwise simulation-based testing methods to test distributed autonomous robotic systems. In particular, the

work on “robot-in-the-loop” simulation allows real and virtual robots to work together for a meaningful system-wide test. For example, when developing a robotic system that includes hundreds of mini mobile robots, one or several real robots can be tested and experimented with other hundreds of virtual robots that are simulated on computers. With the help of this environment, several distributed robotic systems have been successfully developed and investigated. One of them is a “dynamic team formation” system [Hu03b] in which mobile robots search for each other, and then form a team dynamically through self-organization.

To model systems’ dynamic reconfiguration, a feature exhibited by many real-time systems in order to adapt to the continuous changing environment, the variable structure modeling capability is exploited [Hu03c]. Operation boundaries for structure change operations have been defined so that the hierarchical modular property of models will not be violated during systems’ reconfiguration. This variable structure modeling capability has been implemented in the DEVSJAVA modeling and simulation environment. It is also well demonstrated by the “dynamic team formation” robotic example where robots establish connections dynamically.

While real-time systems become more and more pervasive, the scale of these systems also increase rapidly. Using simulation-based methods to design and study these large-scale systems, simulation performance plays a critical role. To improve simulation speed, a high performance simulation engine is developed for large-scale cellular DEVS models. This simulation engine makes use of the characteristics of large-scale cellular models by employing a special data structure so that the smallest tN and the imminent components

can be found efficiently. The speed up of this new simulation engine as compared to the standard *coordinator* has been demonstrated by several examples.

1.8 Dissertation Organization

The remaining of this dissertation is organized as follows: Chapter 2 discusses DEVS as a simulation-based design framework for real-time systems. Specifically, it introduces the DEVS and RTDEVS formalism and discusses how DEVS can be applied to model a real-time system's structure, behavior and timeliness in a systematical way. Based on Chapter 2, Chapter 3 presents the "model continuity" methodology for distributed real-time software development. It discusses the different stages for developing real-time software and illustrates in detail how stepwise simulation-based test methods can be applied to incrementally test the software under development. Chapter 4 focuses on the variable structure modeling capability. It presents the conceptual development of variable structure modeling in DEVS and describes how it can be implemented in the DEVJSJAVA environment. In Chapter 5 and Chapter 6, two distributed autonomous robotic examples are presented to demonstrate the proposed the methodology. While Chapter 5 presents a system with two robots that establish connections dynamically using the variable structure modeling capability, Chapter 6 shows a scalable system that can essentially include any number of robots. Chapter 6 also illustrates how simulation-based methods can be applied to analyze/evaluate the performance of a system under development. In Chapter 7, the high performance simulation engine is presented. Examples and test data are given to demonstrate the speed up of this new simulation engine as compared to the

standard *coordinator*. Chapter 8 concludes this dissertation research and provides some future research directions.

CHAPTER 2

DEVS FOR REAL-TIME SYSTEM DEVELOPMENT

2.1 DEVS as a Simulation-based Design Framework

A system is a set of related elements considered as a unity. System theory is a body of concepts and methods for the description, analysis and design of complex entities leading to some important generalizations about such entities. System theory has evolved a systematic approach to system design. This is based on first defining the system objectives, proceeding to the generation of a candidate design, analyzing and evaluating the candidate in terms of the objectives and then deciding either to implement the candidate or to return to an earlier stage to generate another candidate [Fin88]. As a system design approach, simulation-based system design employs a *plan-generate-evaluate* process. The *plan* phase organizes all the models of design alternatives within the chosen system boundary and design objectives. The *generate* phase synthesizes a candidate design model intended to meet the set of design objectives. Finally, the *evaluate* phase evaluates behavior and/or performance of the generated model through simulation using an appropriate experimental frame derived from the design objectives. The overall design cycle repeats the generation and evaluation phases until an acceptable design is found [zei00].

The DEVS (Discrete Event System Specification) [zei76], [zei00] formalism is a formal modeling and simulation (M&S) framework based on generic dynamic systems

concepts. DEVS is a mathematical formalism with well-defined concepts of coupling of components, hierarchical, modular model construction, support for discrete event approximation of continuous systems and an object-oriented substrate supporting repository reuse. DEVS has a well-defined concept of system modularity and component coupling to form composite models. It enjoys the property of closure under coupling which justifies treating coupled models as components and enables hierarchical model composition constructs. Based on the classic DEVS formalism, RTDEVS [Hon97] formalism was developed for real-time system specification. The formalisms of DEVS and RT-DEVS are given in the next section.

DEVS is not just a theoretical framework, as it has been operationalized to serve as a practical simulation and execution tool in a variety of implementations. A DEVS model can be simulated by a simulator and then executed in the DEVS runtime environment. Over years, DEVS' simulation infrastructures have been implemented using various programming languages such as DEVSC++ [Zei96], DEVSJAVA [DEVJ], and over various middleware such as DEVS/CORBA [Kim99], and DEVS/HLA [zei99].

Based on the simulation infrastructures and techniques that have been developed, DEVS has the ability to test and evaluate a model's correctness as the model being iteratively refined on different abstraction levels. This allows early discovery of high-level defects as well as a complete validation of the finalized model in detail. In DEVS, simulation-based test and evaluation is conducted within experimental frames. An experimental frame is a specification of the conditions under which the system is observed or experimented with. It typically has three types of components: *generator*,

which generates input segments to the system; *acceptor*, which monitors an experiment to see the desired experimental conditions are met; and *transducer*, which observes and analyzes the system output segments. With experimental frames, not only the correctness of a model can be tested and validated, but also the performance of the model, such as average response time, can also be measured by using a transducer. Moreover, different perspectives of the system can be captured through specialized experimental frames and tested in the various phases of development.

To organize the family of alternative designs from which a candidate design can be selected, generated, and evaluated, the system entity structure (SES) [Roz90] can be used. The system entity structure formalism is a structural knowledge representation scheme that systematically organizes a family of possible structures of a system. Such a family characterizes decomposition, coupling, and taxonomic relationships among entities. SES makes it possible to automatically select the best design alternatives by automatically pruning the SES structure.

The combination of systematical modeling capability, simulation-based design approach, experimental frame, and system entity structure makes DEVS a competitive candidate for developing real-time systems. With DEVS, a real-time system's behavior, structure, and timeliness can be effectively modeled and then tested/evaluated by simulation methods. The behavior aspects of a system can be specified by DEVS Atomic models which have well defined state, state transition functions (external and internal), activity, time advance function, etc. The structure aspects of the system can be specified by DEVS Coupled models which allow hierarchical composition of models by coupling

input ports and output ports between models. With the time advance function, a DEVS model can specify time explicitly. This makes it possible to model and evaluate real time requirements in a systematic way. Furthermore, DEVS also has the capability to model dynamic reconfigurations of a real-time system. This variable structure modeling capability will be discussed further in Chapter 4.

The rest of this chapter describes the basic concepts of DEVS and shows how it can be applied to real-time systems design. Notice that although a brief description of DEVS formalism is presented, readers are recommended to read [Zei76], [Zei00] for more information about the DEVS modeling and simulation framework.

2.2 A Brief Review of DEVS Concepts

Figure. 2.1 depicts the conceptual framework underlying the DEVS formalism [Zei76, Zei00]. The modeling and simulation enterprise concerns four basic objects:

- the *real system*, in existence or proposed, which is regarded as fundamentally a source of data.
- *model*, which is a set of instructions for generating data comparable to that observable in the real system. The *structure* of the model is its set of instructions. The *behavior* of the model is the set of all possible data that can be generated by faithfully executing the model instructions.
- *simulator*, which exercises the model's instructions to actually generate its behavior.
- *experimental frame*, which captures how the modeler's objectives impact on model construction, experimentation and validation. In DEVJAVA, experimental frames

are formulated as model objects in the same manner as the models that are of primary interest. In this way, model/experimental frame pairs form coupled model objects with the same properties as other objects of this kind. This uniform treatment yields immediate benefits in terms of modularity and system entity structure representation.

The basic objects are related by two relations:

- a *modeling relation* linking real system and model, defines how well the model represents the system or entity being modeled. In general terms, a model can be considered valid if the data generated by the model agrees with the data produced by the real system in an experimental frame of interest.
- a *simulation relation*, linking model and simulator, represents how faithfully the simulator is able to carry out the instructions of the model.

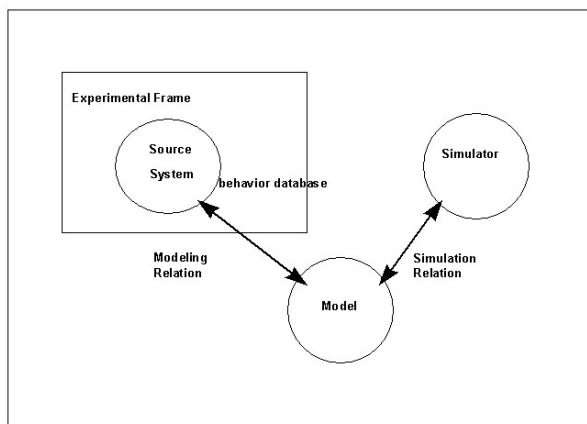


Figure. 2.1. Basic Entities and Relations

The basic data items produced by a system or model are *time segments*. These time segments are mappings from intervals defined over a specified time base to values in the ranges of one or more variables. An example of a data segment is shown in Figure. 2.2.

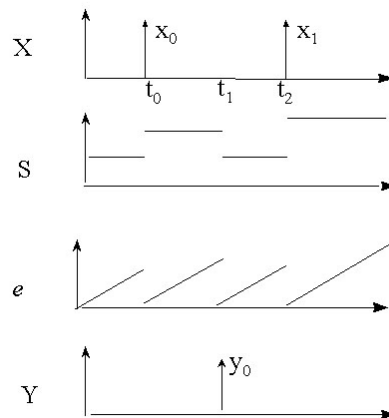


Figure. 2.2. Discrete event time segments.

The structure of a model may be expressed in a mathematical language called a *formalism*. The discrete event formalism focuses on the changes of variable values and generates time segments that are piecewise constant. Thus an event is a change in a variable value that occurs instantaneously.

In essence the formalism defines how to generate new values for variables and the times the new values should take effect. An important aspect of the DEVS formalism is that the time intervals between event occurrences are variable (in contrast to discrete time where the time step is generally a constant number).

In the DEVS formalism, one must specify 1) basic models from which larger ones are built, and 2) how these models are connected together in hierarchical fashion.

To specify modular discrete event models requires that we adopt a different view than that fostered by traditional simulation languages. As with modular specification in general, we must view a model as possessing input and output ports through which all interaction with the environment is mediated. In the discrete event case, events determine

the values appearing on such ports. More specifically, when external events, arising outside the model, are received on its input ports, the model description must determine how it responds to them. Also, internal events arising within the model, change its state, as well as manifesting themselves as events on the output ports, which in turn are to be transmitted to other model components.

A *basic model* contains the following information:

- the set of input ports through which external events are received,
- the set of output ports through which external events are sent,
- the set of state variables and parameters: two state variables are usually present, “phase” and “sigma” (in the absence of external events the system stays in the current “phase” for the time given by “sigma”),
- the time advance function which controls the timing of internal transitions – when the “sigma” state variable is present, this function just returns the value of “sigma”,
- the internal transition function which specifies to which next state the system will transit after the time given by the time advance function has elapsed,
- the external transition function which specifies how the system changes state when an input is received – the effect is to place the system in a new “phase” and “sigma” thus scheduling it for a next internal transition; the next state is computed on the basis of the present state, the input port and value of the external event, and the time that has elapsed in the current state,
- the confluent transition function which is applied when an input is received at the same time that an internal transition is to occur – the default definition simply applies the

internal transition function before applying the external transition function to the resulting state, and

- the output function which generates an external output just before an internal transition takes place.

A Discrete Event System Specification (DEVS) is a structure

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$$

where,

X : set of external input events;

S : set of sequential states;

Y : set of outputs;

$\delta_{int}: S \rightarrow S$: internal transition function

$\delta_{ext}: Q \times X^b \rightarrow S$: external transition function

$\delta_{con}: Q \times X^b \rightarrow S$: confluent transition function

X^b is a set of bags over elements in X ,

$\lambda: S \rightarrow Y^b$: output function generating external events at the output;

$ta: S \rightarrow R_{0,\infty}^+$: time advance function;

$Q = \{ (s,e) \mid s \in S, 0 \leq e \leq ta(s) \}$ is the set of total states where e is the elapsed time since last state transition.

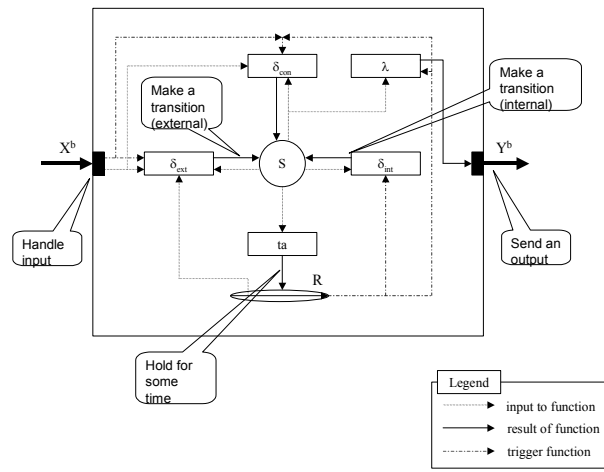


Figure. 2.3. Interpretation of the DEVS structure

The interpretation of these elements is illustrated in Figure. 2.3. At any time the system is in some state, s . If no external event occurs the system will stay in state s for time $ta(s)$. Notice that $ta(s)$ could be a real number and it can also take on the values 0 and ∞ . In the first case, the stay in state s is so short that no external events can intervene – we say that s is a *transitory* state. In the second case, the system will stay in s forever unless an external event interrupts its slumber. We say that s is a *passive* state in this case. When the resting time expires, i.e., when the elapsed time, $e = ta(s)$, the system outputs the value, $\lambda(s)$, and changes to state $\delta_{int}(s)$. Note that output is only possible just before internal transitions.

If an external event $x \in X^b$ occurs before this expiration time, i.e., when the system is in total state (s, e) with $e \leq ta(s)$, the system changes to state $\delta_{ext}(s, e, x)$. Thus the internal transition function dictates the system's new state when no events have occurred since the last transition. While the external transition function dictates the system's new state

when an external event occurs – this state is determined by the input, x , the current state, s , and how long the system has been in this state, e , when the external event occurred. In both cases, the system is then in some new state s' with some new resting time, $ta(s')$ and the same story continues.

Note that an external event $x \in X^b$ is a bag of elements of X . This means that one or more elements can appear on input ports at the same time. This capability is needed since Parallel DEVS allows many components to generate output and send these to input ports all at the same instant of time.

The above explanation of the semantics (or meaning) of a DEVS model suggests, but does not fully describe, the operation of a simulator that would execute such models to generate their behavior. Nevertheless, the behavior of a DEVS is well defined and can be depicted as we mentioned earlier in Figure. 2.2. In that figure, the *input trajectory* is a series of events occurring at times such as t_0 and t_2 . In between such event times may be those, such as t_1 , which are times of internal events. The latter are noticeable on the *state trajectory*, which is a step-like series of states, which change at external and internal events (second from top). The *elapsed time trajectory* is a saw-tooth pattern depicting the flow of time in an elapsed time clock that gets reset to 0 at every event. Finally, at the bottom, the *output trajectory* depicts the output events that are produced by the output function just before applying the internal transition function at internal events.

Basic models may be coupled in the DEVS formalism to form a *coupled model*. A coupled model tells how to couple (connect) several component models together to form a new model. This latter model can itself be employed as a component in a larger coupled

model, thus giving rise to hierarchical construction.

A *coupled* model is defined as follows:

$$DN = \langle X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \rangle$$

where,

X : set of external input events;

Y : a set of outputs;

D : a set of components names;

for each i in D ,

M_i is a component model

I_i is the set of influencees for i

for each j in I_i ,

$Z_{i,j}$ is the i -to- j output translation function

A *coupled model* template captures the following information:

- the set of components
- for each component, its influencees
- the set of input ports through which external events are received
- the set of output ports through which external events are sent
- the coupling specification consisting of:
 - the external input coupling (EIC) connects the input ports of the coupled to one or more of the input ports of the components
 - the external output coupling (EOC) connects the output ports of the components to one or more of the output ports of the *coupled* model

- internal coupling (IC) connects output ports of components to input ports of other components

Real-Time DEVS Formalism

Real-Time DEVS (RTDEVS) formalism extends the classic DEVS formalism in atomic DEVS models. The RTDEVS formalism for coupled models remains the same as the original. An atomic RTDEVS model, RTAM, is defined as follows:

$$\text{RTAM} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta, A, \psi \rangle$$

where,

$X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda$: remains the same as conventional DEVS;

$ta : S \rightarrow I_{0,\infty}^+$: time advance function,

where $I_{0,\infty}^+$ is the non-negative integers with ∞ adjoined;

A : a set of activities with the constraints

$\psi : S \rightarrow A$: an activity mapping function

In the classic DEVS formalism, simulation time advances only when a simulator calls the time advance function ta of the associated model. The time advance function ta in the RTDEVS formalism behaves the same as that in the classic DEVS formalism except that here the time is an integer, while in classic DEVS time is a real number. The time calculated by the time advance function also synchronized with the wall clock time. This is because a simulation clock in RTDEVS is no longer a virtual clock but a real-time

clock. An activity is an operation that takes a certain amount of time to complete the assigned task [Hon97]. This was adopted by [Hon97] to represent some time-consuming operations such as waiting for a message, processing a job, and so forth.

2.3 Modeling a System's Structure, Behavior, and Timeliness Using DEVS

This section gives an informal introduction of using DEVS to model a real-time system's structure, behavior, and timeliness. For each of them, a simple example with the corresponding DEVSJAVA code is given.

The structure of a system identifies the entities that are to be modeled and the relationships between them (e.g., communication relationships, containment relationships). DEVS coupled models are used to model a system's structure. Corresponding to a system with multiple subsystems, a DEVS coupled model contains several component models (DEVS atomic model or coupled model). Each model has its own input and output ports. DEVS couplings can be established between the output/input ports to enable inter-communications between models. Within this framework, a system that exhibits inter-communication relationship as well as hierarchical containment relationship between its entities can be naturally modeled using DEVS coupled model.

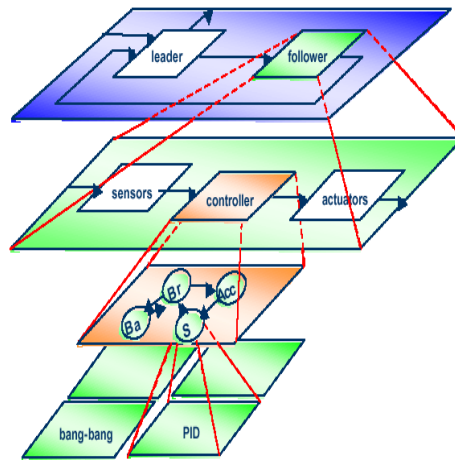


Figure. 2.4: A “leader-follower” system modeled by a DEVS coupled model

Figure. 2.4 shows a “leader-follower” multi-agent system that is modeled using a DEVS coupled model. As can be seen, this hierarchical coupled model clearly models the hierarchical relationship between different components of the system. From the figure we can see that this system has two agents: a *leader* and a *follower*. They communicate directly with each other. The *follower* agent has *sensors*, *controller*, and *actuators*, among which the *controller* is further decomposed to several sub-components such as *PID*, *bang-bang*, etc. The DEVSJAVA code that describes the *leader_follower* coupled model is shown below. Note that in this code, the *Follower* itself is a coupled model, whose subcomponents are not shown in this code.

```

public class leader_follower extends digraph{
    public leader_follower (){
        Leader leader = new Leader();
        Follower follower = new Follower();
        add(leader);
        add(follower);
        addCoupling(leader, "outputPort", follower, "inputPort");
        addCoupling(follower, "outputPort", leader, "inputPort");
    }
}

```

One of the features exhibited by some real-time systems is that a system may dynamically reconfigure itself in order to adapt to the continuous changing environment. To model dynamic reconfiguration of a system, we developed the variable structure modeling capability. Specifically, four structure changing operations: *addModel()/removeModel()*, *addCoupling()/removeCoupling()* have been developed so that models and their couplings can be added/removed dynamically. More information for this variable structure modeling capability can be found in [Hu03c].

While the structure of a system is modeled by DEVS coupled models, the behavior of a system can be modeled by DEVS atomic models. A DEVS atomic model has well-defined state, state transition functions (triggered by external or internal events), time advance function, output function, etc. From the design point of view, an atomic model can be viewed as a timed state machine. The transition from one state to another is triggered by external or internal events. The external event is an external message received from the model's input ports; the internal event is a time out event generated internally. The model can generate output and send it out through its output ports.

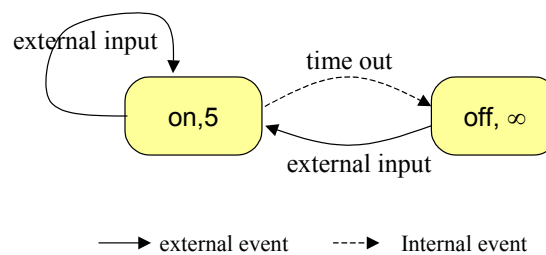


Figure. 2.5: Timed state diagram of a screen saver program

Figure. 2.5 shows an example that exhibits the dynamic behavior of a simple screen saver program. This program has 5 seconds watching time, meaning the screen will be

turned off if there is no input from mouse or keyboard in 5 seconds. When the screen is off, any input from mouse or keyboard will turn it on; when the screen is on, any input will keep it on, while resetting the watching time starting from zero.

The above behavior can be modeled by a DEVS atomic model with the following DEVSJAVA code.

```

public class screen_saver extends atomic{
    .....
    public void initialize() {
        holdIn("on", 5); // 5 seconds --- 5 minutes
    }
    public void deltext(double e, message x) {
        Continue(e);
        for (int i = 0; i < x.getLength(); i++) {
            if (messageOnPort(x, "keyboard", i)) holdIn("on", 5);
            if (messageOnPort(x, "mouse", i)) holdIn("on", 5);
        }
    }
    public void deltint() {
        holdIn("off", INFINITE);
    }
    public message out() {
        message m = new message();
        return m;
    }
}

```

Timeliness is an essential property of any real-time system. DEVS handles time explicitly by defining the time advance function in atomic models. Whenever an atomic model transits to a new state, its time advance function specifies how long the model will stay at that state. During this period of time, if there are external events, a model responds in its external transition function *deltext()*, which may change the model to a new state with a new time period; otherwise an “internal” time out event will be generated and the model responds it in its internal transition function *deltint()*. Figure. 2.6 shows some time patterns that can be easily models in DEVS.

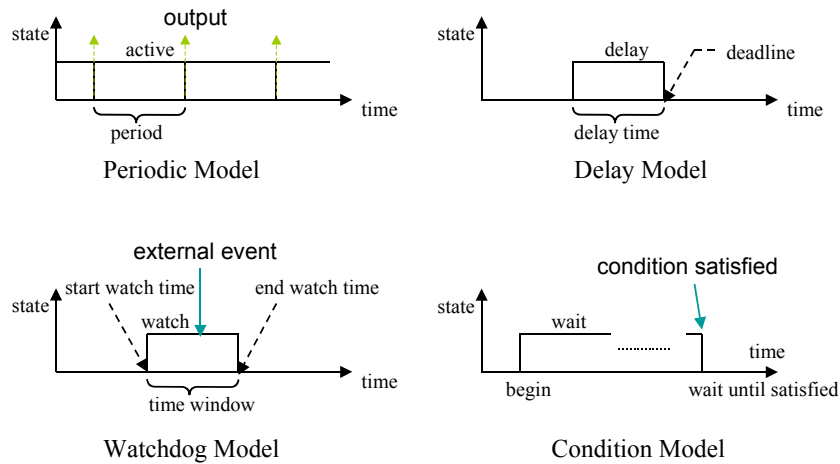


Figure. 2.6: Time patterns modeled in DEVS

To give an example, the corresponding DEVSJAVA code for a *Periodic* model (with period equals to 10) is given below:

```

public class periodic_model extends atomic{
    double period = 10;
    .....
    public void initialize() {
        holdIn("active", period);
    }
    public void deltext(double e, message x) {
        Continue(e);
    }
    public void deltint() {
        holdIn("active", period);
    }
    public message out() {
        message m = new message();
        return m;
    }
}

```

Based on these patterns, models with more advanced behaviors can be built. For example, by adding output message in the *out()*, the *Periodic* model is extended to a *generator* model which generates output periodically. It can be further extended to a more advanced *generator* with variable periods during different time segments.

2.4 Connect to the External Environment through DEVS *activity*

A real-time system continuously interacts with an external environment through sensors, actuators, or other hardware interfaces. Sometimes, it also uses software packages from third-party vendors for special computation purpose. For example, a real-time system may use an image-processing package to process images. To model these hardware or software interfaces in DEVS, DEVS *activity* has been developed through which a model can interact with its external environment.

A DEVS *activity* is a thread that can essentially be any kinds of computation tasks. For example, in the context of real time systems, an *activity* could be hardware (sensor/actuators) interfaces, network proxies, special software computation packages, and so on. Each *activity* belongs to an atomic model, which decides when to start or kill the activity. An *activity* may or may not return result to the atomic model. If an *activity* returns result to the atomic model, the result will be put on a reserved input port (the "*outputFromActivity*" port) as an external event and then processed by the model's external transition function.

Figure 2.7 shows the relationship between a DEVS model, *activity*, and the external environment. Note that dependent on the context of the system, this external environment could be a real physical environment, a third-party software component, or any other entities that are outside the boundary of the DEVS model.

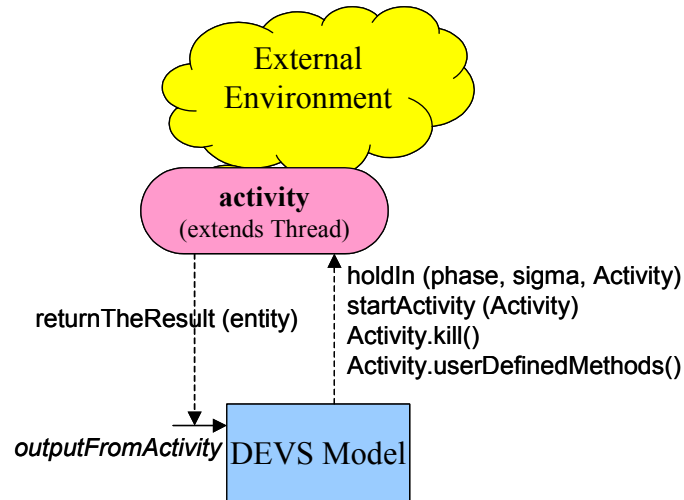


Figure 2.7: DEVS model, activity, and the external environment

As shown by Figure 2.7, the DEVS activity acts as a bridge between DEVS models and the external environment. Specifically, a DEVS model can start an *activity* by calling *holdIn()* or *startActivity()* methods. The difference between them is that the second method only starts an activity, while the first one also changes the state and *sigma* of the model. For example, with the *holdIn()* method, a model can start an activity and in the meantime specify a time window to watch if a desired result returns. A model can stop an *activity* by calling *activity*'s *kill()* method. An *activity* returns computation result to the DEVS model by method *returnTheResult()*, which sends the result as a message to the DEVS model's reserved input port *outputFromActivity*. This message, as an external event, triggers the model's external transition function *deltext()*, which processes the message and gets the result from *activity*.

2.5 Simulation and Execution of DEVS Models

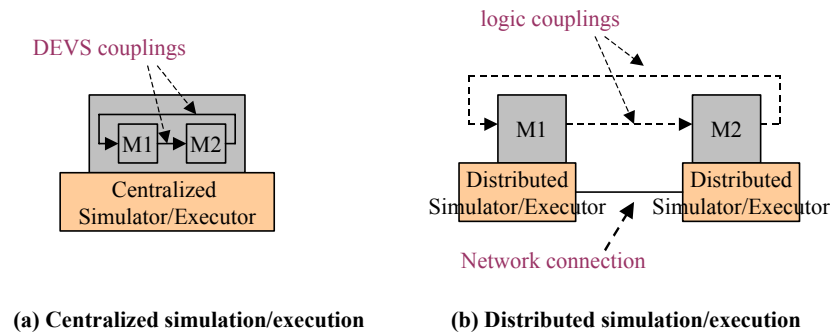


Figure 2.8: Simulate/Execute DEVS model in centralized and distributed environment

To simulate DEVS models, DEVS simulators are developed. While DEVS models model the structure, behavior, and timeliness of a system, DEVS simulators drive the execution of these models. The clear separation between DEVS models and simulators makes it possible for the same model to be simulated by different simulators appropriate to different design stages or different simulation environments. Figure. 2.8 shows the simulation of the same DEVS model (which has subcomponents $M1$, and $M2$) in a centralized and distributed environment. In centralized simulation, the model is simulated by a centralized simulator on a computer. In distributed simulation, subcomponents ($M1$ and $M2$) of the model are deployed to different computers, although the couplings among them are still kept the same as those in centralized simulation. Each subcomponent is simulated by a distributed simulator. If a model sends a message to another model, saying $M1$ sends a message to $M2$, the message is actually passed across the network. Thus one of the important roles of a distributed simulator is to establish network connection and to enable distributed message passing between models³. Note that in distributed simulation

³ In the implementation of DEVSJAVA, a *coordServer* is used to establish communication between distributed executors.

or execution, the time for message passing between distributed models is significantly longer than that in centralized simulation or execution. This is because there exists network latency between two computers.

Two approaches are available to apply a DEVS model to real execution. First, a model can be transformed into executable code based on the target executing platform. This is an approach adopted by most software development methods. It usually results in fast execution of the code. However, the compiler, which transforms the model into executable code and optimizes the code, is typically expensive to develop. On the other hand, a DEVS model itself can be viewed as an implementation and executed by a real-time execution engine⁴. In this case, transformation is not needed and the model remains unchanged from the design stage to implementation stage. Although this approach may sacrifice execution speed dependent on the efficiency of the underline real-time execution engine, it brings several advantages. First, it eases the transfer of model between different execution platforms, between centralized execution and distributed execution. In fact, the model is kept unchanged and different execution engines are chosen for different execution environments. For example, in Figure 2.8, the same model can be executed in both centralized and distributed environments, by centralized and distributed executors respectively. Secondly, from the design point of view, as the model is kept unchanged, the designer works on the same set of models from design, to simulation-based test and finally to the execution. Different simulators and real-time executors can be chosen for the same model based on different stages of a development process. In the dissertation,

⁴ A real-time execution engine, also called executor, is a stripped-down version of a real-time simulator.

we call this second approach the “model execution” approach. This is the approach that we employed from simulation-based design to real execution of a model.

Hierarchical Distributed Topology

While a coupled model can be distributed to multiple computers for simulation or execution, its subcomponents (assuming the subcomponent is a coupled model too) can also be distributed on multiple computers. This results in a hierarchical distributed topology as shown in Figure 2.9.

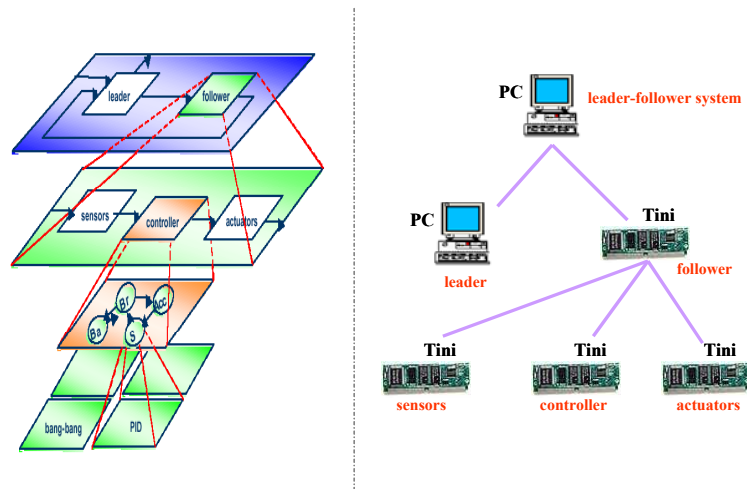


Figure 2.9: Hierarchical distributed simulation or execution topology

In this example, the “leader-follower” system described in Figure 2.4 is simulated or executed in an environment with hierarchical distributed topology. Specifically, the two models *leader* and *follower* are distributed on two different computers. Furthermore, the three components of the *follower* model: *sensors*, *controller*, and *actuators* are distributed on different computers too. Note that the hierarchical structure of computers as shown in Figure 2.9 only reflects the logic topology between computers. It doesn’t mean the

actually physical topology (such as a bus topology) of computers. As this hierarchical distributed topology keeps the same structure as that of the model, it shares the same hierarchical modular properties possessed by the model. For example, the *leader* model only sees its peer model *follower*. It doesn't know, and doesn't care either, that the components of *follower* are actually distributed on three different computers.

Model Mapping

To help to deploy models to a distributed environment, a mapping process is needed. As shown in Figure 2.10, during the mapping process, the designer assigns models to computers for simulation or execution. The decision of choosing which computer for a specific model is dependent on the system requirements and the design considerations. For example, if a model needs to interact with a special hardware, it can only be assigned to the computers that are equipped with the required hardware.

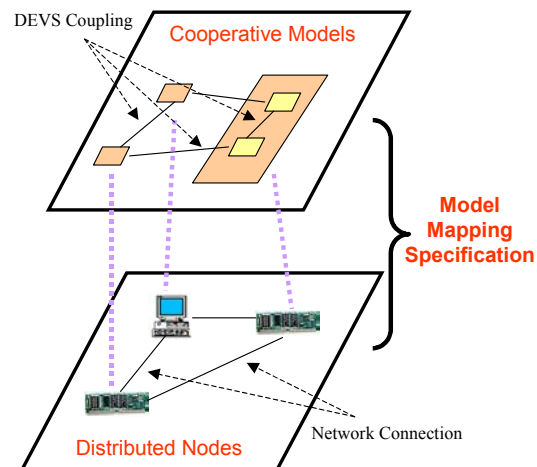


Figure 2.10: Model mapping

To facilitate the mapping process, a prototype model mapping specification is

proposed and interpreted as below:

$\langle M, (S, St), (As, Ps), (Am, Pm) \rangle$

- *M: $M \in DEVS$ is a DEVS Coupled Model or Atomic Model*
- *S: $S \in Simulator$ is a DEVS Simulator to simulate M*
- *St : $St \in \{R^+, NULL\}$ is the time scale of the simulator. St is NULL if S is a fast mode simulator*
- *As is Server's network address. As is NULL if M is the root Model*
- *Ps is Server's socket port. Ps is NULL if As is NULL.*
- *Am is M's network address. Am is NULL if M is a leaf Model*
- *Pm is M's socket port. Pm is NULL if Am is NULL.*

CHAPTER 3

A MODEL CONTINUITY SOFTWARE DEVELOPMENT METHODOLOGY

3.1 Model Continuity in Software Development

The wide acceptance of software engineering has made software development a process including analysis, design, implementation, test, and maintenance stages. At each stage, numerous methods and tools have been developed. Breaking a software development process into multiple stages improves the manageability and productivity of software development in general, as these stages break down the problem into smaller manageable pieces. On the other hand, although these stages each focuses on its own problems, they belong to the same unified process and are not separated from each other. For example, it's very common for a project to go back and forth among different stages to change or to refine the design. Thus one of the most challenging tasks for software development is to maintain coherence, or model continuity, among different development stages. This is becoming more important as today's software systems become more and more complex.

Model continuity refers to the ability to transition as much as possible of a model specification through the stages of a development process. It manages the complexity of software development by emphasizing "coherence" between different design stages, thus resulting in a more "fluent flow" along the development process. Unfortunately, existing frameworks tend to support only one phase of the development process. They do not

work together coherently, i.e., allowing the output of a framework used on one phase to be consumed by a different framework used in the next phase [Ger02],[Sch00],[Ran02]. This discontinuity between different development stages results in inherent inconsistency among design, test, and implementation artifacts. In reality, for example, most executing code is not derived by any formal means from the specification or design models. As a consequence, these design models very often become outdated and in most cases lose their value. The lack of model continuity also tends to lead to errors that are not caught until well into the implementation phase. Since the cost of redesign increases as the design moves through development stages, redesign is the most expensive when performed in implementation phase, thus making the incoherent methodology costly [Pre97]. The feature of model continuity becomes ever more important as software systems become more and more complex. For example, a distributed real-time system might include hundreds of computing nodes, smart sensors and actuators, and needs to fulfill very complex tasks in an uncertain or even hostile environment. Without the feature of model continuity, it's very hard to design and test the software that controls these large-scale complex systems.

This chapter presents a software development methodology that supports model continuity for distributed real-time software development. This methodology is based on the DEVS modeling and simulation framework [Zei76],[Zei00]. Corresponding to the general “Design—Test—Execute” development procedure, this approach provides a “Modeling—Simulation—Execution” process which includes several stages to develop real-time software. During these stages, model's continuity is maintained because the

same control models that are designed will be tested by simulation methods and then deployed to the target system for execution. This approach increase system engineers' confidence that the final system in operation is the system they wanted to design and will carry out the functions as tested by simulation methods.

Ensuring consistency among different development stages has been a research issue in various areas. In software engineering, traceability, in the form of requirements traceability[Ram01] or design-code traceability [Ant00], has been advocated to ensure consistency among software artifacts of subsequent phases of the development cycle. Boyd [Boy93] shows how traceability can be achieved when designing reactive systems. In hardware/software codesign, Janka et al. [Jan02] described a methodology that allows the specification stage and design stage to work together coherently when designing embedded real-time signal processing systems. While the preceding approaches use different artifacts in different stages, the approach presented in this chapter allows the same simulation models to be used in the design and implementation stages (the same simulation models become the software to control the system in real execution). The following research efforts are more closely related to this proposed approach by applying simulation-based design. Bagrodia and Shen [Bag91] describe an approach called MIDAS that supports the design of distributed systems via iterative refinement of a partially implemented design where some components exist as simulation models and others as operational subsystems. Gonzalez and Davis [Gon02] present a simulation and control tool that provides the capability to model as well as to control real-world systems. The work presented in this chapter extends the applicability of simulation-based design in

new significant directions: it is based on a formal modeling and simulation framework that supports variable structure modeling. Furthermore, it adopts stepwise simulation methods to allow the control model of a real-time system to be tested and analyzed incrementally.

3.2 Modeling, Simulation, Execution, and Model Continuity

In general, model continuity refers to the ability to transition as much as possible of a model specification through the stages of a development process. Specifically in the context of this dissertation, it means the control models of a distributed real-time system can be designed, analyzed, and tested in DEVS-based modeling and simulation frameworks, and then migrated with minimal additional effort to be executed in a distributed environment [Cho01], [Hu02], [Cho03]. Below we elaborate how this is achieved for non-distributed and distributed real-time systems respectively.

3.2.1 Model Continuity for Non-Distributed Real-time Systems

Real-time Systems are computer systems that monitor, respond to, or control, an external environment. This environment is connected to the computer system through sensors, actuators, and other input-output interfaces [Sha01]. A real-time system from this point of view consists of sensors, actuators and the real-time control and information-processing unit. For simplicity, we call this last one the control unit. The sensors get inputs from the environment and feed them to the control unit. The actuators get commands from the control unit and perform corresponding actions to affect the

environment. The control unit processes the input from sensors and makes decisions based on its control logic. Depending on the complexity of the system, the control unit could have one component or it could have multiple subcomponents, which in turn may have their own sub-control components in a hierarchical way.

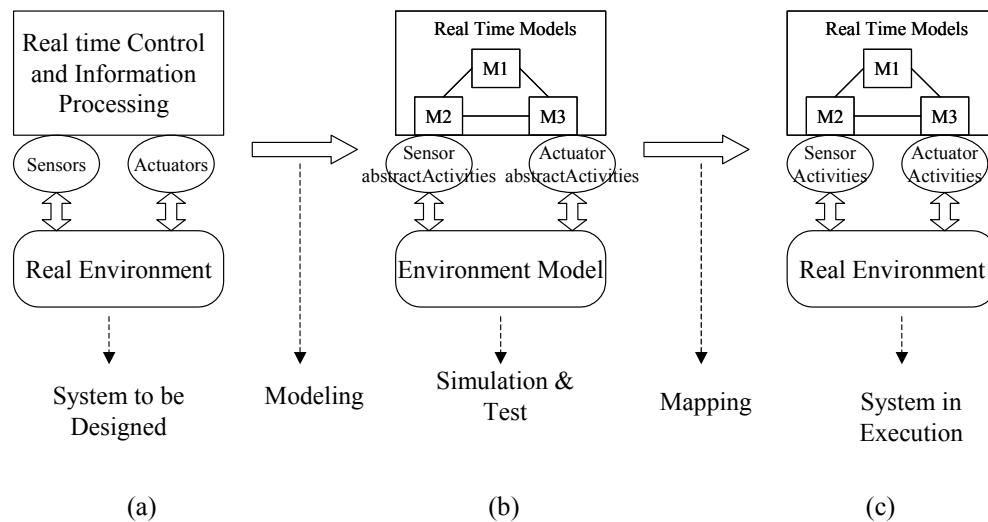


Figure 3.1: Modeling, Simulation and Execution of Non-distributed Real-time System

Once we establish this view of a real-time system as shown in Figure 3.1(a), we can model it easily. In our approach, sensors and actuators are modeled as DEVS *activities*, which is a concept introduced by RT-DEVS for real-time system specification [Hon97]. A DEVS *activity* can be any kind of computer task. However, in this dissertation we only consider the sensor/actuator *activities*. The control unit is modeled as a control model that might have a set of subcomponents. These subcomponents are coupled together so they can communicate and cooperate. With this approach, the control model acts as the brain to process data and make decisions. It could be a simple atomic model or a complex hierarchical coupled model. Sensor/actuator *activities* act as hardware interfaces

providing a set of APIs for the control model to use. To give an example, let us consider the design of a mobile robot. A motor *activity* that drives the robot's motors may be developed. Some typical functions for this motor *activity* could be *move()*, *stop()*, *turn()*, *etc.* How to define an *activity* and its APIs is dependent on how the designer delineates the “control model—*activity*” boundary. For example, we can model a sensor module that may have its own control logic as a sensor *activity*. Or we can include that part of logic into our control model and only model the sensor hardware as an *activity*. The clear separation between control model and *activity*'s functions makes it possible for the designer to focus on his design interest. In the context of real-time systems, the control logic is typically very complex, as the system usually operates in a dynamic, uncertain or even hostile environment. As such, the control model is the main interest of design and test. In our approach, simulation methods are applied to test the correctness and evaluate the performance of this model. The “continuity” of this model is emphasized during the whole process of the methodology, thus model continuity actually means this control model's continuity.

To test and analyze the control model using simulation methods, a virtual testing environment is developed. To build this virtual testing environment, we model the real physical environment as an environment model, which is a reflection of how the real environment affects or is affected by the system under design. Meanwhile, a “simulated” sensor/actuator hardware interface is also provided for the control model to interact with the environment model. This “simulated” sensor/actuator interface is implemented by the *abstractActivity* concept. In contrast to an *activity*, which drives real hardware and is

running in real execution, an *abstractActivity* imitates an *activity*'s interface/behavior and is only used during simulation. A sensor *abstractActivity* gets input from the environment model just as a sensor *activity* gets input from the real environment. An actuator *abstractActivity* does similar things as an actuator *activity* does too. Note that it is important for an *activity* and its *abstractActivity* to have the same interface functions, which are used by the control model in both simulation and real execution. By imposing this restriction, the control model can be kept unchanged in the transition from simulation to execution (it interacts with the environment model and real environment using the same interface functions). By this way, model continuity is supported.

With all the models being developed as shown in Figure 3.1(b), different simulation strategies can be employed to test the control model. Meanwhile, different design alternatives and system configurations can be applied to experiment and exercise the system under design. In our approach, step-wise simulation methods have been developed so that a model can be simulated and tested incrementally before its real execution. During the simulation stage, if we find the simulated result is not correct, the model can be revised and then re-simulated. This “modeling-simulation-revising” cycle repeats until we are satisfied with the result or nothing more can be learned in the simulation stage. A more detailed description of how to use these simulation methods is given in section 3.3.

After the model being tested through simulations, it is mapped (deployed) to the real hardware for execution as shown in Figure 3.1(c). For a non-distributed application, the mapping mainly is the “*activity* mapping” to associate the sensor/actuator *activities* to the

corresponding sensor/actuators hardware. For a distributed application, an extra “*model mapping*” is needed to map a set of cooperative models to a set of networked nodes. By associating the models and *activities* to their corresponding hardware, the system can be executed in a real environment. In execution, the control logic is governed by the control model, which has been tested in step-wise simulations. If the real environment has been modeled in adequate detail by the environment model, this control model will carry out the control logic during execution just the same as it did when simulated. In practice, one may not be able to capture every aspect of the real environment in the environment model in adequate detail, and there will be potential for design problems to surface in real execution. When this happens, re-iteration through the stages can be more easily achieved with the model continuity approach.

3.2.2 Model Continuity for Distributed Real-time Systems

A distributed real-time system consists of a set of subsystems. Each subsystem, like a stand-alone system, has its own control and information processing unit and it interacts with the real environment through sensor/actuators. However, these subsystems are not “along”. They are physically connected by network, and they logically communicate to each other and cooperate to finish system wide tasks. Figure 3.2(a) shows a distributed real-time system example with three computing nodes (subsystems). Distributed real-time systems are much harder to be designed and tested because one subsystem’s behaviors may affect one or all of other subsystems. These subsystems influence each other not only by explicit communications, but also by implicit environment change as they all

share the same environment. For example, in Figure 3.2(a), if *Node1* changes the environment through its actuators, this change will be seen by the sensors of *Node2*, thus affects *Node2*'s decision making. With this kind of influence property, it's not practical to design and test each subsystem separately and then integrate them together. Instead, the system as a whole needs to be designed and tested together from the very beginning of the development.

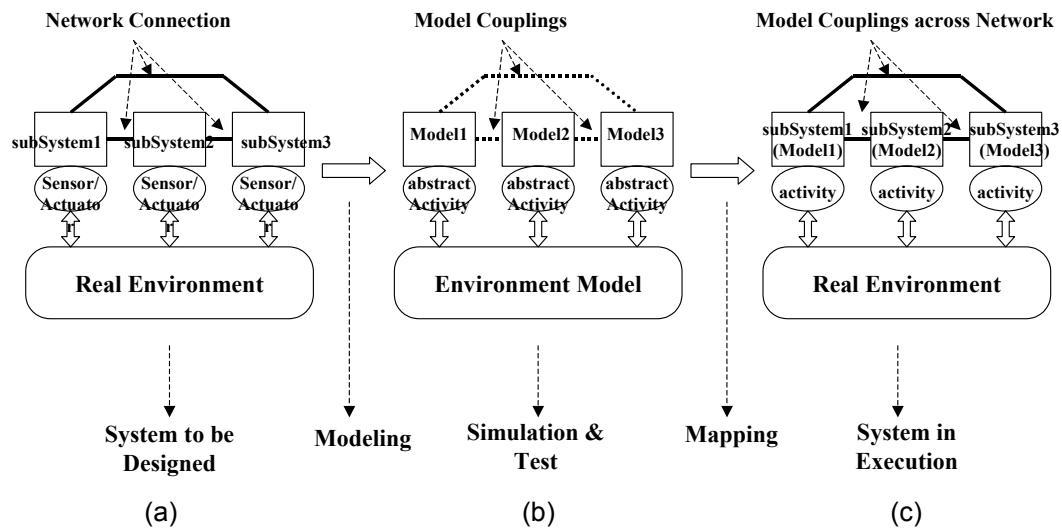


Figure 3.2: Modeling, Simulation and Execution of Distributed Real-time System

In our approach, a distributed real-time system is modeled as a coupled model that consists of several subcomponents. Each subcomponent is corresponding to a subsystem of the distributed real-time system. As described in section 3.2.1, these subsystems are also modeled as DEVS models, which consist of control models and sensor/actuator *activities*. The control model of each subsystem interacts with the real world through sensor/actuator *activities*. These subsystem models are coupled together (by connect one model's output port to another model's input port) so they can communicate. The

couplings among the models are corresponding to the communication connections among the subsystems in the real world.

As shown in Figure 3.2(b), to test the models of distributed real-time systems, environment model and sensor/actuator *abstractActivities* are developed to provide a virtual testing environment. Again, *abstractActivities* should have the same interface functions as its corresponding *activities* so the model using them can be kept unchanged from simulation to execution. Different simulation methods can be employed to simulate and test the models incrementally. A more detailed description of simulation-based test and analysis will be given in section 3.3. Note that each subcomponent can also be tested/simulated independently because DEVS has a well-defined concept of system modularity.

After the models are tested by simulation methods, they are mapped to the real hardware for execution. Similar to a stand-alone system, each subsystem needs to conduct an “*activity mapping*” to associate the sensor/actuator *activities* to the corresponding sensor/actuator hardware. In addition, as the models are actually executed on different computers, the “*model mapping*” is needed to map the models to their corresponding host computers. As shown in Figure 3.2(c), these computers are physically connected by the network and they execute the models that are logically coupled together by DEVS couplings. To govern this mapping, a prototype *Model Mapping Specification* as described in Chapter 2 have been developed, which will facilitate the mapping of models to their network nodes, while maintaining the couplings among them. As such, model continuity for distributed real-time systems means not only the control model of

each subsystem remains unchanged but also the couplings among the component models are maintained from the simulation to distributed execution.

In real execution, the control model of each subsystem makes decisions based on its control logic. It interacts with the real environment through sensor/actuator *activities*. If a model sends out a message, based on the coupling, this message will be sent across the network and put on another model's input port.

3.3 Simulation-based Test for Real-time Systems

3.3.1 A Virtual Test Environment

Testing real-time software is a very challenging task. This is because real-time software interacts with a real environment through sensor/actuator hardware. Traditionally, the software has to be hooked up with the sensor/actuators and placed in the physical field for a meaningful test. This results in a very costly, time consuming, and inefficient process. To improve this process, we developed a virtual testing environment to allow real-time software to be tested in a virtual environment, using virtual sensor/actuators. Within this virtual testing environment, step-wise simulation methods have been developed to test and evaluate the software under development incrementally.

This virtual testing environment consists of the environment model, *abstractActivities*, and the network delay model. The environment model imitates the execution environment of the system and *abstractActivities* act as abstract sensors or actuators. To simulate the network latency for distributed real-time systems, network delay models are developed so that a distributed real-time system can be tested by central simulation in a

more realistic way. Note that all these models can be modeled at different abstraction levels dependent on the test or analysis goals. To maintain model continuity, special implementation techniques, such as the same interface functions between an *abstractActivity* and its *activity*, are developed so that the control model can be easily migrated from simulation to execution.

The core of this virtual testing environment are the stepwise simulation methods that have been developed to allow different aspects, such as the logic and temporal properties, of a system to be tested incrementally. Simulation technology is being increasingly recognized in industry as a useful means to assess the quality of design choices [Son01], [Sch02], [Wei01]. In our work, we view simulation in the following, three-fold perspective: *a)* as a means of verifying the functionality of the proposed solutions by executing the model's dynamics, *b)* as a way of assessing how well performance requirements are met by the proposed design solution, and *c)* as an means of experimenting and exercising the system under design to have a better understanding of the system and reach a better solution for the problem.

3.3.2 Incremental Simulation and Test for Non-Distributed Real-time System

For a non-distributed real-time system, four different simulation steps can be applied to test the model under design. They are fast-mode simulation, real-time simulation, hardware-in-the-loop simulation, and real system test. As shown in Figure 3.3, these simulation methods apply different simulation configurations to test different aspects of the model under test.

In fast-mode simulation, the control model is configured to interact with the environment model through sensor/actuator *abstractActivities*. These models stay in one computer and a DEVS fast-mode simulator is chosen to simulate them. As fast-mode simulation runs in logical time (not connected to a wall-clock), it generates simulation results as fast as it can. Based on these results, the designer can analyze the data to see if the system under test fulfills the logical behavior as desired.

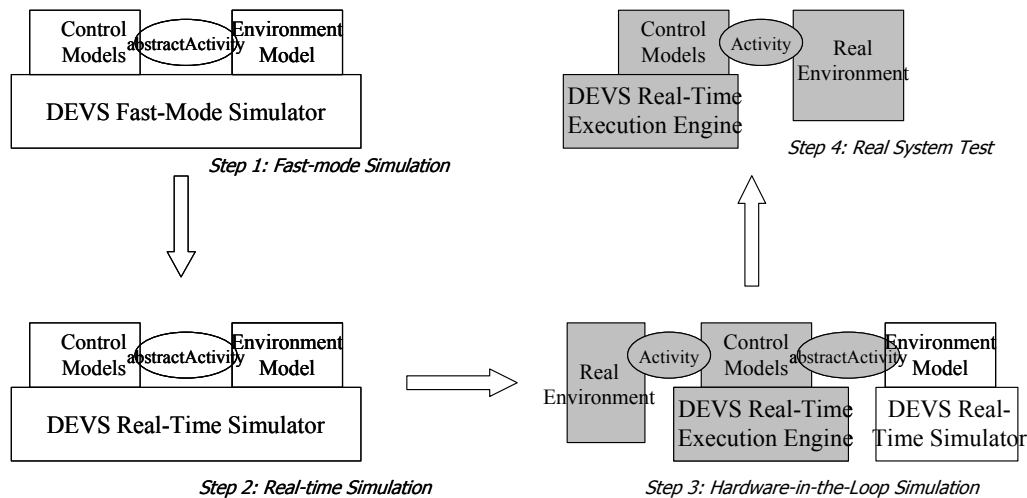


Figure 3.3: Step-wise Simulations of Non-distributed Real-time System

After fast-mode simulation, a real-time simulator is employed to run simulation in a “timely” fashion. This real-time simulator can take a *timeScale* factor, which determines how “fast” a real-time simulation will run as opposed to the wall clock time. For example, *timeScale* being 1 means the simulation will run at the same speed as wall clock; *timeScale* being 0.5 means the simulation will run twice as fast as the wall clock, and so on. By employing the real-time simulator with different *timeScale* factors, the speed of simulation can be controlled. Real-time simulation provides the flexibility to test and analyze a model “online”. For example, with real-time simulation, a GUI interface

can be developed to real-time display the system's states and state transitions. This is usually infeasible for fast-mode simulation, unless all the history data are saved and then played back after the simulation.

In fast-mode and real-time simulations, the model under test and the simulators reside in one computer. This computer is not the same computer as the one in which the model will actually be executed. It is known that for real-time embedded systems, the executing hardware can have significant impact on how well a model's functions can be carried out. For example, processor speed and memory capacity are two typical factors that can affect the performance of an execution. Thus, to make sure that the control model, having been tested in fast-mode and in real-time simulation, can also be executed correctly and efficiently in the real hardware, hardware-in-the-loop (HIL) simulation [Gom01], [Son01], [Wel01] is adopted. As shown in step 3 of Figure 3.3, in HIL simulation, the environment model is simulated by a DEVS real-time simulator on one computer. The control model under test is executed by a DEVS real-time execution engine on the real hardware. This DEVS real-time execution engine is a stripped-down version of DEVS real-time simulator. It provides a compact and high-performance runtime environment to execute DEVS models [Hu01]. In HIL simulation, the model under test interacts with the environment model through *abstractActivities*. These *abstractActivities* act as abstract sensors or actuators. Real sensors or actuators can also be included into HIL simulation by using sensor/actuator *activities*. The decision of which sensors/actuators will be real and which sensors/actuators will be abstract is dependent on the test engineer's testing objectives. With different testing objectives, different combinations of real

sensors/actuators and abstract sensors/actuators can be chosen to conduct an exhaustive test of the control model. Notice that in HIL simulation, as the control model and environment model reside on different computers, a bi-directional connection must be established between the two computers. To serve this purpose, the LAN connection based on TCP/IP protocol is used because it is widely used in industry, can sustain high-speed data transfer, and very portable. This connection is taken care of by the DEVS real-time simulator and execution engine so it is transparent to the model.

Once we passed hardware-in-the-loop simulation, we are ready to leave the simulation stages for real system test. As shown in step 4 of Figure 3.3, in real system test, DEVS real-time execution engine executes the control model. There is no environment model because the control model will interact with the real environment through sensor/actuator *activities*. Note this is also the same setup as that in final execution where the control model interacts with the real environment through sensor/actuator *activities*.

One of the basic rules to conduct these stepwise simulation-based test methods is to put as much as possible of the test in the early steps. This is because the latter the step is, the more costly and time consuming it is to set up the test environment. Unfortunately, in reality most engineers start their test directly from step 4.

3.3.3 Incremental Simulation and Test for Distributed Real-time Systems

Distributed real-time systems are inherently complex because the functions of the systems are carried out by distributed computers over network. Four simulation-based

test steps have been developed to incrementally test these systems. These steps are central simulation, distributed simulation, hardware-in-the-loop simulation, and real system test. To help to understand these steps, an example system with two network computing nodes (two component models) is shown in Figure 3.4.

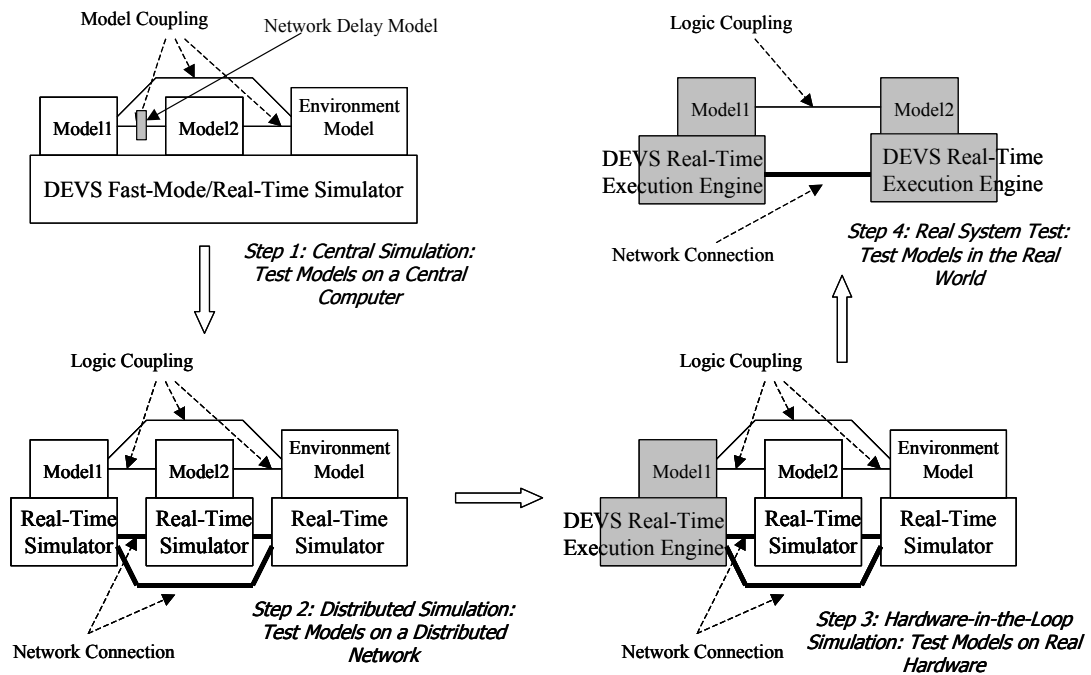


Figure 3.4: Simulation-based test of Distributed Real-time System

The first step in the process is central simulation (step 1 of Figure 3.4). In central simulation, the two models and the environment model are all in one computer. The control models interact with the environment model through sensor/actuator *abstractActivity*. As will be further discussed later, special couplings between *abstractActivity* and the environment model are established to allow them to exchange messages. To model the network latency between the two models that are actually executed on different computers in real execution, network delay models are inserted into

the couplings between models. As the delay model holds on received messages for a period of delay time, messages sent from *Model1* won't reach *Model2* immediately, thus the network latency is simulated. In central simulation, fast-mode simulator and real-time simulator can be chosen to simulate and test the models. As fast-mode simulation runs in logical time (not connected to a wall-clock), it generates simulation results as fast as it can. Based on these results, the designers can analyze the data to see if the system under test fulfills the dynamic behavior as desired. In real-time simulation, the simulation speed is synchronized with the wall-clock time. This provides designers the flexibility to trace the simulation trajectory in real time. For example, a graphic user interface can be developed to display the state changing of each model in real time.

While in central simulation, network delay models are used to model the network latency between different subsystems, in distributed simulation, the control models are tested on the real network. As shown in step 2 of Figure 3.4, in distributed simulation, two models reside on two different computers. The environment model may reside on another computer or on the same computer with one of the models. The couplings between these computers remain the same, but happen across the network. All of these models are simulated by real time distributed simulators. These real time simulators take care of the underlying network synchronization/coordination and make it transparent to the models. The network delay models are no longer needed because the models are tested in a real network. Note that in step 2, distributed simulation has to run in a real-time fashion. This is because part of the real physical world, the real network, is involved in this simulation-based test.

In distributed simulation, the real network is included so the system is simulated and tested over the real network. To further this test, real hardware on which the model will be executed can also be included into the simulation-based test. This is the hardware-in-the-loop (HIL) simulation as shown in step 3 of Figure 3.4. In HIL simulation, one or more models can be deployed to their hardware to be simulated and tested. In the example of Figure 3.4, *Model1* along with its real-time execution engine stay on the real hardware. *Model2*, the environment model, and their real-time simulators stay on other computers. These models still keep the same couplings. However, the model on the real hardware may use some or all of its sensors/actuators to interact with the real world. Similar to the description of section 3.3.2, different configurations can be applied to test different aspects of the model. Another valuable benefit of HIL simulation is that it allows a subsystem to be tested without waiting for all other subsystems to be completely built. This is because the HIL simulation still works within the virtual testing environment that may provide virtual subsystems. As a result, in HIL simulation, real and virtual subsystems can work together to conduct a meaningful system-wide test. To give an example let's consider the design of a distributed robotic system that includes hundreds of mini mobile robots. With this HIL simulation approach, one or several real robots can be tested and experimented with other hundreds of virtual robots that are simulated on computers. A detailed example of how to set up this kind of "robot-in-the-loop" simulation is given in Chapter 5.

The final step is real system test, where all models are tested on the real hardware within the real environment. As shown in step 4 of Figure 3.4, DEVS real-time execution

engines execute the models and take care of the underlying network synchronization/coordination. The environment model is no longer needed as the system is tested in the real environment. This is also the same setup as that in real execution where all models interact with the real environment through sensor/actuator *activities*.

3.4 The Development Process of the Methodology

While the preceding sections present how model continuity can be achieved and how step-wise simulation-based test methods can be applied, this section describes the whole development process of the methodology. This process is useful to provide a map that guide designers to develop real-time software step-by-step. Figure 3.5 shows this process.

The development process starts from an early *system requirement analysis*. Based on that, the first step is to *identify the system and its external environment*. This includes identify which part belonging to the system that need to be designed, and which part belonging to the external environment within which the system will operate. By clearly separating the system from its environment, the designers can go ahead to *develop the environment model* as shown by the yellow box in Figure 3.5. Note that the question of at what abstract level to model the environment is dependent on the test objectives, which come from the system requirement analysis and can be refined iteratively through the development process.

For the system to be designed, the next step is to *identify subsystems* (non-distributed systems do not need this step). The goal of this step is to identify how many subsystems the system has, how these subsystems are connected to each other, and what kind of

network supports the communication between subsystems, *etc.* Based on this analysis, the designers can go ahead to *develop the network delay models*, which will be used in central simulation-based test.

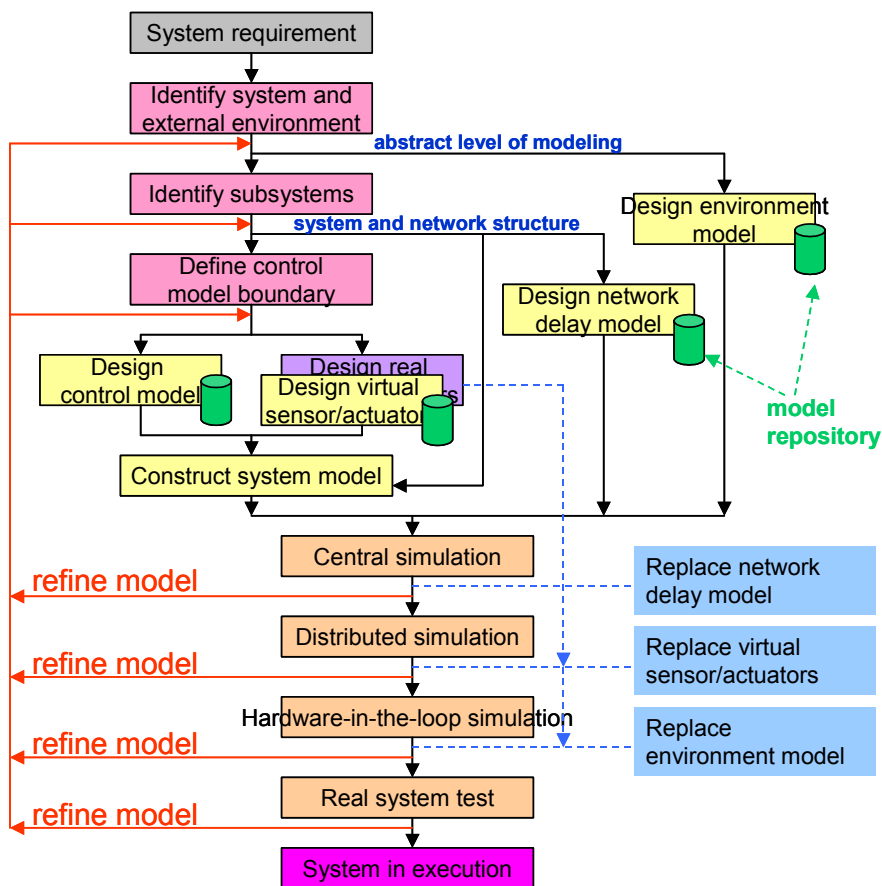


Figure 3.5. Development Process of the Methodology

For each subsystem, the next step is to *define the control model boundary*. This is because the methodology clearly separates the control model from the hardware interfaces (sensor/actuator interfaces). The control model is the one that will be maintained continuity from simulation-based design to real execution; the sensor/actuator interfaces are the interfaces between the control model and external environment. Once

the boundary of the control model is determined and the interface functions of sensors/actuators are defined, the designers can start to *develop the control model*. In the meantime, the designers can start to *develop real and virtual sensors/actuators*. The real sensors/actuators are the ones that drive the real hardware and will be used in real execution; the virtual sensors/actuators are the ones that imitate the behavior of real sensors/actuators and will be used in simulation-based test. To support model continuity, the interface functions of a real sensor/actuator and its corresponding virtual sensor/actuator should be the same.

After the models for each subsystem are developed, the next step is to *construct the system model* by constructing models (control model and virtual sensors/actuators) for each subsystem and then adding couplings between these subsystems. This step needs to use the system structure information from the “*identify subsystems*” step. Then based on the system model, the environment model, and the network delay models that have been developed, the stepwise simulation-based test process is applied to test the models incrementally. This process includes four testing steps.

The first testing step is *central simulation* where all models are simulated and tested on a central computer. The second testing step is *distributed simulation* where models of subsystems are deployed to different computers and simulated/tested in a distributed environment. Note that going from central simulation to distributed simulation, the network delay models are replaced by the real network. The third testing step is *hardware-in-the-loop simulation* where some of the subsystem models are deployed to the real target hardware and tested on the real target hardware. In hardware-in-the-loop

simulation, the model on the real target hardware may replace some of its virtual sensors/actuators with real sensors/actuators. The fourth testing step is *real system test* where all the models are deployed to their target hardware and tested in the real physical environment. In this case, the environment model is replaced by the real physical environment. A more detailed description of this stepwise simulation-based test process is given in section 3.3.

At the end of each testing step, the designers may need to go back to the early steps to change the design or to refine the models. So multiple iterations may be needed before reaching the final step, which is *system in execution*.

3.5 *abstractActivity*

3.5.1 *activity* and *abstractActivity*

As mentioned in section 2.4, an *activity* can be any kind of computation tasks for real-time execution. In the model continuity methodology, *activities* act as sensor/actuator interfaces that allow the control models to interact with the real environment. These sensor/actuator *activities* are only used in real execution. To allow the control models to be tested by simulation methods in a virtual testing environment, *abstractActivities* are developed. The basic idea of *abstractActivity* is to provide an interface so that the control model can interact with the environment model in simulation. To support model continuity, this interaction should be the same as that when the control model interacts with the real environment through *activity* in real execution. To make sure that a DEVS model can treat *abstractActivity* and *activity* in the same way, *ActivityInterface* is

developed to be implemented by both *activity* and *abstractActivity*. Below is this

ActivityInterface:

```
public interface ActivityInterface{
    public void setActivitySimulator(CoupledSimulatorInterface sim);
    public String getName();
    public void kill();
    public void start();
    public void returnTheResult(entity myresult);
}
```

A brief description of these methods is given below. For simplicity, below we use *Activity* to refer to both *activity* and *abstractActivity*.

- Method *setActivitySimulator()*: set the atomic model's simulator in *Activity*.
- Method *getName()*: get the name of an *Activity*.
- Method *kill()*: stop an *Activity*.
- Method *start()*: start an *Activity*.
- Method *returnTheResult()*: returns result to the DEVS model.

With *ActivityInterface*, the relationship among the environment, the control model, *activity*, and *abstractActivity* is shown in Figure 3.6.

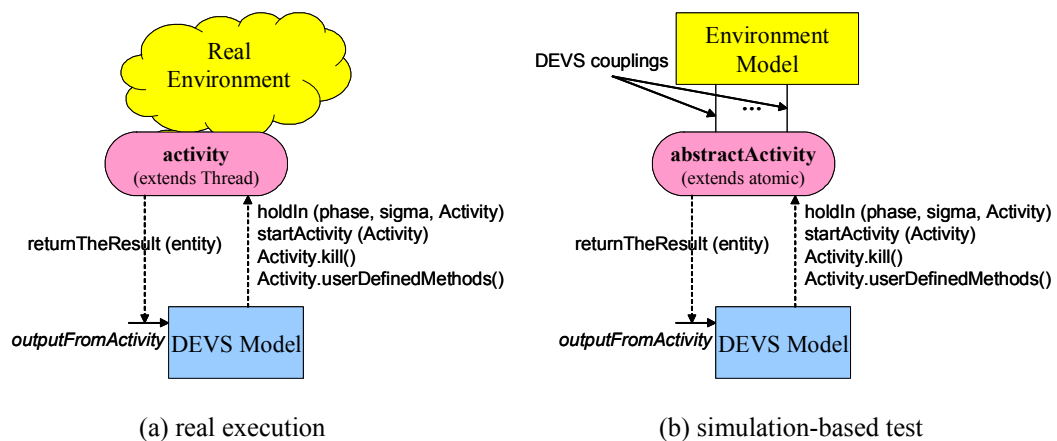


Figure 3.6. Environment, models, activity, and abstractActivity

Figure 3.6 shows that both *activity* and *abstractActivity* have the same interfaces with the DEVS control model. Specifically, a DEVS model can start an *Activity* by calling *holdIn()* or *startActivity()* methods. It can stop an *Activity* by calling *Activity*'s *kill()* method. An *Activity* can return computation result to the DEVS model by calling method *returnTheResult()*. This method sends the result as a message to the DEVS model's reserved input port *outputFromActivity*. This message, as an external event, triggers the model's external transition function *deltext()*, which processes the message and gets the result from *Activity*. Besides these standard-defined methods, Figure 3.6 also shows that an *activity* can have user-defined methods, such as *move()*, *stop()*, *etc.* This is because an *activity* is a thread (not a DEVS model) that can have arbitrarily defined methods. These same user-defined methods should also be defined by the corresponding *abstractActivity*.

In the current implementation, both an *abstractActivity* and the environment model are DEVS models. To allow interaction between *abstractActivity* and the *Environment* model, a method *addActivityCoupling()* is specially developed to add couplings between an *abstractActivity* and the *Environment* model so they can exchange messages. By calling this function, an *abstractActivity* establishes a "direct" communication channel with the *Environment* model so the message exchange between them does not interfere with the control models, which are the ones that need to be maintained with model continuity. To allow an *abstractActivity* to imitate the behavior of those user-defined methods that are defined by an *activity*, auxiliary functions *sendInstantOutput()* and *putInstantInput()* are developed. These two functions allow *abstractActivity* to generate

and pass DEVS messages to the *Environment* model. They provide the flexibility for an *abstractActivity* to imitate the behavior of its corresponding *activity*.

Below we describe the implementation principles to support *Activity* and its auxiliary functions. The description is roughly divided into three groups: interfaces between an atomic model and its *Activity*; interfaces between *abstractActivity* and the *Environment* model; and how an *abstractActivity* implements an *activity*'s user-defined functions.

3.5.2 Interfaces between an Atomic Model and *Activity* (*activity/abstractActivity*)

An atomic model can start an *Activity* using two methods: *startActivity(Activity)* or *holdIn(phase, sigma, Activity)* as shown below:

```
Activity a = new Activity(name);
holdIn("wait",5, a); // or startActivity(a);
```

The difference between *holdIn()* and *startActivity()* is that *holdIn()* changes the *state* and *sigma* of the atomic model while *startActivity()* doesn't. Method *holdIn()* is typically used when a model starts an *Activity* and wants to monitor when or if the desired result returns. For example, if the model notices that the *sigma* elapses and *Activity*'s result doesn't come, it may handle this situation as an exception so an "exception handling routine" is called to kill the *Activity* or to wait for longer time. Unlike *holdIn()*, method *startActivity()* allows an atomic model starts an *Activity* without interrupting its current *state*. Below shows how these two methods are implemented:

```
public void holdIn(String phase, double sigma, ActivityInterface a) {
    this.phase = phase;
    setSigma(sigma);
    this.startActivity(a);
}
```

```

public void startActivity(ActivityInterface a){
    this.a = a;
    mySim.startActivity(a);
}

```

As can be seen, the *holdIn()* method set the *phase* and *sigma* of the atomic model and then call *startActivity()*; the method *startActivity()* basically calls the *startActivity()* method of the corresponding simulator (*coupledSimulator* or *coupledRTSimulator*). Below shows this method of *coupledRTSimulator*:

```

public void startActivity(ActivityInterface a){ // of coupledRTSimulator
    if(a instanceof activity){
        a.setActivitySimulator(this);
        a.start();
    }
    else if(a instanceof abstractActivity){
        ((atomic)myModel).addModel((abstractActivity)a);
        a.setActivitySimulator(this);
    }
}

```

The method first distinguishes if this is an *activity*, which is a thread, or an *abstractActivity*, which is an atomic model. If it is an *activity*, the method calls *activity*'s *start()* to start the thread (executing the *run()* function). If it is an *abstractActivity*, the method adds the *abstractActivity* (an atomic model) into the system by executing method *addModel()*. As will be described in the next chapter, this method creates a simulator for the added model and then initializes and starts the simulator. For both cases, *Activity*'s *setActivitySimulator()* is called so the *Activity* has a reference to this simulator (denoted by *modelSim*). As will be described below, this reference is used by *returnTheResult()* to send result back to the atomic model.

During simulation or execution, an atomic can stop an *Activity* by calling the *kill()* method. This is shown below:

```

public void kill(){           // of activity which is a thread
    interrupt();
}

public void kill(){           // of abstractActivity which is an atomic model
    removeModel(getName());
}

```

As mentioned before, an *Activity* can return the computation result to the atomic model by calling method *returnTheResult()*:

```

public void returnTheResult(entity myresult) {
    modelSim.returnResultFromActivity(myresult);
}

```

This method calls the simulator's *returnResultFromActivity()* method and pass the result as a parameter:

```

public void returnResultFromActivity(EntityInterface result) {
    content c = new content("outputFromActivity",(entity)result);
    putMessages(c);
}

```

Based on the result, method *returnResultFromActivity()* creates a message and put this message on the atomic model's reserved input port "*outputFromActivity*". The message will be handled by the model's external transition function *deltext()*. A sample *deltext()* to handle the *Activity* event is given below:

```

public void deltext(double e,message x){
    .....
    if (messageOnPort(x,"outputFromActivity",i)){
        entity ent = x.getValOnPort("outputFromActivity",i);
        ..... // process the result
    }
    .....
}

```

3.5.3 Interfaces between *abstractActivity* and Environment Model

The *Environment* model is the major part of a virtual testing environment. It imitates how the real environment reacts to the input of the system under development. Because the environment model is developed for the purpose of test, it will not be included in real execution of the final system⁵. In current implementation, in order to avoid interference with the control models of the system, the environment model is only allowed to be added as an independent component of the system's coupled model. For example, in Figure 3.7, *Coupled* is the system model; *Coupled1* is the model that needs to be tested. In this example, the *Environment* model can only be added as a component of *Coupled*. It cannot be added as a component of *Coupled1*.

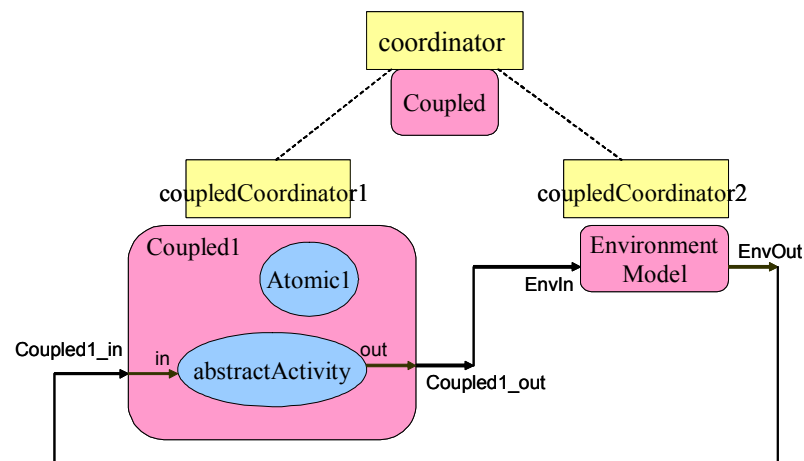


Figure 3.7. “Hierarchical” coupling between *abstractActivity* and environment model

While the *Environment* model always stays on the top level in the hierarchical tree of the system model, an *abstractActivity* that is started by an atomic model may be anywhere in the hierarchical coupled model. Thus a new auxiliary method

⁵ For more sophisticated examples, the system under development could have knowledge of its whereabouts and appropriate extension of the environment model could become a “real” component in the system

addActivityCoupling() is developed to allow adding couplings between an *abstractActivity* and the *Environment* model. Because we want to maintain the hierarchical property of the model, this new method adds the couplings between an *abstractActivity* and the *Environment* model in a hierarchical way. As a result, intermediate couplings are created in order to build the path from an *abstractActivity* to the *Environment* model. In the example of Figure 3.7, a path “out—Coupled1_out—EnvIn” is built by method *addActivityCoupling*(“*abstractActivity*”, “out”, “*EnvironmentModel*”, “*EnvIn*”); Similarly, *addActivityCoupling*(“*EnvironmentModel*”, “*EnvOut*”, “*abstractActivity*”, “in”) builds a path “*EnvOut*—Coupled1_in—in” as shown in Figure 3.7.

The implementation of *addActivityCoupling()* is shown below:

```
public void addActivityCoupling(String src, String p1, String dest, String p2){
    digraph P = (digraph)getParent();
    if(P!=null) P. addHierarchicalCoupling (src, p1, dest, p2);
}
```

This method first gets *abstractActivity*'s parent model (the *Coupled1* model in the example of Figure 3.7), and then calls parent's *addHierarchicalCoupling()* method which is shown below:

```
public void addHierarchicalCoupling(String src, String p1, String dest, String p2){
    coordinator PCoord = getCoordinator();
    if(withName(src)!=null){ //this is the output coupling
        if(withName(dest)!=null){ // we found the destination
            addPair(new Pair(src,p1), new Pair(dest,p2)); // add the coupling to the coupled model
            PCoord.addCoupling(src,p1,dest,p2); // update simulator's coupling information
        }
        else{ // doesn't find the destination, needs to go one layer upper
            String myName = getName();
            String myPort = src+"_"+p1; // output port = modelName + portName
            addPair(new Pair(src,p1), new Pair(myName,myPort)); //add external output coupling
            PCoord.addCoupling(src,p1,myName,myPort);
            digraph P = (digraph)getParent();
        }
    }
}
```

```

    if(P!=null) P.addHierarchicalCoupling(myName, myPort, dest, p2); //recursively call
  }
}
else if(withName(dest)!=null){ //this is the input coupling
  String myName = getName();
  String myPort = dest+"_"+p2; // input port = modelName + portName
  addPair(new Pair(myName,myPort), new Pair(dest,p2)); // add external input coupling
  PCoord.addCoupling(myName,myPort,dest,p2);
  digraph P = (digraph)getParent(); //go one layer upper
  if(P!=null) P.addHierarchicalCoupling(src, p1, myName, myPort);
}
else System.out.println("the source or the destination of the coupling couldn't be found!");
}
}

```

The *addHierarchicalCoupling()* method adds a coupling between two models in a hierarchical way. This method starts from the “bottom” model and goes up layer by layer until it finds the “top” model, which is the source or destination of the coupling. During this process, intermediate ports are created and couplings are added so a hierarchical path is formed. The method starts from the “bottom” model and first checks if this is an “output coupling” or an “input coupling”. An “output coupling” means the coupling is from this “bottom” model to other models; an “input coupling” means the coupling is from other models to this “bottom” model. If the source of the coupling is not *null*, this is an “output coupling”; otherwise if the destination of the coupling is not *null*, this is an “input coupling”; if both of them are *null*, the method prints out an error message. For the “output coupling” case, the method first checks if it can find the destination model on this level. If it finds the destination model, a coupling is established and the method terminates. Otherwise, the destination model is not on this level so the method goes to the upper levels to find it. Before it actually goes one level upper, the method creates a new output port for the parent model and adds a coupling from the source model’s output port to the parent model’s output port (an external output coupling). Then it recursively calls

the parent model's *addHierarchicalCoupling()* method until the destination model is found. As can be seen, this method dynamically establishes a path as it searches the destination model level by level. Similarly, for the “input coupling” case, external input couplings are dynamically established and the *addHierarchicalCoupling()* method is recursively called until the source model is found.

3.5.4 Implementing *activity*'s User-defined Functions in *abstractActivity*

An *activity* is a thread that can define its own functions, which can be called by the atomic model during execution. To support model continuity, an *abstractActivity* should have the same interface functions as those of *activity*. This means the *abstractActivity* also needs to implement those user-defined functions so in simulation an atomic model calls these functions (of *abstractActivity*), just like it calls them (of *activity*) in execution. Dependent on the complexity of those functions, different ways can be applied for the *abstractActivity* to implement them. For example, if a function is very simple and has no interaction with the real environment, the *abstractActivity* can have the function defined in the same way as that in *activity*. However, if a function of *activity* is complex or has interactions with the real environment, the same function of *abstractActivity* may also need to interact with the *Environment* model. As both *abstractActivity* and *Environment* model are DEVS models that deal with DEVS messages, two auxiliary methods are developed so that a function can generate and pass DEVS messages to these models.

These two methods are *sendInstantOutput(String outputPort, entity ent)* and *putInstantInput(String inputPort, entity ent)*. Method *sendInstantOutput()* puts a message

on *abstractActivity*'s output port. The message is then sent to the *Environment* model and thus triggers *Environment* model's *DeltFunc()*. Method *putInstantInput()* puts an instant message on *abstractActivity*'s input port, thus triggering its own *delttext()*. Below we use *sendInstantOutput()* as an example to see how it is implemented:

```
public void sendInstantOutput(String outputPort, entity ent){
    content ct = makeContent(outputPort,ent);
    if(mySim instanceof CoupledRTSimulatorInterface){
        message m = new message();
        m.add(ct);
        ((coupledSimulator)mySim).setOutput(m);
        mySim.sendMessagees();
    }
    else if(mySim instanceof CoupledSimulatorInterface)
        ((coupledSimulator)mySim).sendInstantMessages(ct);
    }
}
```

The method first checks if it is for real-time simulation or fast-mode simulation. If it is for real-time simulation, the method adds the message into the output message list and then calls simulator's *sendMessagees()* method. This method puts the message to the *Environment* model and notifies that there is external event so the corresponding models' external transition functions are executed. If it is for fast-mode simulation, the *coupledSimulator*'s *sendInstantMessages()* is called. This method looks similar as below: (Note: the *sendInstantMessages()* method of *coupledCoordinator* is the same as this one).

```
public void sendInstantMessages(ContentInterface c) {
    ..... // create message based on c
    Relation r = convertMsg(m);//convert message
    Iterator rit = r.iterator();
    while (rit.hasNext()){
        Pair p = (Pair)rit.next();
        content co = (content)p.getValue();
        Object ds = p.getKey();
        if(modelToSim.get(ds) instanceof coupledSimulator){ // the destination is found
            coupledSimulator sim = (coupledSimulator)modelToSim.get(ds);
            sim.putMessages(co);
            sim.DeltFunc(new Double(getCurrentTime()));
        }
    }
}
```

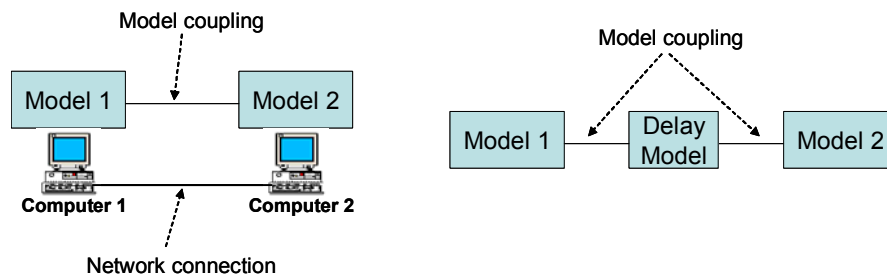
```

else if(modelToSim.get(ds) instanceof coupledCoordinator)
    ..... // similar as the case of CoupledSimulatorInterface
else // the message goes one level upper and is handled by the parent
    if(getParent() != null) ((coupledCoordinator)myParent).sendInstantMessages(co);
}
}

```

This method first tries to find the destination of the message. If it can't find it, the method recursively call the parent simulator's *sendInstantMessages()* methods. If the destination is found, the message is put to the destination model's input message list and the destination simulator's *DeltFunc()* is called so that the message is promptly handled.

3.6 Network Delay Model and *addCouplingWithDelay()*



(a) Execution of distributed coupled model (b) Central simulation of distributed coupled model

Figure 3.8. The *addCouplingWithDelay()* method

In the model continuity methodology, each component of a distributed real-time system is modeled as a DEVS model. The interactions between these distributed components are modeled as DEVS couplings, which are established by the method *addCoupling()*. In real execution, the models are mapped to network computers for execution and the couplings between models are kept. This is shown in Figure 3.8(a). Although a coupling in distributed execution is the same as that in central simulation, due

to the network latency, the time for a message to pass through this coupling is different. So to model the network latency, we have developed network delay models which hold any received messages for a period of delay time before the messages are sent out. These network delay models are used in central simulation to allow more authentic simulation-based test. A new method *addCouplingWithDelay()* is implemented to automatically create and add a network delay model in the middle of a coupling path as shown in Figure 3.8(b). As the *addCouplingWithDelay()* method makes this process transparent to the user, it eases the transition of models from central simulation to distributed execution -- a user only needs to change the code from *addCouplingWithDelay()* to regular *addCoupling()*. This *addCouplingWithDelay()* is shown below.

```
public void addCouplingWithDelay(IODevs src, String p1, IODevs dest, String p2, double delay){
    int UID = 0; // // find a unique ID for the DelayModel
    componentIterator cit = getComponents().cIterator();
    while (cit.hasNext()){
        IOBasicDevs iod = cit.nextComponent();
        if((iod.getName()).startsWith("DelayModel_")) UID++;
    }
    delayModel dM = new delayModel("DelayModel_"+UID, delay);
    add(dM);
    addCoupling(src,p1, dM, "in");
    addCoupling(dM, "out", dest, p2);
}
```

This method takes one more parameter *delay* as compared to the regular *addCoupling()* method. It first creates a new *delayModel* with two parameters: model name and the delay time. As the *addCouplingWithDelay()* method may be called multiple times, multiple *delayModel* may be created. To make the names of these *delayModels* unique, a variable *UID* is used as part of the model name. Then the *delayModel* is added

into the system and couplings are added from the source to the *delayModel*, and then from the *delayModel* to the destination.

The delay models are developed in such a way that any received messages will be held for a period of *delay* time (fixed or with arbitrary probability distributions) before they are sent out. For a delay model with fixed delay time, the messages will be sent out in exactly the same order as they are received. However, for a delay model with random distributed delay time, message may be sent out in a different order as they are received. We call the delay models that can preserve the order of messages *order-preserved delay models*. Network delay models typically are *order-preserved delay models*.

CHAPTER 4

VARIABLE STRUCTURE MODELING

4.1 System Evolutions and Adaptive Computing

Although system evolution is a common phenomenon for ecologic systems, its analogy in the computer world, adaptive computing is a relatively new research area [Vil97]. The main thrust of this research area so far has been on programmable/configurable hardware. For example, the programmable gate array (FPGA) technology has matured producing very high-density gate arrays (~1 million gates) with lower configuration times. The promise of adaptive computing however extends far beyond that. An adaptive computing system should be able to adapt to changes in environment at runtime, without compromising the consistency of the system. To that effect the main challenge that needs to be addressed is dynamic reconfiguration, which refers to the ability of a system to reconfigure its software or hardware components dynamically. Dynamic reconfiguration systems have the potential of realizing efficient systems as well as providing adaptability to system's changing requirements. Although both hardware and software components could be the ones to be reconfigured in a system evolution, this chapter mainly focuses on the reconfiguration of software components in the context of real-time systems.

The ability of dynamic reconfiguration brings several advantages for real-time systems. a) As real-time systems usually operate in dynamic, continuous changing, and

even unpredictable environments, dynamic reconfiguration allow those systems to adapt to the changing environment by reconfigure themselves accordingly. b) Some real-time systems, such as telecommunication switches require on-line upgrades, because off-line upgrades result in unacceptable delays and increased cost. For these systems, dynamic reconfiguration allows them to extend, customize or upgrade the services without the need for system recompilation or reboot. c) Dynamic reconfiguration is an often-used technique for real-time systems to achieve fault tolerance. This is because a system can reconfigure itself when a failure happens. Finally, for most real-time systems that have limited computing resources such as memory and battery, dynamic reconfiguration provides the flexibility for a system to configure only the necessary components for system operation. As a matter of fact, this idea is widely applied by reconfigurable RTOS kernels.

Motivated by these advantages, much research has been conducted to build middleware [Blair01, Kon01, DC00] and architectures [Dow01, Nee99, Gor02] that supports dynamic reconfiguration of software. Furthermore, research is also conducted to ensure these systems' safety and consistency [Pal], [Che02]. Most of these research works are based on component-based technology. This is because a component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture [Bro98]. A component system is built by composition of individual components and establishing relationship among them. As each component holds a high degree of autonomy and has well defined interfaces, dynamic reconfiguration of components can be achieved during runtime.

In general, there are six forms of reconfiguration of a component-based system [Che02]: addition of a component; removal of a component; addition of a connection between components; removal of a connection between components; update of a component; and migration of a component. The first four operations result in a structure change of the component-based system. The update of a component means a component is updated by a new version which might have totally different behavior or interfaces from the old one. This can be accomplished either by replacing the old version with a new one or by directly upgrading a component to a new version. Replacing a component involves the process of adding the new component and removing the old one, as can be realized by the addition and removal operations. The migration of a component actually implies two involved entities: a component and the location (physical or soft) of the component.

As mentioned in Chapter 2, DEVS supports a hierarchical modular modeling approach, which makes it possible for DEVS models to reconfigure themselves by adding/removing models or their couplings dynamically. In fact, this is referred as the variable structure modeling capability [Bar97a,Bar94,Uhr93,Uhr01,Paw02] in DEVS based modeling and simulation environments. This chapter presents our recent work on variable structure modeling (including adding/removing DEVE models, couplings, and ports) and its implementation in the DEVSJAVA environment.

4.2 Conceptual Development for Variable Structure in DEVS

4.2.1 Variable Structure Modeling

Variable structure models are the models that can dynamically change their model structure such as the inner components of the model and the connections between those components. Figure 4.1 gives an example that shows a simple process of structure change. In this example, the initial system has two components *A* and *B*. Then component *C* and the connection from *C* to *B* are added. After that component *A* is removed, resulting in a final system with two components *C* and *B*. Note that removal of a component will automatically remove all the connections related to that component.

In DEVS-based modeling and simulation environments, DEVS models are the components and DEVS couplings are the connections. Thus variable structure in DEVS means DEVS models and couplings can be added or removed dynamically. Corresponding to the four operations of structure change, four methods are provided in a DEVS environment. They are *addModel()/removeModel()* to add/remove DEVS models; *addCoupling()/removeCoupling()* to add/remove DEVS couplings. Note that the *addCoupling()* and *removeCoupling()* methods take four parameters: the source model, source model's output port, the destination model, destination model's input port. With these methods, the structure change process showed in Figure 4.1 can be realized as below:

- (1) *addModel(C);*
- (2) *addCoupling(C, COutputPort, B, BInputPort);*
- (3) *removeModel(A);*

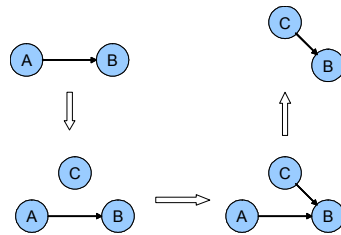


Figure 4.1. A variable structure process

Natural questions for variable structure systems arise concerning the authorization and timing of the structure changes. Generally speaking, there is no specific restriction on which component cannot initiate a structure change. However, because a DEVS coupled model does not have its own behavior, so an atomic model is needed to initiate a structure change. The initiation typically happens in the atomic model's internal or external transition functions. This is reasonable because a structure change is usually triggered by situation changes, which are captured as events in DEVS and are handled by the external or internal transition functions. In this sense, the atomic model acts as a supervisor to monitor the conditions of interest. For the system showed in Figure 4.1, component *B* could be the one to monitor system's situations and initiate the structure change. For example, it may monitor the input from *A*. If this input is less than a predefined value, it adds component *C* and the coupling from *C* to *B*. Then it monitors the input from *C* and if this input is greater than a predefined value, it removes *A*.

4.2.2 Operation Boundaries

Another important question for variable structure systems is how to determine the particular components that can be affected by a structure change operation. To answer

this question, we introduce the *operation boundary* concept and define it as the safe scope to conduct a meaningful operation. For example, in a distributed environment, a component can remove components on its local computer, but it is not allowed to remove components on remote computers. The latter violates the operation boundary of the remove operation in distributed environment. To support operations boundary in DEVS, models can maintain information on their locations in relation to the hierarchical structure of the overall coupled model. Components of the same coupled model, therefore belonging to the same parent, are called brothers. This approach is based on the structure knowledge maintenance concepts in [Zei89].

Thus the structure change operations also need to work within this hierarchical structure and to maintain this structure. Based on this, we define the operation boundaries of the four structure change operations as follows:

- *addModel(...)*: a model can only add components to its parent coupled model.
- *removeModel(...)*: a model can only remove itself and its brothers.
- *addCoupling(...)*: a model can only add couplings involving itself, its parent, and its brothers.
- *removeCoupling(...)*: a model can only remove couplings involving itself, its parent, and its brothers.

These clearly defined operation boundaries make it easier for a user to check if an operation is legal or illegal. For example, it can be easily seen that a model can remove itself, but it cannot remove its parent. This approach differs from that formalized by Barros [Bar97b] who restricts the ability to initiate change to a central network executive.

We find that much greater flexibility, at minimal cost, is achieved by allowing any component in a coupled model (or network) to initiate changes within the operations boundary.

Note that operations boundaries are defined in terms of model hierarchical structure independently of any distribution considerations. In distributed simulation, components reside on different computers and it is up to the distributed environment to ensure that the correct structure changes are carried out as prescribed by the structure modification commands. The distributed coupling change capability is supported by the DEVSJAVA environment. That is, couplings can be added or removed between models on different computers. It's up to the DEVS simulators to determine whether the coupling change is local or involves other computers. However, remotely adding/removing models in DEVSJAVA is currently not supported⁶.

4.2.3 Changing Port Interfaces

Besides structure change, another reconfiguration feature is provided in DEVS to allow an atomic model to add/remove input or output ports dynamically. For this purpose, the *addInport()* and *addOutport()* are provided for an atomic model to add new input and output ports respectively; the *removeInport()* and *removeOutport()* are provided for an atomic model to remove existing input and output ports respectively. As input and output ports are the interfaces of DEVS models, changing ports of a model usually requires that model's behavior also change accordingly. Thus, special attention has to be

⁶ Although it can be accomplished by sending a message to a remote simulator which then conducts the adding/removing operation locally, we have not completed the design details.

paid when adding/removing ports dynamically. The modeler has to ensure that if a model receives a new input (or output) port, the model has, or obtains, a corresponding way to handle the possible input received (or generated) on this port. In order not to violate the autonomy property of a component, we define the operation boundary of adding/removing ports as a model can only add/remove ports of itself and its brothers. Thus, atomic models inside a coupled model have the capability to modify the interfaces of their brothers, though the functionality to handle messages at those interfaces should be there or should be provided in the modified models. Particular ways of accommodating new ports are known. For example, one can make ports adhere to a labeling scheme such as *name+index* which can be analyzed and interpreted. As a new feature of DEVS variable structure, more research is needed to answer questions such as how to provide a general mechanism to update a model's external transition and output functions accordingly after the model's input and output ports are added/removed dynamically.

4.3 Examples of Variable structure

To illustrate the role of variable structure, two examples are presented in this section. The first example illustrates the ability to employ variable structure to dynamically emulate the system entity structure (SES). The second one describes an advanced workflow model which dynamically reconfigures itself by adding/removing models and changing the interface of models.

4.3.1 Dynamically Emulate the System Entity Structure (SES)

The System Entity Structure (SES) provides a way for specifying system composition[Zei90] with information about decomposition, coupling and taxonomy. It also provides a formal framework for representing the family of possible structures. From the design point of view, SES represents the design space with various possible design configurations. Thus the process of design/analysis is to prune SES, in other words, to search the best design configuration. For complex systems, the number of combination of different configurations is very large. Thus it is desirable to be able to emulate SES and automatically search the best design configuration. This section shows an example which demonstrates how this can be achieved by employing the variable structure capabilities.

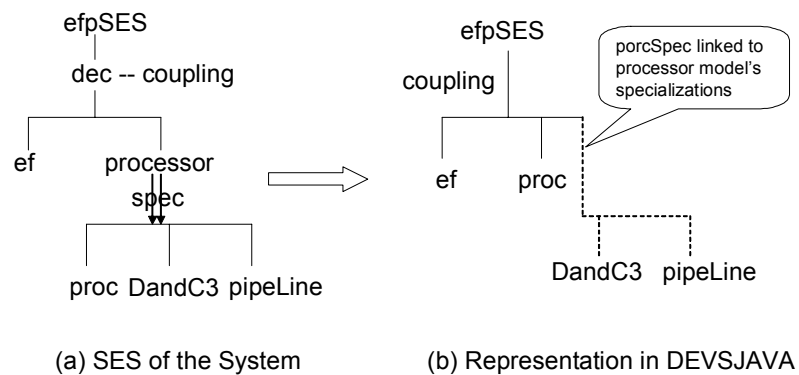


Figure 4.2: Dynamically Emulate the System Entity Structure (SES)

This example system is *efpSES* as shown in Figure 4.2(a). It has two components: an experimental frame model *ef* and a processor model which has three specializations representing three design choices of the system. The specializations of the processor model include a single processor *proc*, a divide and conquer processor *DandC3*, and a pipeline processor *pipeLine*. To automatically simulate all these alternatives of the

processor model, *efpSES* employs an instance of class *specEntity* to control the successive substitution of alternatives. *specEntity* is a specialized entity developed to emulate the SES of a system. In this example, the user defines *procSpec*, a subclass of *specEntity*, and provides it with the first and subsequent specializations: *proc*, *DandC3*, and *pipeLine*. Then as shown in Figure 4.2(b), the user adds *procSpec* to the coupled model and tells it which component to control (the dash line in Figure 4.2(b) shows *procSpec* is linked to processor model's specializations). Based on this information, during simulation the *procSpec* automatically replaces the processor model with different specializations until all of them are tested. Since the addition of local control components preserves hierarchical, modular structure, the hierarchical properties of the SES are automatically obtained. Moreover, this variable structure capability provides a general way to emulate the SES and automatically test all the alternatives of a system's design space as described in [Cou99].

While the SES involves only replacement of components by alternatives, the approach can be further extended to allow a restructuring executive to observe the simulation and make decisions regarding the alternatives to employ based on prevailing conditions. Such restructuring is discussed in the following example.

4.3.2 A Reconfigurable Workflow System

A simple workflow prototype is referred to as GPT.⁷ This is a coupled model that is composed of a *Generator*, a *Processor*, and a *Transducer*. It is the simplest self-contained

⁷ See *gpt.java* in the *SimpArc* package of *DEVJSJAVA*.

model that simulates three basic components of any workflow system. The *generator* generates jobs; the *processor* processes them, and the *Transducer* keeps track of the system state as a whole computing performance indexes such as system throughput (jobs processed per second) and average job turnaround time. In this section we describe a reconfigurable GPT system where *Processor(s)* can be dynamically added or removed and *Generator* and *Transducer* change their interfaces accordingly.

As shown in Figure 4.3(a), this system starts with the basic GPT components: *Generator*, *Proc1* and *Transducer*. *Generator* generates jobs and sends them out through *out1* port coupled to the *Proc1*'s *in* port. *Proc1* executes the job and sends the solved job to *Transducer* at *solved1* port. Note that the *Generator* has input ports: *add* and *addBank* and the *Transducer* has output ports *addModel* and *addProcBank* coupled to the two *Generator* ports, respectively. This suggests that the system has the capability to add a processor and a processor Bank.

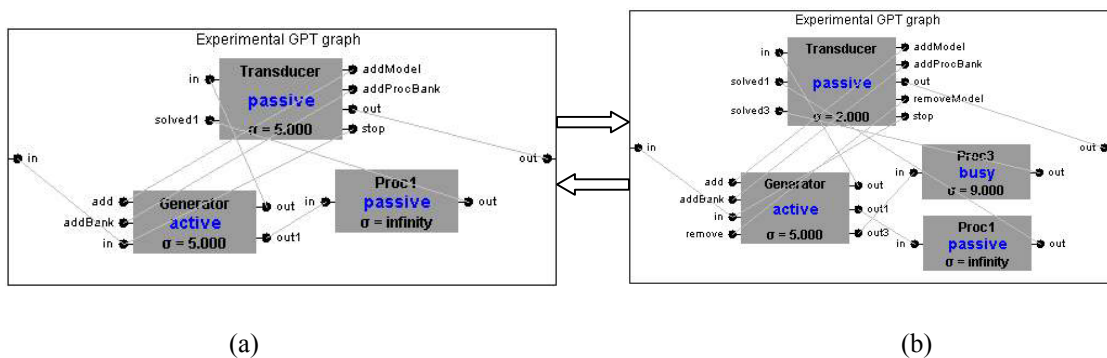


Figure 4.3: Stages of the Reconfigurable GPT System

In this example, the *Transducer* makes decisions of when to add or remove processor(s). The *Generator* executes the addition or removal operations. Thus, if the *Transducer* notices *Proc1* can't handle all the generated jobs, it sends out a message to

the *Generator*, which then adds another processor *Proc3*. As shown in Figure 4.3(b). *Proc3* is in a similar position as *Proc1* in the system. Note that the interfaces of *Generator* and *Transducer* also change accordingly. Besides the *Generator*'s earlier output port *out1*, a new output port *out3* has been added explicitly for *Proc3*. Similarly, the *Transducer* has added input port *Solved3* to collect jobs processed by *Proc3*. Also, the *Generator* and *Transducer* are now outfitted with ports for removing processor (*remove* and *removeModel* port). This is a new functionality that has been added in this stage. The interface change of *Generator* and *Transducer* is a reflection of the system's structure change. Initially there wasn't any functionality to remove any model as there was no need of it. As new processors are added so is the corresponding functionality to remove them. A typical set of commands that were executed by the *Generator* after receiving the addition message from the *Transducer* is:

```
mg = new modelProc("Proc"+index); // in this example, the value of index is 3
addModel(mg);
addOutport("Transducer","removeModel");
addInport("Generator","remove");
addOutport("Generator","out"+index);
addInport("Transducer","solved"+index);
addCoupling("Transducer","removeModel","Generator","remove");
addCoupling(getName(),"out"+index, ("Proc"+index," in"));
addCoupling(("Proc"+index,"out", "Transducer", "solved"+index);
```

Notice that a labeling scheme is used as the *Generator* model adds output port *out+index* for the new processor. Similarly, the *Transducer* handles the jobs solved by the processor using input ports with name *solved+index*. This allows expressing the *Transducer*'s processing by parsing port names to obtain their role and index parts, independently of the number of processors. The *Transducer* retains its basic behavior

independent of the structure change by providing the code in advance to handle the messages coming on new ports. More flexible approaches may be obtained by providing schemas that can be accessed at run time to support desired interfaces, a subject for further research.

In this example, after *Proc3* is added, it can also be removed when the *Transducer* thinks *Proc1* alone is enough to process all the generated jobs. To achieve this, the *Transducer* sends out a *removal* message using the *removeModel* port to the *Generator*. The *Generator* then removes *Proc3* and the system goes back to the initial stage. Similarly, a processor Bank (a coupled model) which contains multiple processors can also be added and removed.

From the above description, we can see that the system is able to expand itself, modify the interfaces of its components according to the structure change, and then shrink back to the original system. It displays a complete cycle of growth, from a basic functional level to an expanded system capable of high throughput and coming back to the initial state when its job (maximizing throughput) is done.

4.4 Implementation of Variable Structure in DEVS

The implementation of variable structure is based on the earlier development of DEVSJAVA modeling and simulation environment. So our discussion starts from a review of this environment, with emphasis on the hierarchical structure of DEVS models and simulators. Although a particular implementation environment is employed as basis,

the design is generic and can be employed in any hierarchical, modular DEVS environment.

4.4.1 Hierarchical Structure of DEVS Models And Their Simulators

In a DEVS modeling and simulation environment, there is a clear separation between models and their simulators. DEVS models are defined by the users to model the system under development. DEVS simulators are provided by the DEVS simulation environment to simulate or execute DEVS models. Corresponding to the hierarchical structure of a DEVS model, its simulators also form a hierarchical structure. Figure 4.4 gives an example which shows the relationship of a hierarchical coupled model and its corresponding simulators (the dash lines show the hierarchical relationship between simulators). This model has three components: *Atomic3*, *Atomic4*, and *Coupled1*, which has two sub-components: *Atomic1* and *Atomic2*. The simulators manage the information of the hierarchical coupled model in a hierarchical way. On the very top level, there is a *coordinator* assigned to the coupled model. This *coordinator* is the parent of all its sub-simulators, which have one to one relationship to the components of the coupled model. Following the hierarchical structure of the coupled model, there is a *coupledSimulator* assigned to each atomic model and a *coupledCoordinator* assigned to each coupled model. A *coupledCoordinator* acts as both a *coordinator* and a *coupledSimulator*. This is because it needs to communicate not only with its children (like a *coordinator*), but also with its parent and brothers (like a *coupledSimulator*).

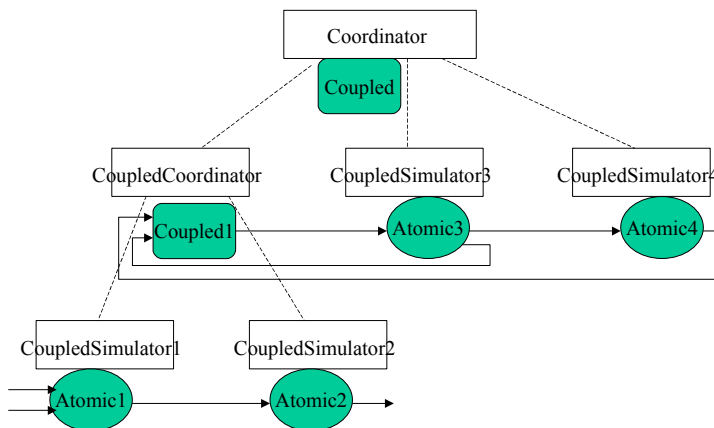


Figure 4.4: Relationship between models and their simulators (fast-mode simulation case)

This hierarchical structure of models and simulators requires several data structures to keep information so that the system can be efficiently implemented. Figure 4.5 shows the related data structures managed by simulators and models. This figure also shows that the *atomic* class implements *variableStructureInterface*, which defines the methods for adding/removing DEVS models, couplings, and ports. For simplicity, Figure 4.5 only shows the information related to the implementation of variable structure.

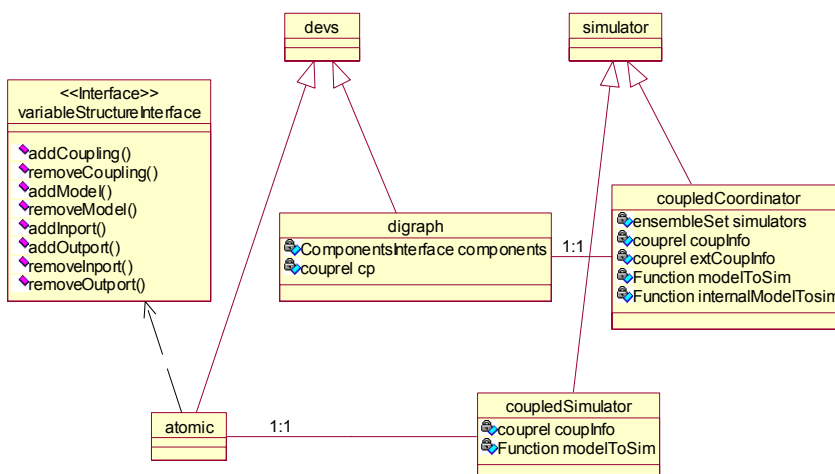


Figure 4.5: Methods and data structures used in variable structure implementation

First let's see the data structures managed by DEVS coupled models as shown by the *digraph* class in Figure 4.5 (atomic models don't need them). This is straightforward because coupled models need to keep track of their sub-components and the couplings among them. Thus, each coupled model has two variables as defined below:

- ComponentsInterface *components*;
- *couprel cp*;

The data structure for simulators can be categorized into three categories to store three different types of information as shown below:

- Children simulator info: *ensembleSet simulators*;
- Model's coupling info: *couprel coupInfo*, *extCoupInfo*;
- Model-simulator mapping info: *Function modelToSim*, *internalModelToSim*;

The first variable *simulators* is used by a *coupledCoordinator* (*coupledSimulator* doesn't use it) to store its children simulators. For example in Figure 4.4, the *simulators* variable for *coordinator* has three instances: *coupledCoordinator1*, *coupledSimulator3*, and *coupledSimulator4*. The *simulators* variable for *coupledCoordinator1* has two instances: *coupledSimulator1*, and *coupledSimulator2*. The second group of variables *coupInfo* and *extCoupInfo* are used by simulators to store models' coupling information. Specifically, *coupInfo* stores the couplings which start from a model and ends to the model's brothers or parent. *extCoupInfo* is used by *coupledCoordinator* (*coupledSimulator* doesn't use it) to store the couplings which start from a model and ends to the model's children models. Using *coupledCoordinator1* in Figure 4.4 as an example, the *coupInfo* has one coupling instance which starts from *Coupled1* and ends to *Atomic3*;

the *extCoupInfo* has two coupling instances. Both of them start from *Coupled1* and end to *Atomic1*. The third group of variables *modelToSim* and *internalModelToSim* are used by simulators to store the model-simulator mapping information. Again, using *coupledCoordinator1* in Figure 6 as an example, the *modelToSim* has three instances: (*Coupled1,coupledCoordinator1*), (*Atomic3,coupledSimulator3*), and (*Atomic4,coupledSimulator4*); the *internalModelToSim* has two instances: (*Atomic1,coupledSimulator1*), and (*Atomic2,coupledSimulator2*).

Note that in this implementation, each model and simulator manages its own copy of information. This approach relieves central coordinator's involvement in simulators' local activities. For example, by keeping a local copy of the coupling information, a simulator can send its model's output messages directly to the destination simulators. More information about this approach can be found in [Cho01,Cho03].

4.4.2 Add/Remove Coupling Dynamically

Because DEVS models and simulators use coupling data structures to keep all the coupling information, the basic idea to implement this feature is to update those data structures. Below we use *addCoupling()* to show how it works.

```
public void addCoupling(String src, String p1, String dest, String p2){
    digraph P = (digraph)getParent();
    P.addPair(new Pair(src,p1),new Pair(dest,p2)); //update its parent model's coupling info
    coordinator PCoord = P.getCoordinator();
    PCoord.addCoupling(src,p1,dest,p2); // update the corresponding simulator's coupling info
}
```

The method first gets its parent which is a coupled model. Then it calls its parent's *addPair()* method to update parent's coupling information, the *cp* variable as described in

section 4.4.1. To update the coupling information of the affected simulators, the atomic model then calls the *coordinator's addCoupling()* method. This method uses the source model's name to find the corresponding simulator and then update that simulator's coupling information, which is kept in the *coupInfo* or *extCoupInfo* variables. Note that for implementation convenience, the *getParent()* method is used. This method returns the parent model's reference which was established during simulation's construction stage. As this method is not accessible to the modelers, it doesn't violate the hierarchical modular property of DEVS models.

4.4.3 Add/Remove Model Dynamically

Adding a model dynamically means not only a new model is added, but also a new simulator needs to be created and added into the system. Furthermore, the new simulator needs to be initialized and synchronized with the ongoing simulation system. The *addModel()* method is shown below:

```
public void addModel(IODevs iod){
    digraph P = (digraph)getParent();
    P.add(iod);
    coordinator PCoord = P.getCoordinator();
    PCoord.setNewSimulator((IOBasicDevs)iod);
}
```

This method first adds the model as a new component to its parent by calling the *add()* method (update parent's *components* variable). Then it calls the *coordinator's setNewSimulator()* method. Depending on the type of simulation, the coordinators to control the simulation are different. For decentralized real-time simulation, *RTcoordinator* is used; for centralized real-time simulation, *RTCentralCoord* is used; for

fast-mode simulation, the regular *coordinator* is used. While *RTCentralCoord* and *coordinator* implement the *setNewSimulator()* in the same way, *RTcoordinator* implements it in a different way. Below we describe these difference.

Add model in decentralized real-time Simulation

RTcoordinator is used in decentralized real-time simulation. The *setNewSimulator()* method of *RTcoordinator* looks like below:

```
public void setNewSimulator(IOBasicDevs iod){
    if(iod instanceof atomic){ //do a check on what model it is
        coupledRTSimulator s = new coupledRTSimulator (iod);
        internalModelToSim.put(iod.getName(),s);
        simulators.add(s);
        //update all simulators' modToSim with the new internalModelToSim
        Class [] classes = {ensembleBag.getClass("GenCol.Function")};
        Object [] args = {internalModelToSim};
        simulators.tellAll("setModToSim",classes,args);
        s.initialize();
        s.simulate(numIter);
    }
    else if(iod instanceof digraph){
        coupledCoordinator s = new coupledCoordinator((Coupled) iod);
        ..... // same as when the model is atomic
    }
}
```

As can be seen, the method creates a new simulator based on the model type (atomic model or coupled model). It updates *RTcoordinator*'s corresponding data structures such as *internalModelToSim*, and *simulators*. Then it calls *simulators.tellAll("setModToSim", classes, args)* and passes *internalModelToSim* as parameter to update all simulators' *modelToSim*. Finally it initializes the created simulator and calls the *s.simulate()* to start that simulator. After these steps, the new simulator is created and started, and all other

simulators' related data structures are updated. So the simulation can go ahead with the new added model.

Add model in fast-mode simulation or centralized real-time simulation

Although the basic idea of *setNewSimulator()* method in fast-mode simulation or centralized real-time simulation is the same as that in decentralized real-time simulation, that is to create and start a new simulator and to update other simulators' data structures, the implementation of this method is a little bit different. Below we describe why we have to implement it in a different way.

Fast-mode simulation or centralized real-time simulation is controlled by central coordinators, which control the simulation based on the simulation cycle. A typical simulation cycle looks like below:

```
while( tN < DevsInterface.INFINITY) && (i<=num_iter) ) {
    computeInputOutput(tN);
    wrapDeltFunc(tN);
    tL = tN;
    tN = nextTN();
    i++;
}
```

Within this cycle, the *wrapDeltFunc()* method triggers all imminent simulators to execute their external or internal transition functions. Specifically, the execution of *simulators.tellAll("DeltFunc",classes,args)* makes that happen. This is shown below:

```
public void wrapDeltFunc(double time) {
    sendDownMessages();
    Class [] classes = {ensembleBag.getClass("java.lang.Double")};
    Object [] args = {new Double(time)};
    simulators.tellAll("DeltFunc",classes,args);
    input = new message();
    output = new message();
}
```

As mentioned in section 4.2, adding/removing models happen in an atomic model's external or internal transition functions. Thus, the execution of *simulators.tellAll("DeltFunc",classes,args)* will cause models to be added or removed which implies the *simulators* data structure itself needs to be updated. To avoid confliction, we implement the *setNewSimulator()* method in a different way from that in decentralized real-time simulation so that the *simulators* data structure will not be updated directly. Below shows *setNewSimulator()* method in fast-mode simulation or centralized real-time simulation:

```

public void setNewSimulator(IOBasicDevs iod){
    if(iod instanceof atomic){ //do a check on what model it is
        coupledSimulators = new coupledSimulator (iod);
        internalModelTosim.put(iod.getName(),s);
        newSimulators.add(s);
        s.initialize(getCurrentTime());
    }
    else if(iod instanceof digraph){
        coupledCoordinator s = new coupledCoordinator((Coupled) iod);
        ..... // same as when the model is atomic
    }
}

```

A new data structure *newSimulators* is used to store the created simulator. To add the elements of *newSimulators* to the *simulators* data structure, a new method *updateChangedSimulators()* is implemented as shown below. Note that in order to synchronize with the current simulation time, the *initialize()* method as shown in the *setNewSimulator()* takes the parameter of *getCurrentTime()* which returns the current simulation time.

```

public void updateChangedSimulators() { //for variable structure capability
    //check if there are added or removed simulators
    Iterator nsit = newSimulators.iterator();
    Iterator dsit = deletedSimulators.iterator();
}

```

```

if(nsit.hasNext()||dsit.hasNext()){
    // need to update the simulators and download the internalModelToSim to simulators
    while (nsit.hasNext()) simulators.add(nsit.next());
    while (dsit.hasNext()) simulators.remove(dsit.next());
    //download the new ModtoSim info to all the simulators
    Class [] sclasses = {ensembleBag.getClass("GenCol.Function")};
    Object [] sargs = {internalModelToSim};
    simulators.tellAll("setModToSim",sclasses,sargs);
}
// reset newSimulators and deletedSimulators to empty
newSimulators = new ensembleSet();
deletedSimulators = new ensembleSet();
}

```

This method checks if there are elements in *newSimulators* or *deletedSimulators* to update the *simulators* data structure (The *deletedSimulators* is set when removing model dynamically. This is the similar, but reverse case of adding model dynamically). Then it calls *simulators.tellAll("setModToSim",...)* update all simulators' *modelToSim* data structure. With this new method, the *wrapDeltFunc()* method is changed to execute *updateChangedSimulators()* method after *simulators.tellAll()*. It is shown below:

```

public void wrapDeltFunc(double time) {
    sendDownMessages();
    Class [] classes = {ensembleBag.getClass("java.lang.Double")};
    Object [] args = {new Double(time)};
    simulators.tellAll("DeltFunc",classes,args);
    input = new message();
    output = new message();
    updateChangedSimulators();
}

```

By calling the *updateChangedSimulators()* method, the coordinator updates the *simulators* and *modelToSim* data structures, thus the added model becomes eligible to participate in the subsequent simulation cycle, contributing to the determination of the global time of next event and able to receive inputs and generate outputs in the normal

manner. Further details on modification of the DEVS protocol needed for well-defined variable structure are given in [Zei97].

Remove model dynamically

Reverse to what adding a model means, removing a model dynamically means to remove a model and its corresponding simulator(s) from the system. It also implies removing all the couplings related to that model from the system. Below is the *removeModel()* method. The method is basically the reverse of what *addModel()* does. It first removes the model from the parent model, then it calls *coordinator/coupledCoordinator's removeModel()* method to remove the simulator of that model. One extra step here is the *removeModelCoupling()* method which removes all the couplings related to the model.

```

    public void removeModel(String modelName){
        digraph P = (digraph)getParent();
        coordinator PCoord = P.getCoordinator();
        PCoord.removeModelCoupling(modelName); // remove the couplings of that model
        IODevs iod = P.withName(modelName);
        P.remove(iod); // remove the model
        PCoord.removeModel(iod); // remove the simulator
    }

```

4.4.4 Add/Remove Coupling in Distributed Environment

Before we proceed to discuss how to implement the distributed coupling change capability, let's see how distributed simulation is implemented. Figure 8 shows a distributed example with the same model as in Figure 4.6. In this example, the three components of the coupled model: *Coupled1*, *Atomic3*, and *Atomic4* are distributed on three different computers. As can be seen, for each distributed component on a computer,

there is a client simulator assigned to it (*CoupledSimulatorClient* for atomic model; *CoordinatorClient* for coupled model). These clients connect to an *CoordinatorServer*, which may reside on another computer (The dashed circles mean different parts of the system reside on different computers). During initialization, the *CoordinatorServer* waits for connections from clients. For each client, the *CoordinatorServer* creates a *SimulatorProxy* to communicate with it. After all the connections are received, the *CoordinatorServer* establishes the *modelToSim* and *coupInfo* and download them to *SimulatorProxies*. As *modelToSim* and *coupInfo* are kept in *SimulatorProxies* (not in the client simulators), all messages sent between clients will be firstly passed to *SimulatorProxies*. For example in Figure 4.6, if *Atomic4* sends a message to *Coupled1*, the message will first be sent to *SimulatorProxy3*. Based on the *coupInfo* and *modelToSim*, *SimulatorProxy3* passes the message to *SimulatorProxy1*, which then sends the message to *CoordinatorClient1* (*Coupled1*).

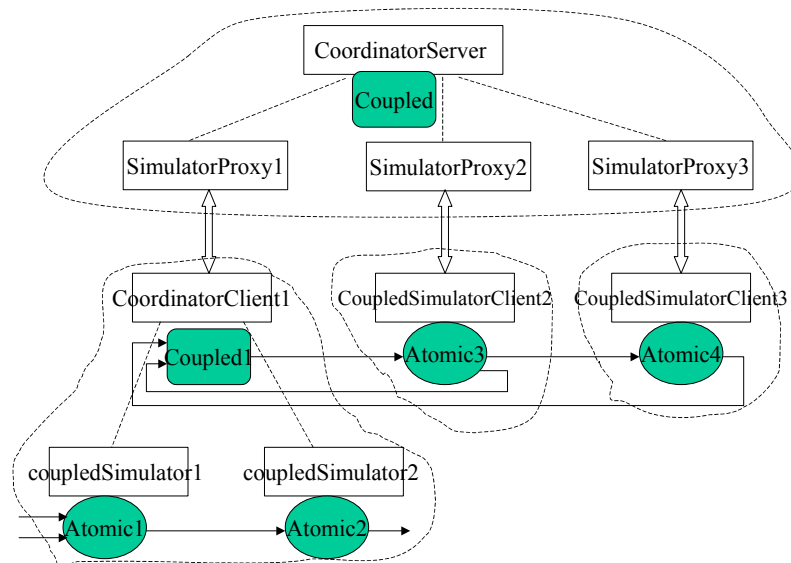


Figure 4.6: Models and their simulators in distributed simulation

As the coupling information of distributed models are kept in *SimulatorProxies*, so the basic idea of implementing distributed coupling change is to update those *SimulatorProxies*' coupling information. To implement this, whenever an atomic model wants to add or remove a distributed coupling, the *CoupledSimulatorClient* for that atomic model generates a distributed coupling change request and sends it to the *SimulatorProxy* as shown below:

```
public void addDistributedCoupling(String src, String p1, String dest, String p2){
    String dcc = Constants.addCouplingSymbol+": "+src+": "+p1+": "+dest+": "+p2;
    client.sendMessageToServer(dcc);
}
```

On the *SimulatorProxy*'s side, the *waitForMessageFromClient()* method is modified so that it can handle the distributed coupling change request. This method is shown below:

```
protected void waitForMessageFromClient() {
    String string = readMessageFromClient();
    //check to see if the message is a dynamic coupling change message
    if(string.startsWith(Constants.addCouplingSymbol)||
        string.startsWith(Constants.removeCouplingSymbol))
        DynamicCouplingStrReceived(string);
    else{ // this is a regular DEVS message
        ..... // process the message
    }
}
```

The method checks to see if the received string starts with *addCouplingSymbol* or *removeCouplingSymbol*. If that is true, the received string is a distributed coupling change request, so the *DynamicCouplingStrReceived()* is called. Otherwise, the received string is a regular DEVS message so the method processes it as usual. The *DynamicCouplingStrReceived()* method processes the string to get the source, the

source's port, destination, and the destination's port of the coupling. Then it call *CoordinatorServer's addCoupling()* or *removeCoupling()* methods to update the coupling information of *SimulatorProxies*.

4.4.5 Add/Remove Ports

The operation of adding and removing ports dynamically is done by:

- *addInport(String modelName, String portName),*
- *addOutport(String modelName, String port),*
- *removeInport(String modelName, String port)*
- *removeOutport(String modelName, String port)*

The functionality of modifying interfaces exists just at one horizontal level and is not present a level above (parent level) and a level below (brothers children). This restricts the ability of a model to alter the dynamics of the system to within its operations boundary. As mentioned above the four forms of adding/removing inports/outports take the *modelName* as a parameter referring to the destination model to which the change is desired. The functioning of these methods can be seen in the *reconfigurable* GPT model.

Internally, they are implemented as:

```
public void addInport(String modelName, String port){
    digraph P = (digraph)getParent();
    IODevs iod = (IODevs)P.withName(modelName);
    if (P != null){
        if (iod instanceof atomic)
            iod.addInport(port);
        else
            ((digraph)iod).addInport(iod.getName(),port);
    }
}
```

The above function adds an input port to the model specified by the *modelName*. Inside the function the models is accessed through the common parent (as they are brothers) and if its an instance of atomic, then the port is added here directly, otherwise the corresponding function in the digraph model is called, which adds the *port* to this brother digraph.

The mechanics of *addOutport()* is exactly same as that of *addInport()*. For the removal of ports, internally they are implemented in the same manner as the code described above except that the line *iod.addInport(port);* is replaced by the line *iod.removeInport(port)* where the variables have their usual meaning. Same thing happens in the case of *removeOutport()* which is implemented on the same lines with the change in the line mentioned above (*iod.removeOutport(port)*).

CHAPTER 5

DISTRIBUTED AUTONOMOUS ROBOTIC SYSTEM – A DYNAMIC TEAM FORMATION EXAMPLE

5.1 Distributed Autonomous Robotic Systems

Distributed Autonomous Robotic Systems (DARS) have been proposed in the last decade in a variety of settings and applied in different tasks. Special attention has been given to DARS developed to operate in a dynamic environment, where uncertainty and unforeseen changes can happen due to the environment and other agents that are external to the system itself. In the work of this dissertation, we view distributed autonomous robotic systems as a particular form of distributed real-time systems, with the systems interacting with external environments governed by their control models.

The field of DARS has gained growing research interests with a wide variety of topics being addressed. Surveys and summaries of the current state of the art for this field can be found in [Cao97, Par00, Ioc01]. For example, Parker [Par00] identifies eight primary research topics in multi-robot systems: *Biological Inspiration; Communication; Architecture, Task planning, and Control; Localization, Mapping, and Exploration; Object transportation and manipulation; Motion coordination; Reconfigurable robotics; Learning*. Iocchi [Ioc01] classifies the taxonomy of multi-robot systems into four levels of system structure characterization: *Cooperation Level, Knowledge Level (Awareness), Coordination Level, and Organization Level (Centralization, or Distribution)*. Both

[Cao97] and [Par00] identify some open issues in DARS. These issues include *defining metrics of various forms of cooperation, identifying characteristics of DARS, enabling effective human control, achieving scalability, etc.*

Although multiple robots potentially provide more robust and fault-tolerant services than a single robot, they also introduce extra software design and test complexity. First, unlike a single robot, the control schema of a distributed robotic system is distributed over multiple robots, thus making the cooperation and coordination among robots very important. Secondly, as a distributed robotic system interacts with the real world, the decision making of each robot not only needs to be logically correct but also needs to be timely in order to satisfy real time constraints. Moreover, with the advance of research and technology in this field, there is a continuous trend for distributed robotic systems to fulfill more complex tasks and scaled up to include more robots. All these factors make the software design and test for distributed robotic systems a challenging task, especially when a large number of mobile robots and task synchronization are involved. Systematic development methods and integrated development environments are needed to handle the design complexity of DARS [Par00, Wan97].

This chapter describes our work of developing a “dynamic team formation” robotic system using the model continuity methodology presented in Chapter 3. The task of team formation belongs to the research area of pattern formation and formation maintenance, and is one of the challenging issues in DARS. Related work can be found to address different aspects of this topic. For example, [Bal98], [Bal00] address the problem of physical implementation and formation keeping; [Car02] addresses systems’ robustness

in dynamic environments; [kow01] focuses on architectures of coordination. While these works address different aspects of robot formation, they all presume that robots know each other's existence before the systems are started. The system presented in this chapter, without this presumption, emphasizes the process of how two robots can form a team dynamically, starting from searching for each other, then establishing connections dynamically and finally conducting *leader-follower* march. Though this system only includes two robots at this time, with changes it can be scaled up to include more robots, where additional robots can be added into the team incrementally to form an indefinitely large convoy following the leader. The system demonstrates a dynamic structure process because the couplings between robots change dynamically during runtime. An integrated framework is particularly important to support the development of this kind of system, whose complexity would otherwise overwhelm the designers.

5.2 Hardware Description of the ACIMS Robot

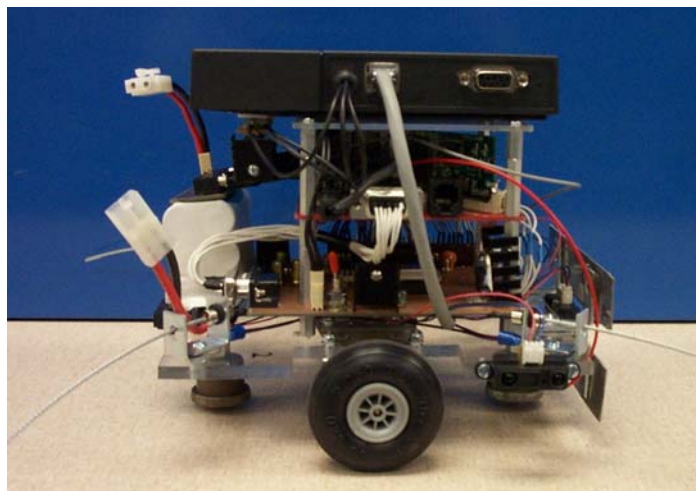


Figure 5.1: The ACIMS Mobile Robot

Figure 5.1 shows the type of robot that is used for the team formation task. It is a car-type mobile robot with wireless communication capability. The robot is built on a 20cm by 20cm Plexiglas base. The hardware of the robot has a three-layer structure, where each layer is responsible for different services [Pei02].

At the bottom is the hardware interface layer, which includes the motor and sensor interface (MSI) board and all the sensors and actuators. The MSI board uses a PIC16F877 processor from Microchip. This processor has eight analog inputs and twenty-four digital input/outputs. The three types of sensors that are implemented on the robots are the whiskers, infrared range finders, and wheel encoders. There are four whisker sensors located at each corner of the robot. The whiskers are mounted directly to the chassis and indicate to the PIC that they have been tripped when they are displaced from their resting position by approximately one inch or more. The purpose of the whisker sensors is to detect close range collisions and to avoid damaging the robot or nearby object. There are four infrared sensors around the four sides of the robot. These infrared sensors can detect obstacles in the range of 10 to 60 cm in the direction of the sensor. The infrared sensors return to the PIC an analogue signal indicating the distance to the nearest obstacle. Wheel encoders are the third type of sensor used by the robot. There are two wheel encoders that are mounted to each wheel of the robot and photo-reflectors mounted on the underside of each motor. The photo-reflectors bounce IR pulses off the wheel encoder disk and detect whether it sees black or white. The encoders each return a digital signal to the MSI board, which can then be used to maintain equal motor speeds, and to detect the distance moved by each wheel. Thus, the position and

orientation of the robot can be determined using the so-called dead reckoning method. The robot uses two DC motors mounted beneath the center of the chassis, which implement a differential drive-train. This mounting configuration allows the robot to move freely in both the forward and reverse direction as well as being able to rotate about its center. The shaft of each motor is connected to a three-stage planetary gear set which has an overall gear ratio of 80:1.

The second layer of the robot is the processing layer, which mainly includes the Tiny Network Interface (TINI) processor. The TINI has 1MB of nonvolatile SRAM and an apparent maximum execution speed of 120 Mhz. It also supports a JavaTM-programmable runtime environment and has its own Linux-like platform, which hosts its own FTP and TELNET server.

The third layer is the communication layer, including the 3Com Ethernet Client Bridge (ECB). This ECB is used by the TINI to connect to the wireless network over its Ethernet interface. The ECB operates on IEEE 802.11b wireless standard which allows it to communicate with most commercial access points. The ECB enables connectivity at the speed of 10Mbps, with a range of 150 feet from the robot to the access point, or from the robot to the next nearest robot.

The power board distributes power received from the battery to each of the three main components of the robot: the TINI, the MSI, and the ECB. Each of these components has its own voltage regulator that converts the voltage to around five volts. The battery supplies 3700 milliamp hours to the robot which under normal operation, the battery will

last two to three hours although the recommended operating time of the robot is two hours before having to be recharged.

5.3 The Dynamic Team Formation Process

This example consists two robots. It intends to show that connections between two robots can be established dynamically, and then communication and synchronization between them can be achieved. The development of this example demonstrates that a modeling and simulation framework, based on the DEVS formalism, can support model continuity and handle the development complexity for distributed robotic systems.

In this example, two robots (*Robot1* and *Robot2*) are put on a field with static objects such as walls, cabinets and boxes. The team formation process starts with both robots moving around and trying to find each other to establish a connection. (In our example, only *Robot1* is initially set to moving; *Robot2* is waiting). Both robots check their whisker sensors and infrared sensors regularly to see if there is any object around. At any time if a whisker sensor is tripped, the robot will react instantly by moving forward or backward in order to avoid collision or damage. Based on its infrared sensor data, a moving robot will turn around when it detects there are obstacles ahead.

Initially, there is no direct connections between two robots. Both robots are connected to a *Manager* on a laptop. They regularly send their four-side distance data (from front, back, left and right infrared sensors) to the *Manager*. The *Manager* checks these distance data from two robots and see if there is any match between them. If there exists a match, for example, *Robot1*'s front distance data equals *Robot2*'s left distance data, this means

that *Robot1* is possibly heading to *Robot2*'s left side. However, it's also possible that both *Robot1*'s front sensor and *Robot2*'s left sensor are heading to some static objects. In order to check if two robots are really seeing each other, the *Manager* will stop both robots and then ask them to start a *Dance* process to recognize each other. Specifically, it asks one robot, saying *Robot2*, to move away. If the other robot, *Robot1* in this case, notices that there is a distance change on its corresponding side, it will notify the *Manager* that it noticed a moving object. Otherwise it will notify the *Manager* that it didn't notice any change. A negative answer from *Robot1* means two robots are apart so they will continue their search. A positive answer means two robots are seeing each other (because we assume there are no other moving objects in the field). So the *Manager* will establish connections between them and ask them to organize into a team.

Once connections are established, two robots will communicate directly to each other. First, they will line up to form a *Leader-Follower* relationship. Then they begin to march: one follows the other with the same movement. During the march, the *Leader* moves forward or turns around to avoid obstacles based on its infrared sensor data. The *Follower* is "blind" in the sense that it doesn't use any data from its sensors. Instead, the *Follower* gets movement parameters from the *Leader* and moves the same way as the *Leader*. The movements of both robots are synchronized to each other. During the movement, if robots lose each other, they will inform the *Manager* and go back to the searching stage as initially started.

5.4 Developing Models of the System

5.4.1 System Model and Dynamic Coupling Change

From the above description, three basic components can be recognized in this system: the *Manager* that resides on a laptop (computer), *robot1* and *robot2* that reside on mobile robots. Figure 5.2 shows the model of this system. In this system, the *Manager* is an atomic model and each *robot* is a coupled model. The coupling of the system is as follows: (*R1* stands for *robot1*; *R2* stands for *robot2* and *man* stands for *Manager*):

```
addCoupling(R1, "distanceData", man, "Robot1Data");
addCoupling(R1, "report", man, "Robot1Report");
addCoupling(man, "Robot1Check", R1, "Check");
addCoupling(R2, "distanceData", man, "Robot2Data");
addCoupling(R2, "report", man, "Robot2Report");
addCoupling(man, "Robot2Check", R2, "Check");
```

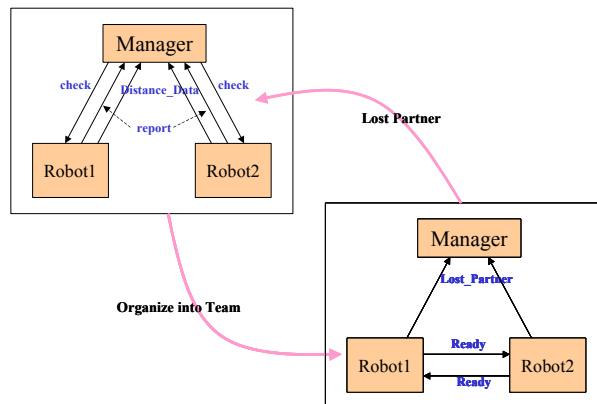


Figure 5.2: Model of Multi-Robot System

As can be seen there is no coupling between *robot1* and *robot2*. Each robot has output ports *distanceData* and *report*. These ports are coupled to *Manager*'s corresponding input ports. Meanwhile, the *Manager* has output ports coupled to each robot's input port *Check*,

so that *Manager* can ask them to check if they are within-line-of-sight. The robots return the check result using the *report* port. Once the report messages returned from the robots are both positive, this means two robots are close and they see each other. In this case, the *Manager* changes the couplings of the system dynamically to establish direct connections between the two robots. Specifically in this example, the manager executes the following DEVSJAVA code:

```
removeCoupling("Robot1", "distanceData", "Manager", "Robot1Data");
removeCoupling ("Manager", "Robot1Check", "Robot1", "Check");
removeCoupling ("Robot2", "distanceData", "Manager", "Robot2Data");
removeCoupling ("Manager", "Robot2Check", "Robot2", "Check");
addCoupling("Robot1", "readyOut", "Robot2", "readyIn");
addCoupling("Robot2", "readyOut", "Robot1", "readyIn");
```

Note that the *addCoupling* method is overloaded so it accepts strings to specify components in addition to object references. This feature makes it convenient for the modeler to keep track of models that have been added using string names. Explicit references can also be obtained from the parent coupled model by supplying the string names. This requires that all models be given unique names. After executing the DEVSJAVA code, bi-directional connections are established by coupling two robots' *Ready* port to each other, so two robots can communicate directly. The *distanceData* and *Check* couplings between robots and *Manager* are removed because they are no longer needed during the process of robot march. The *Report* coupling remains so robots can still inform the *Manager* in case they lose each other. During the march, if two robots

lose each other, they send the “*Lost Partner*” message to *Manager* using the *Report* port. This will trigger the *Manager* to add and remove couplings among the components. As a result, the system goes back to the situation as it is initially started, where two robots move independently and try to find each other.

5.4.2 Robot Model

The model of each robot is built based on Rodney A. Brooks’ Subsumption Architecture. As pointed out in [Bro86], classical AI usually runs into problems of extensibility (software or hardware), robustness (software or hardware), integration of multiple sensor devices and achieving multiple competing goals. Brooks however decomposes the problem of building autonomous vehicles into layers of desired behavior or *levels of competence*, rather than a sequential, functional form. Within this setting, he introduced the idea of subsumption, that is, more complex layer not only depended on lower, more reactive layer, but could also influence their behavior. The resulting architecture was one that could simultaneously service multiple, potentially conflicting goals in a reactive fashion, giving precedence to high-priority goals. The architecture was further developed into the behavior language [Bro90]. The first three levels defined by Brooks are as follows:

- Avoid contact with objects (whether the objects move or are stationary).
- Wander aimlessly around without hitting things.
- Explore the world by seeing places in the distance that look reachable and heading for them.

As can be seen each higher level contains as a subset each lower level of competence. The important part of this is that each layer of control can be built as a completely separate component and simply added to existing layers to achieve the overall level of competence.

In our example, the *Robots* move around trying to find each other. They begin the *Dance* process if the *Manager* thinks they are close. And they begin to *March* after they establish direct connection and organize into a “*Leader-Follower*” team. As such, we model each *Robot* as a coupled DEVS model. There are four components inside this coupled model (Figure 5.3):

- *Avoid* Model to avoid contact with objects.
- *Wander* Model to move around without hitting things.
- *March* Model to organize into a team and move in a “*Leader-Follower*” fashion.
- *Monitor* Model to check if two robots really see each other and report to *Manger*.

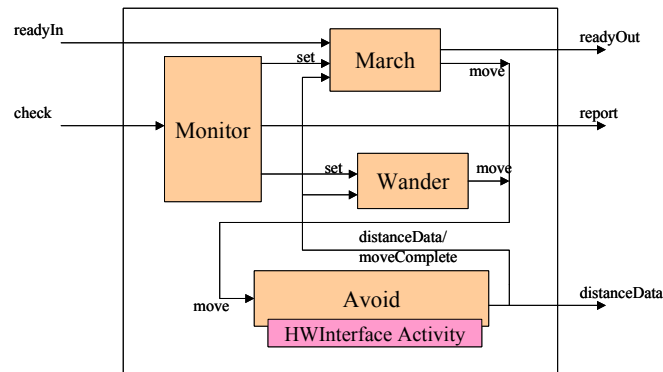


Figure 5.3: Model of Robot

To be more specific, in this example, the *Avoid* model always checks the whiskers to see if any of them has been tripped. If front whiskers are tripped, the *Avoid* model will

issue a command instantly to ask robot to move backward. If back whiskers are tripped, the *Avoid* model will ask the robot to move forward. We treat the situation that whiskers are tripped as an emergent situation, so the robot is supposed to respond to this situation immediately. As such, the robot's avoid movement has the highest priority. It can not be inhibited by other models.

In normal situation, the robot is either wandering or marching, controlled by *Wander* model and *March* model respectively. The *Wander* model follows a “*move--moveComplete-- move*” routine. Whenever a move is completed, the *Wander* model checks the distance data returned from robot's four Infrared sensors. If the front distance (to any object) is greater than a pre-defined threshold, saying 40cm, the *Wander* model will ask the robot to move forward. Otherwise, it will ask the robot to turn right by 90 degree.

Just as the *Wander* model controls each robot to move independently, the *March* model controls the robot system to move in a synchronized fashion. First, the *March* model asks the robotic system to line up to form a “*Leader-Follower*” team. Then the *Leader's March* model will check its infrared distance data, similar to the *Wander* model, and ask the robot to move forward if its front is clear and turn around otherwise. The *Follower's March* model does not use any of its sensor data. Instead, it gets move parameters from the *Leader* and conducts the same movement as the leader. The movements between the *Leader* and the *Follower* are not independent. They are synchronized in the sense that a robot cannot conduct the next movement until it gets the “ready” message from the other robot.

To switch between wandering and marching, the *Monitor* model is developed to check the current situation such as if two robots really see each other or if they lose each other. In the wandering state, whenever the *Manger* thinks that two robots are possibly seeing each other, it will send a “check” message to the *Monitor* model. The *Monitor* model then inhibits both *Wander* and *March* so it takes control of the robot. Then it asks robot to conduct a “*move and detect*” dance to check if two robots really see each other. In this dance, the *Leader’s Monitor* model asks the robot to moves away and the *Follower’s Monitor* model checks the corresponding distance data to see if there is any change. If the corresponding data changes, it means the *Follower* saw a moving object so two robots are seeing each other. Then the *Monitor* model will inhibit the *Wander* and start the *March*. Otherwise, the *Monitor* model will inhibit the *March* and resume the *Wander*. During the process of marching, the *Monitor* model is responsible to monitor if two robots lose each other. If they do, the *Monitor* model reports to *Manger*, meanwhile it will inhibit *March* and start *Wander*.

5.4.3 Hardware Interface *activity*

Just as the *Avoid*, *Wander*, *March* and *Monitor* models are responsible for the control logic of each robot; the *HWInterface Activity* (as shown in Figure 5.3) is responsible for sensor/actuator hardware interfaces. This *HWInterface Activity* communicates to the motor and sensor interface (MSI) board through RS232 serial interface. It reads sensor data periodically and issues commands to drive the motors. It is also responsible to collect *moveComplete* messages from the MSI board and pass them to the control models.

To avoid unnecessary message passing, quantization is used so no message passing is needed if there is no change in the distance data. In this example, the *Avoid* model starts this *HWInterface Activity*. As a result, all move commands from *Wander* model and *March* model are sent to the *Avoid* model, which acts as a relay to pass these commands to the *HWInterface Activity*.

The psuedo code of *HWInterface Activity* (the *HWActivity* class) is given below. In the code, *RobotControl* is the device driver class of the robot. Whenever *HWActivity* gets sensor data, it will call the *returnTheResult()* function, which puts the sensor data to *Avoid* model's *outputFromActivity* port as an external message, thus triggering *Avoid* model's external transition function (shown below). The function *setMovePara* is provided so that the *Avoid* model can call this function to drive the motors.

```

public class HWActivity extends activity implements RobotListener {
    .....
    public void run() {
        while (true) {
            Whisker_Data = RobotControl.checkWhiskers();
            returnTheResult(Whisker_Data);
            Distance_Data = RobotControl.getObsticalDistance();
            returnTheResult(Distance_Data);
        }
    }
    .....
    public void setMovePara(String direction, int speed, int distance){
        if(direction.startsWith("forward")) RobotControl.moveForward(speed, distance);
    }
}

```

```

else if(direction.startsWith("backward")) RobotControl.moveBackward(speed, distance);
else if(direction.startsWith("rotatecc")) RobotControl.CCRotation(speed, distance);
else if(direction.startsWith("rotatecw")) RobotControl.CWRotation(speed, distance);
}
.....
}

```

The psuedo code of *Avoid* model's external transition function is as following:

```

public void delttext(double e,message x){
.....
if (messageOnPort(x,"outputFromActivity",i)) {
    sensorData = x.getValOnPort("outputFromActivity",i);
    .....
    if (frontTripped) HWA.setMovePara("backward",avoidSpeed,avoidDist);
    else if (backTripped) HWA.setMovePara("forward",avoidSpeed,avoidDist);
}
else if (messageOnPort(x,"move",i)) HWA.setMovePara( direction, speed, distance);
.....
}

```

In the code, *HWA* is an instance of *HWActivity*. As can be seen, in its external transition function *delttext()*, the *Avoid* model processes the sensor data returned from *HWActivity* and will call *HWActivity*'s *setMovePara()* to move backward/forward if its front/back Whisker sensors are tripped. Meanwhile, if there is message on the *move* port

(sent from *Wander* model or *March* model), the *Avoid* model will call *HWActivity*'s *setMovePara()* to pass the move parameters.

To start the *HWActivity*, the *Avoid* model initializes an instance of *HWActivity* and calls the *startActivity()* method in its *initialize()* function. This is shown below.

```

HWActivity HWA;
HWA = new HWActivity();
startActivity(HWA);

```

5.4.4 Environment Model and *abstractActivity*

Simulation methods are applied in this example to test the correctness and efficiency of the robot system. In order to simulate and test the system, a simulation and testing environment is developed. This environment includes the *Environment* model to reflect how the real environment affects or is affected by the robot system, and the *HWInterface abstractActivity* to imitate *HWInterface Activity*'s behavior and interface functions.

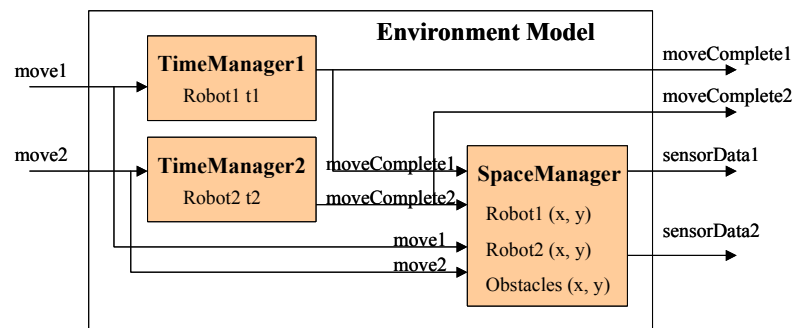


Figure 5.4: Environment Model

The main function of the *Environment* model is to model the time for a robot's movement. Meanwhile, the *Environment* model is also responsible to calculate the whisker sensor data and infrared sensor data and returns them to the control models

whenever a robot moves. As such, there are three components in the *Environment* model (Figure 5.4): *TimeManager1*, *TimeManager2* and *SpaceManager*. *TimeManager1* and *TimeManager2* model how long it takes for *Robot1* and *Robot2* respectively to finish a movement. Based on the moving distance and moving speed, they calculate the moving time and then issue *moveComplete* messages after that period of time elapses. To model the system in a more realistic way, random numbers are used when calculating the moving time. Below is the formula for moving time calculation of Robot_i (i=1,2).

$$\text{MovingTime}_i = \text{MovingDistance}_i / \text{MovingSpeed}_i + \text{Random}_i \quad (i=1,2)$$

(Note: *Random_i* is a random factor for Robot_i)

As *TimeManager* models robots' moving time, the *SpaceManager* models robots' moving space, which includes field shape and dimension, objects' shape, position and dimension, and robots' shape, position and dimension. Whenever the *SpaceManager* receives a *moveComplete* message from a *TimeManager*, it will update the corresponding robot's position (x, y) and direction (angle). For simplicity, in this example we have ignored the dynamic process of a movement. Instead, we treat each movement as discrete event so the position and direction of a robot are updated discretely. We think this simplification is good enough to serve our simulation and testing purpose.

Besides the *Environment* model, the simulation and testing environment also includes the *HWInterface abstractActivity* (the *abstractHWActivity* class), which act as an abstract sensor/actuator hardware interface to bridge between control models and the *Environment* model. This *abstractHWActivity* imitates the behavior and interface functions of the *HWActivity*, so the control model can treat it in simulation the same way as it treat the

HWActivity in real execution. Similar to the *HWActivity*, the *abstractHWActivity* regularly passes sensor data from the *Environment* model and provides the same interface function, *setMoveParameters()* to move the robot. Here is a code fragment:

```
public class abstractHWActivity extends abstractActivity{
    .....
    public void deltxt(double e,message x){
        .....
        if (messageOnPort(x," sensorData ",i) ) {
            sensorData = x.getValOnPort("sensorData ",i);
            returnTheResult(sensorData);
        }
    }
    .....
    public void setMoveParameters(String direction, int speed, int distance){
        moveParaString = direction + " " + speed + " " + distance;
        sendOutput("move",new entity(moveParaString));
    }
    .....
}
```

In our implementation, an *abstractActivity* is actually an atomic model. As can be seen in its external transition function *deltxt()*,*abstractHWActivity* handles the sensor data sent from the *Environment* model and then passes these sensor data to *Avoid* model. In the *setMoveParameters()*, it passes move parameters to the *Environment* model by calling the *sendOutput()*. In order to do so, in its *initialize()* method, *abstractHWActivity* adds couplings between itself and the *Environment* model as shown below:

```
addActivityCoupling (getName(),"move","Environment","move1");
addActivityCoupling ("Environment","sensorData1",getName(),"sensorData");
```

As mentioned in Chapter 3, the function *addActivityCoupling ()* is specially designed to add couplings between an *abstractActivity* and the *Environment* model so they can exchange messages. Note that the *deltxt()* of *Avoid* model remain the same because we maintain the same interface functions between *HWActivity* and *abstractHWActivity*. The

initialize() function of *Avoid* model is changed to initialize an instance of *abstractHWActivity* instead of *HWActivity*. This is shown below:

```
abstractHWActivity HWA;
HWA = new abstractHWActivity ();
startActivity(HWA);
```

5.5 Stepwise Simulations, Deployment, and Execution

We applied four steps to incrementally simulate and test this example before we deploy the models to real hardware for execution. As shown in Figure 5.5, these steps are central simulation, distributed simulation, robot-in-the-loop simulation, and real system test. The following text describes these steps in detail, including how to setup and what is the role for each step.

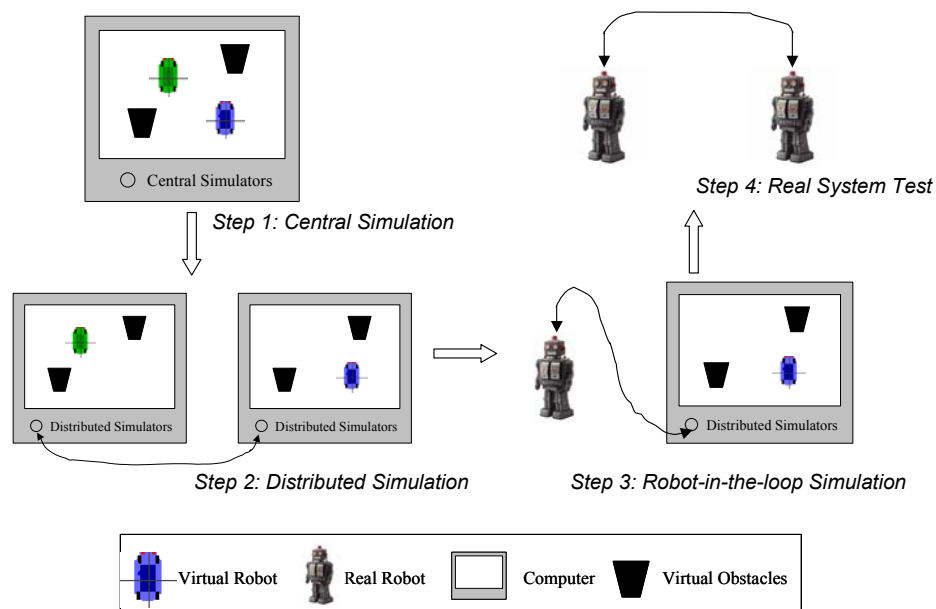


Figure 5.5: Simulation-based test of the “team formation” system

5.5.1 Step 1: Central Simulation

In central simulation, all the models, *Robot1*, *Robot2* (including their *abstractHWActivities*) and *Manager*, along with the *Environment* model reside in a single computer. Network delay models are used to model the network latency between robot models and the *Manager* model. These delay models are automatically added by the *addCouplingWithDelay()* method. As such, the constructor function of the system model *teamFormation* is shown below (Note that this piece of code uses 2 seconds as network delay, which needs to be changed based on the real delay of a network):

```
public teamFormation (String nm){
    double delay = 2; // 2 seconds delay
    Environment env = new Environment("Env"); add(env); //add the environment model

    Robot R1 = new Robot("Robot1"); add(R1); // add robot1
    Robot R2 = new Robot("Robot2"); add(R2); // add robot2
    Manager man = new Manager("Manager"); add(man); //add manager

    addCouplingWithDelay (R1,"distanceData",man,"Robot1Data", delay);
    addCouplingWithDelay (R1,"confirmOut",man,"Robot1ConfirmIn", delay);
    addCouplingWithDelay (man,"Robot1Explore",R1,"organize", delay);
    addCouplingWithDelay (man,"confirmOut",R1,"confirm", delay);
    addCouplingWithDelay (R2,"distanceData",man,"Robot2Data", delay);
    addCouplingWithDelay (R2,"confirmOut",man,"Robot2ConfirmIn", delay);
    addCouplingWithDelay (man,"Robot2Explore",R2,"organize", delay);
    addCouplingWithDelay (man,"confirmOut",R2,"confirm", delay);
}
```

We first employ the fast-mode simulator to simulate and test this *teamFormation* model. Based on the simulation result, we can trace problems such as why robot do not see each other even they stay closely and to analyze system properties such as how often robots will lose each other, *etc.* After fast-mode simulation, real-time simulator is employed to run simulation in a “timely” fashion. Within real-time simulation, a Graphic

User Interface was developed to show how robots move and react to the environment in real-time. This makes it easy to detect if the system operates as desired.

The psuedo codes to start the fast-mode simulation and real-time simulation are shown below respectively.

```
//fast-mode simulation
coordinator cs = new coordinator(new teamFormation ("teamFormation "));
cs.initialize();
cs.simulate();

//real-time simulation
RTcoordinator cs = new RTcoordinator(new teamFormation ("teamFormation "));
cs.initialize();
cs.simulate();
```

5.5.2 Step 2: Distributed Simulation

In distributed simulation, the three components of the system, *Robot1*, *Robot2* and *Manager*, are distributed on three computers as shown in Figure 5.6, step 2. (Note that for clarity, the *Manager* model that resides on a wireless laptop is not shown in Figure 5.5). The *Environment* model, which is not shown in Figure 5.6 either, can stay on one of the three computers or on a different computer. As the real network is used, the network delay models are no longer needed. Thus the *addCouplingWithDelay()* functions of the *teamFormation* model are replaced by *addCoupling()*. With these changes, the *teamFormation* model is shown below:

```
public teamFormation (String nm){
    Environment env = new Environment("Env"); add(env); //add the environment model

    Robot R1 = new Robot("Robot1"); add(R1); // add robot1
    Robot R2 = new Robot("Robot2"); add(R2); // add robot2
    Manager man = new Manager("Manager"); add(man); //add manager

    addCoupling(R1,"distanceData",man,"Robot1Data");
    addCoupling(R1,"confirmOut",man,"Robot1ConfirmIn");
    addCoupling(man,"Robot1Explore",R1,"organize");
```

```

    addCoupling(man,"confirmOut",R1,"confirm");
    addCoupling(R2,"distanceData",man,"Robot2Data");
    addCoupling(R2,"confirmOut",man,"Robot2ConfirmIn");
    addCoupling(man,"Robot2Explore",R2,"organize");
    addCoupling(man,"confirmOut",R2,"confirm");
}

```

Distributed real-time simulators are chosen to simulate and test the system in the distributed environment. As pointed out in Chapter 3, distributed simulation has to run in a real-time fashion. This is because part of the real physical world, the real network, is involved in this simulation-based test. In simulation, the two robots share the same virtual environment as depicted by the *Environment* model. So when *Robot1* moves, *Robot2*, which on a different computer, will notice it.

The psuedo codes to start the *RTCordinatorServer*, and clients for *Robot1*, *Robot2*, *Manager*, and *Environment* model are shown below respectively.

```

// start RTCordinatorServer with port 7000, simulation iteration number =1000
new RTCordinatorServer(new TeamFormation("Team"),1000,7000);

//Start RTCordinatorClient for Robo1 to connect to the ServerAddress with port 7000
new RTCordinatorClient(new Robot("Robot1"), ServerAddress, 7000);

//Start RTCordinatorClient for Robot2 to connect to the ServerAddress with port 7000
new RTCordinatorClient(new Robot("Robot2"), ServerAddress, 7000);

//Start RTCoupledSimulatorClient for Manger to connect to the ServerAddress with port 7000
new RTCoupledSimulatorClient(new Manager("Manager"), ServerAddress, 7000);

//Start RTCordinatorClient for Environment to connect to the ServerAddress with port 7000
new RTCordinatorClient(new Environment("Environment"), ServerAddress, 7000);

```

5.5.3 Step 3: Robot-in-the-loop Simulation

In robot-in-the-loop simulation, one or both of the models *Robot1* or *Robot2* are downloaded to real robots (Figure 5.1). We note that there is no need to transform the

model code in this example because the TINI chip that used by the robots supports Java-implemented DEVS real-time execution environment. Other models such as the *Manager*, and *Environment* models can reside on other networked computers and driven by the same DEVS distributed real-time simulator architecture. Depending on the desired configuration, the DEVS model resident on real robot may use robot's sensor/actuator hardware (*Activity*) to interact with the real environment or use *abstractActivity* as virtual sensor/actuators to interact with the *Environment* model. This extra flexibility allows us to test the code within the TINI environment with both simulated and real hardware (although it is the same logical code as in earlier test situations, its execution time characteristics may be quite different, due to the TINI chip's processing and memory limitations).

Below we consider an experimental setup to see how robot-in-the-loop simulation can be achieved. In this experiment, model *Robot1* is downloaded to a real robot *robot1* and is executed by a real-time execution engine that run on the TINI chip; model *Robot2*, *Manager* and the *Environment* are simulated on computers, among them *Robot2* uses *abstractHWActivity* to interact with the *Environment* model. We configure *robot1* to use its real motors to move within a real physical world, and to use virtual whisker sensors and IR sensors to get sensor data from the *Environment* model. To serve this purpose, *motorActivity* and *sensorAbstractActivity* are developed. The *motorActivity* defines the *setMovePara()* method that drives the real robot's motors; the *sensorAbstractActivity* combines the functions of virtual whisker sensors and virtual IR sensors. It gets the virtual sensor data from the *Environment* model and calls *returnTheResult()* to sends the

data to the *Avoid* model. The psuedo codes for these two classes are shown below respectively:

```

public class motorActivity extends activity implements RobotListener {
    .....
    public void setMovePara(String direction, int speed, int distance){
        if(direction.startsWith("forward")) RobotControl.moveForward(speed, distance);
        else if(direction.startsWith("backward")) RobotControl.moveBackward(speed, distance);
        else if(direction.startsWith("rotatecc")) RobotControl.CCRotation(speed, distance);
        else if(direction.startsWith("rotatecw")) RobotControl.CWRotation(speed, distance);
    }
    .....
}

public class sensorAbstractActivity extends abstractActivity{
    .....
    public void delttext(double e,message x){
        .....
        if (messageOnPort(x," sensorData ",i)) {
            sensorData = x.getValOnPort("sensorData ",i);
            returnTheResult(sensorData);
        }
    }
    .....
}

```

After defining the *motorActivity* and *sensorAbstractActivity*, the *Avoid* model can use them. Specifically, in its *initialize()* function, the *Avoid* model initializes an instance of *motorActivity* and an instance of *sensorAbstractActivity*. It then calls the *startActivity()* methods to start them. This is shown below.

```

motorActivity motorA = new motorActivity ();
startActivity(motorA);
sensorAbstractActivity sensorA = new sensorAbstractActivity ();
startActivity(sensorA);

```

Then, the *motorActivity* and *sensorAbstractActivity* are used by the *Avoid* model's external transition function *delttext()* as shown below:

```

public void delttext(double e,message x){
    .....
    if (messageOnPort(x,"outputFromActivity",i)) {

```

```

    sensorData = x.getValOnPort("outputFromActivity",i);
    .....
    if (frontTripped) motorA.setMovePara("backward",avoidSpeed,avoidDist);
    else if (backTripped) motorA.setMovePara("forward",avoidSpeed,avoidDist);
  }
  else if (messageOnPort(x,"move",i)) motorA.setMovePara( direction, speed, distance);
    .....
}

```

As can be seen, the *Avoid* model processes the sensor data returned from virtual sensors *sensorA*. It calls *motorA*'s *move()* to move the real robot backward/forward if the (virtual) whiskers sensors are tripped. Meanwhile, if there is message on the *move* port (sent from *Wander* model or *March* model), the *Avoid* model calls *motorA*'s *move()* to move the real robot. This example shows that the *Avoid* model uses its virtual sensor interface *sensorA* to get sensor data from the virtual environment (the *Environment* model) and uses its real motor interface *motorA* to move the robot. As a result, the real robot *robot1* moves in a physical field based the sensor data from a virtual environment. Within this virtual environment, *robot1* can “see” virtual obstacles and other robots, such as *Robot2*, which are simulated on computers.

The above experiment shows how robot-in-the-loop simulation is set up. In general, robot-in-the-loop simulation provides the flexibility to create test scenarios on computers to test how a control model works on a real robot.

5.5.4 Step 4: Real System Test

After passing these simulation-based tests, the next step is real system test, where all models are deployed to their target hardware and tested in a physical environment. In this example, models *Robot1* and *Robot2* are downloaded to the TINI chips on the respective

real robots while the *Manager* model is downloaded to a wireless laptop. For both robots, virtual sensor/actuator interfaces (*abstractHWActivity*) are replaced by real sensor/actuator interfaces (*HWActivity*). The *Environment* model is eliminated since the robots now tested in the real world. With these changes, the *teamFormation* model is shown below:

```
public teamFormation (String nm){
    Robot R1 = new Robot("Robot1");  add(R1);  // add robot1
    Robot R2 = new Robot("Robot2");  add(R2);  // add robot2
    Manager man = new Manager("Manager");  add(man);  //add manager

    addCoupling(R1,"distanceData",man,"Robot1Data");
    addCoupling(R1,"confirmOut",man,"Robot1ConfirmIn");
    addCoupling(man,"Robot1Explore",R1,"organize");
    addCoupling(man,"confirmOut",R1,"confirm");
    addCoupling(R2,"distanceData",man,"Robot2Data");
    addCoupling(R2,"confirmOut",man,"Robot2ConfirmIn");
    addCoupling(man,"Robot2Explore",R2,"organize");
    addCoupling(man,"confirmOut",R2,"confirm");
}
```

5.5.5 Deployment and Execution

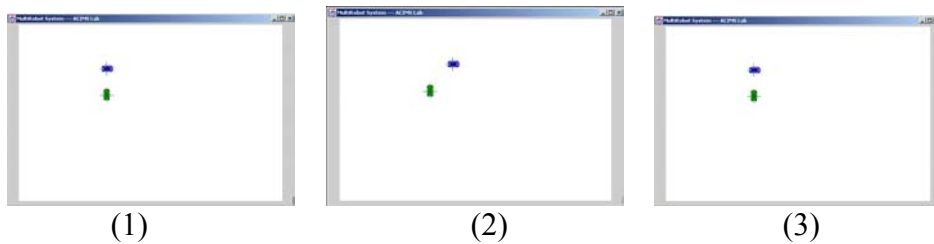
After passing these stepwise simulation-based tests, the final models are deployed to the hardware execution environment. The basic task of the deployment stage is to download models to their execution hardware. Same as configuration of real system test, *Robot1* and *Robot2* models are downloaded to real robots and the *Manager* model is downloaded to a wireless laptop. For both robots, real sensor/actuator interfaces (*HWActivity*) are used. The *Environment* model is not needed since the robots now operate in the real world. The final execution *teamFormation* model is the same as that in real system test.

Although at this time the deployment process is still not automated as we download each model to its execution hardware manually, a prototype type *Model Mapping Specification* as described in Chapter 2 is under development which intends to facilitate automated mapping of models to their execution hardware.

During execution, the two robots and the laptop, along with a wireless access point form a wireless network and are controlled by the DEVS real-time execution engines. Robots move in a physical field and response to a real environment. Some results and discussion are given in the following section.

5.6 Results and Discussion

One of the important results we are interested in is to check if robots move in real execution in the same (similar) way as they are simulated in simulation-based test. For this purpose, we have recorded a movie [MOV] which shows the process of robots forming a team and then conducting “leader – follower” march during simulation and execution. Some pictures taken from that movie are shown below. Figure 5.6 shows the pictures captured in simulation stage; Figure 5.7 shows the pictures captured in real execution.



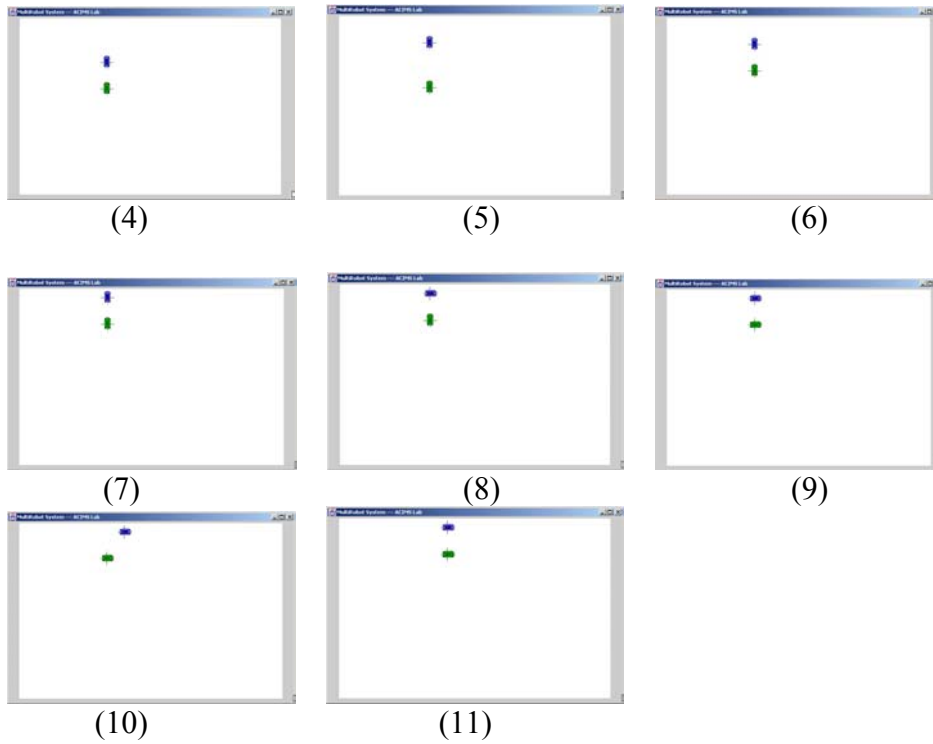
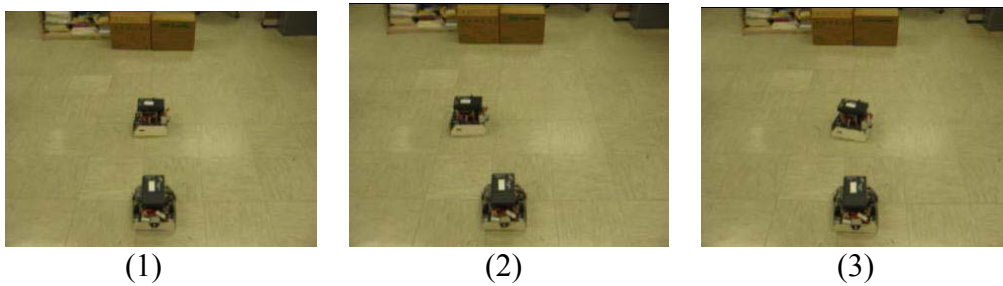


Figure 5.6: Simulation of robots

(1) Robots notice something. (2) Robot1 dances to move ahead. (3) Robot1 dances to move back. (4) “Yes, what I see is a robot!” Establish couplings dynamically then Robot1 turns to organize. (5) Robot1 marches ahead. (6) Robot2 marches ahead to follow. (7) Robot1 notices a wall ahead. (8) Robot1 turns to avoid. (9) Robot2 turns to follow. (10) Robot1 marches ahead. (11) Robot2 marches ahead to follow.



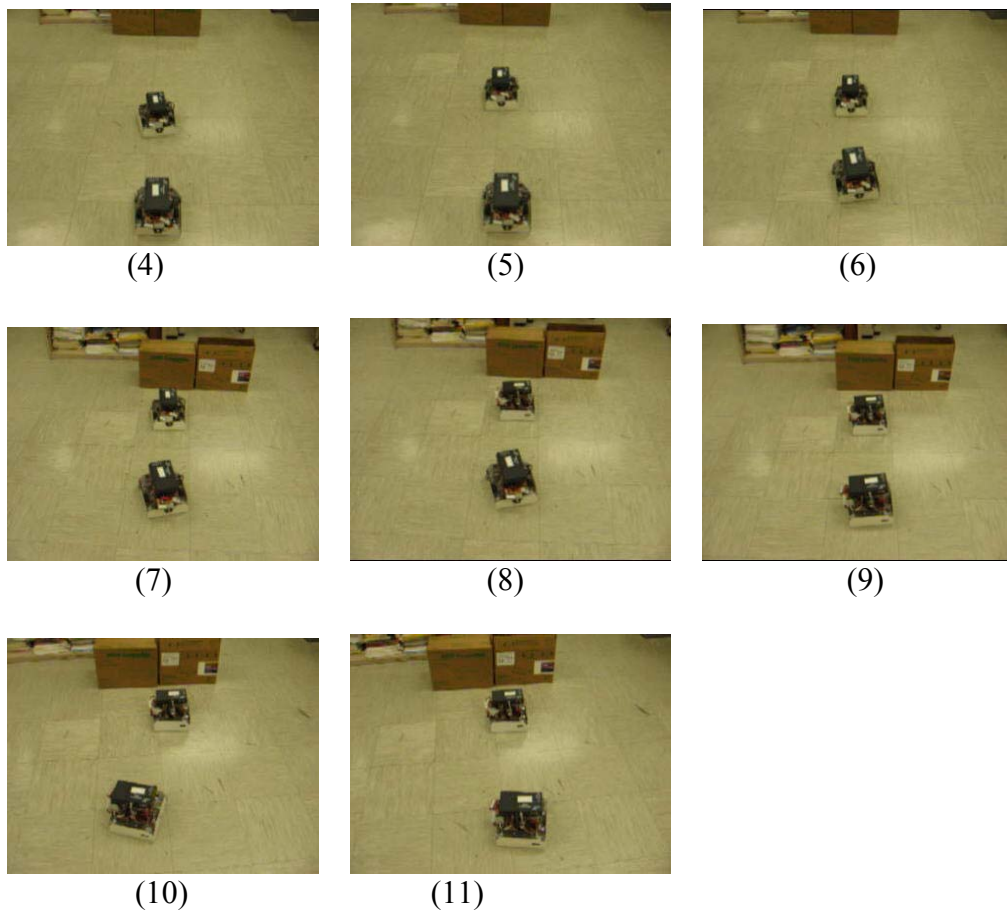


Figure 5.7: Execution of robots

(1) Robots notice something. (2) Robot1 dances to move ahead. (3) Robot1 dances to move back. (4) “Yes, what I see is a robot!” Establish couplings dynamically then Robot1 turns to organize. (5) Robot1 marches ahead. (6) Robot2 marches ahead to follow. (7) Robot1 notices a wall ahead. (8) Robot1 turns to avoid. (9) Robot2 turns to follow. (10) Robot1 marches ahead. (11) Robot2 marches ahead to follow.

As can be seen, this movie and the above pictures clearly demonstrate the continuity between the simulation and execution stages. We note that although the above example does not involve a complex environment setting, we expect this “continuity” will be preserved even in complex environments.

While the above system only includes two robots, more scalable systems with indefinite number of robots can be developed based on the same dynamic reconfiguration idea. Figure 5.8 shows an example with ten independent robots searching for each other, forming groups dynamically, and finally organizing into one large *Leader-Follower* team. During this process, couplings between models are added and removed, resulting in a variable structure system. The complexity and scalability of this kind of systems make a more persuasive case to apply the proposed model continuity methodology.

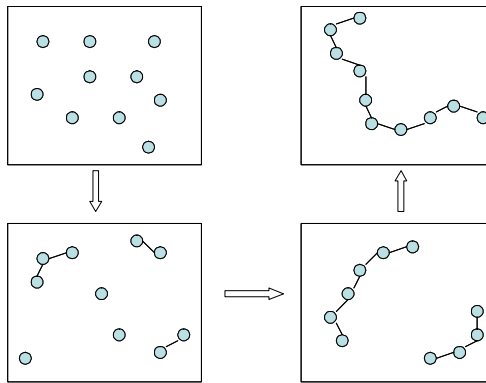


Figure 5.8: A Scalable Dynamic Team Formation Example

As mentioned before, the simulation-based test methods can not only test the correctness of control models, but also evaluate and analyze the performance of the system to be developed. For example, in the “team formation” example, the efficiency of different “search” schemas for robots to search each other before they organize into a team can be evaluated (We current employ a random search schema). A more detailed example of applying simulation-based analysis to evaluate system performance is given in the next chapter.

CHAPTER 6

A SCALABLE ROBOTIC CONVOY SYSTEM

6.1 A Description of the Robot Convoy System

The robotic system presented in Chapter 5 has two robots. This chapter presents a robot convoy system that essentially can include any number of mobile robots. We developed this system intentionally to demonstrate the scalability of the proposed software development methodology and to illustrate how the complexity of this kind of a large-scale system can be handled by the simulation-based “model continuity” methodology.

This robot convoy system consists of an indefinite number of robots, saying N robots ($N > 1$). These robots are in a line formation before the convoy begins. In this line formation, the leader robot is followed by its immediate “back” robot. The ender robot follows its immediate “front” robot. All other intermediate robots are connected both to its immediate “front” and “back” robots. Thus robots in this system are only directly connected to its neighbors. There is no global coordination in the system, although a more advanced system can be developed to have both local and global coordination. The robot used in this system is the same as that described in Chapter 5 – it can move forward/backward and rotate about its center, and has whisker sensors and infrared sensors.

During the team convoy process, robots are required to keep the line formation and to synchronize their movement. Here synchronization means a robot cannot move forward if its “front” robot doesn’t move, and a robot has to wait if its “back” robot doesn’t catch up. This synchronization feature is necessary for robots to move in a coordinated way, especially when robots are heterogeneous and have different moving speeds. To guarantee synchronization, synchronization messages are passed between a robot and its neighbors. Meanwhile, to facilitate a robot to follow its “front” robot, the moving parameters of a “front” robot are passed back so that a robot can conduct the same movement as that of its “front” robot.

During movement, if a robot’s whisker sensors are tripped, the robot instantly moves backward or forward to avoid the collision. Otherwise, in normal condition, the leader robot makes decisions to move forward or to turn around based on the distance data returned from its infrared sensors. If there is obstacle ahead, the leader robot turns around. Otherwise, it moves forward. All other robots conduct movement based on the moving parameters passed back from their direct “front” robots. Thus the leader robot’s movements are actually propagated backward robot by robot. For a perfect system in a perfect environment, this means each robot will follow exactly the same steps as the leader robot does. However, noise and variance exist in real executions. So if a robot simply “blindly” follows its “front” robot, eventually robots will lose each other and the system fails to keep the line formation. Thus to enhance the formation coherence of this convoy system, we have implemented an “adjust process”, which means a robot will adjust itself after every movement to make sure it still follows its “front” robot. Because

each robot knows the desired distance to its “front” robot, this “adjust process” uses the infrared sensor to check the distance after a movement and then makes necessary adjustment. Note that robots may head to different directions after some movements, so during the adjust process, a robot may need to turn an angle in order to find its “front” robot. This “adjust process” continues until a robot finds its “front” robot and adjusts itself to the right position/direction. Only after this process finishes can a robot send out its synchronization messages to its neighbors. In current implementation, due to the limited sensing capability, only infrared sensor data is used in this “adjust process”. This may not be enough in a complex environment because a robot may mistakably recognize an obstacle as a robot. For more advanced robots, other sensors, such as color sensors, may be used for a robot to track its “front” robot.

6.2 Models of the Robot Convoy System

6.2.1 System Model

Based on the above description, the model of this scalable robotic convoy system is developed as shown below:

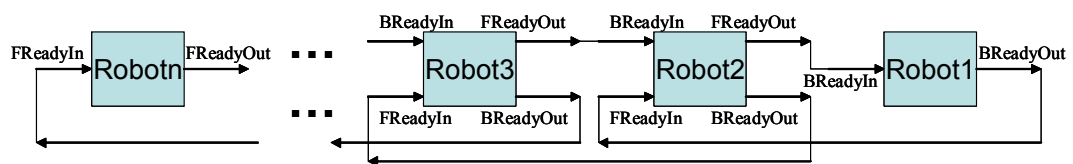


Figure 6.1. System Model of the Scalable Robotic System

As can be seen, this system includes N robots, which are modeled as DEVS coupled models. Among them *Robot1* is the leader; *Robotn* is the ender; others are intermediate

robots. For each intermediate robot model, there are two input ports: *FReadyIn*, *BReadyIn* and two output ports: *FReadyOut*, *BReadyOut*. For the leader and ender robots, only one input port and one output port are needed. These ports are used to send and/or receive synchronization messages between robots and to pass moving parameters from a “front” robot to the “back” robot. The couplings between robots are shown in Figure 6.1. Basically, a robot’s output port *FReadyOut* is coupled to its “front” robot’s input port *BReadyIn*; a robot’s output port *BReadyOut* is coupled to its “back” robot’s input port *FReadyIn*. Note that for simplicity, we have used the same output port *BReadyOut* of a robot to pass both synchronization messages and moving parameters to its “back” robot. The “back” robot is responsible to parse the content of the messages received on port *FReadyIn* and to distinguish between these two situations.

The structure of this system has the advantage that this system is not limited to any specific number of robots. In fact, any number of robots can be included into the system, and there is no need to change robots’ model when the number of robots in the system changes.

6.2.2 Robot Model

As mentioned above, each robot is modeled as a DEVS coupled model. Similar to the *Robot* model described in Chapter 5, the robot model is built based on Brooks’ *Subsumption Architecture* and is shown below:

From the figure we can see that there are two components in the *Robot* model: *Avoid* model and *Convoy* model. Both of them are DEVS atomic models. The *HWInterface* activity is a DEVS activity started by the *Avoid* model. This activity acts as the

sensor/actuator hardware interface. As it basically shares the same role as that of the *activity* described in Chapter 5, readers can refer to Chapter 5 for this *activity*'s function and implementation. In this example, because the *HWInterface activity* belongs to the *Avoid* model, *Convoy* model's moving commands are first sent to the *Avoid* model and then passed to *HWInterface activity* to drive the motors. Similarly, a *moveComplete* message returned from hardware is first sent to *Avoid* model and then passed to *Convoy* model.

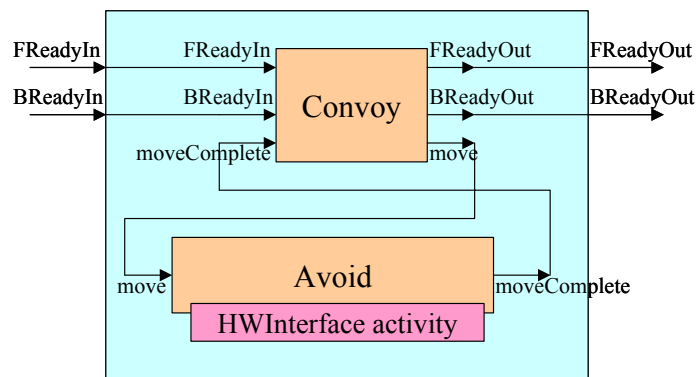
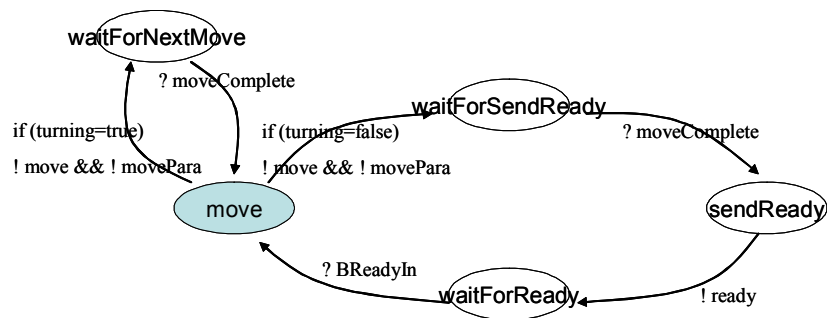


Figure 6.2. Robot Model

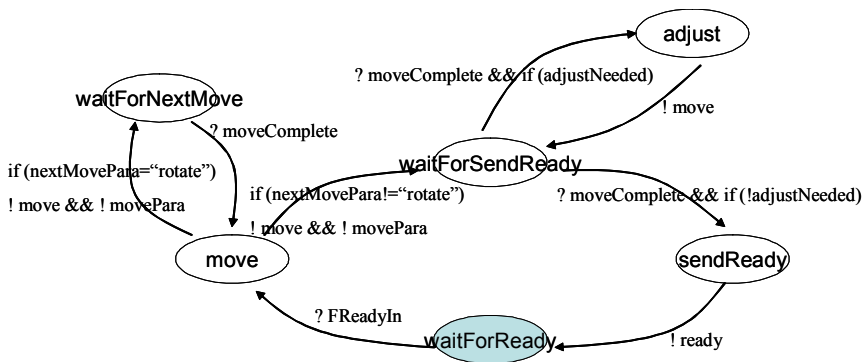
The *Avoid* model controls a robot to move away if the robot collides with anything. To be more specific, in this example, the *Avoid* model continually checks the whiskers to see if any of them has been tripped. If front whiskers are tripped, the *Avoid* model moves the robot backward. If back whiskers are tripped, the *Avoid* model moves the robot forward. We treat the situation that whiskers are tripped as a reactive situation, so the robot responds to this situation with the highest priority. This means the avoid behavior cannot be inhibited by other tasks.

The *Convoy* model is fully responsible to control a robot to convoy in the team. Specifically, it has two roles: to move a robot so that it won't lose its "front" robot, and to synchronize with its neighbor robots so they convoy in a synchronized way. For the first role, the model issues moving commands either based on infrared sensor data or based on the moving parameters received from its "front" robot. The "adjust process" is conducted after every movement. The input port *moveComplete* and output port *move* are used for this role. A message received in the *moveComplete* port means that a movement is completed. This message also contains infrared sensor data which indicate how far the robot is from an obstacle. Based on this data, the robot can decide how to move for the next step. For the second role of the *Convoy* model, input ports *FReadyIn*, *BReadyIn* and output ports *FReadyOut*, *BReadyOut* are used to pass synchronization messages between robots. As mentioned above, for simplicity, port *BReadyOut* is used to pass both synchronization message and moving parameters to a "back" robot.

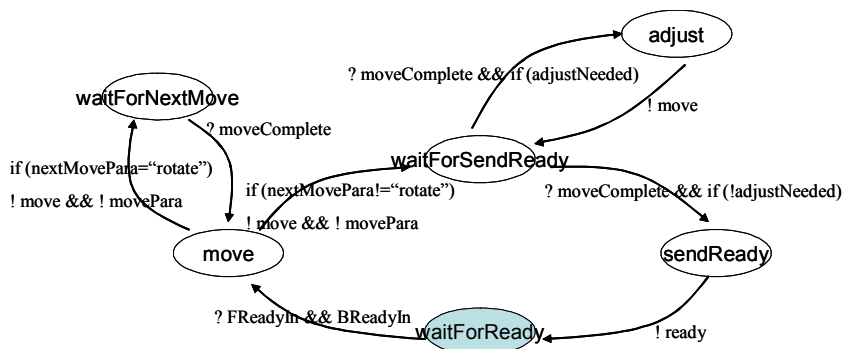
In this system, robots can be categorized into three types: the leader robot that only has a "back" robot, the ender robot that only has a "front" robot, and the intermediate robots that have both "front" and "back" robots. The leader robot conducts movement based on the distance data returned from its infrared sensors. It doesn't need to make adjust movement. The ender and intermediate robots conduct movement based on the moving parameters sent from their "front" robots. If they are not in a desired position/direction after a movement, adjust movements are needed. Figure 6.3 shows the state diagrams of these three types of robots respectively.



(a) State diagram of the leader robot's *Convoy* model



(b) State diagram of the ender robot's *Convoy* model



(c) State diagram of the intermediate robots' *Convoy* models

LEGEND			
	state		initial state
if	condition	?	input
		!	output

Figure 6.3. State charts of robots' *Convoy* models

From the figure we can see that the *Convoy* models of all robots goes through a basic cycle “*move— waitForSendReady— sendReady— waitForReady— move*”. This cycle guarantees that robots move in a synchronized way. Below let us use an intermediate robot as an example to walk through this cycle (as shown by Figure 6.3(c)). Assuming the *Convoy* model of this robot starts with an initial state *waitForReady*. It then goes to the *move* state after both *FReadyIn* and *BReadyIn* messages are received from its “front” and “back” robots (this means both of them are ready so this robot can move). In the *move* state, the model issues a moving command and then goes to the *waitForSendReady* state to wait for the *moveComplete* message returned from the hardware. If the *moveComplete* message is received and there is no need to adjust, the model goes to the *sendReady* state to send ready messages to its “front” and “back” robots, and then retunes to the *waitForReady* state to complete a cycle. Notice the difference among the conditions for a leader, ender, and intermediate robot to go from *waitForReady* state to *move* state. The leader only waits for the *BReadyIn* message; the ender only waits for the *FReadyIn* message; while an intermediate robot waits for both *BReadyIn* and *FReadyIn* messages.

Figure 6.3 shows that normally the *Convoy* model of a robot (leader, ender, or intermediate robots) goes from the *move* state to *waitForSendReady* state. However, if the current movement is a rotation, the model goes to the *waitForNextMove* state. This is because rotation only changes the direction of a robot. The robot still stays at the same location. In this case, if a robot goes to the *waitForSendReady* state instead of the *waitForNextMove* state, it will eventually issue a “ready” message to its “back” robot.

This means the back robot may move forward thus causing a collision because this robot still stays at the same location. To avoid collision, the model goes to the *waitForNextMove* state whenever the current movement is rotation.

Figure 6.3 also shows that an adjust state exist for the ender or intermediate robots. For these robots, whenever they reach the *waitForSendReady* state, they will wait for the *moveComplete* message returned from hardware and check the distance data contained in that message to decide if adjustment is needed or not. As mentioned above, the “adjust process” is used to make sure a robot still follows its “front” robot. If adjustment is needed, the model goes to the *adjust* state and issues an adjustment moving command to ask the robot to adjust its position or direction. This process continues until no further adjustment is needed. To be more specific, after every movement, a robot checks the distance data returned from its front infrared sensor. If the robot sees its “front” robot (the distance data is not infinite) but the distance is too far or too close, the robot makes forward or backward adjusting movement. If the robot cannot see its “front” robot, the commands in the *adjustQ* (shown below) are executed consecutively so that the robot turns left and right (the “scan process”) in order to find its “front” robot. Notice at the end of the queue, the robot returns to its initial direction and then moves forward 20 units. After that the robot executes commands starting from the beginning of the queue again. This “scan, then move, then scan” adjustment process continues until the robot finds its “front” robot. The *adjustQ* is shown below:

```
public void setAdjustQ(){
    adjustQ.add("rotatecc_"+convoySpeed+"_"+6); //turn left 18 degree
    adjustQ.add("rotatecw_"+convoySpeed+"_"+12); //turn right 36 degree
    adjustQ.add("rotatecc_"+convoySpeed+"_"+18); //turn left 54 degree
    adjustQ.add("rotatecw_"+convoySpeed+"_"+24); //turn right 72 degree
```

```

adjustQ.add("rotatecc_" + convoySpeed + "_" + 30); //turn left 90 degree
adjustQ.add("rotatecw_" + convoySpeed + "_" + 36); //turn right 108 degree
adjustQ.add("rotatecc_" + convoySpeed + "_" + 42); //turn left 126 degree
adjustQ.add("rotatecw_" + convoySpeed + "_" + 48); //turn right 144 degree
adjustQ.add("rotatecc_" + convoySpeed + "_" + 54); //turn left 162 degree
adjustQ.add("rotatecw_" + convoySpeed + "_" + 60); //turn right 180 degree
adjustQ.add("rotatecw_" + convoySpeed + "_" + 50); //turn right 150 degree
adjustQ.add("rotatecc_" + convoySpeed + "_" + 40); //turn left 120 degree
adjustQ.add("rotatecc_" + convoySpeed + "_" + 40); //turn left 120 degree to return to initial direction
adjustQ.add("forward_" + convoySpeed + "_" + 20); //forward 20
}

```

6.3 Test the System in a Virtual Testing Environment

To build a virtual testing environment, the *Environment* model and *HWInterface abstractActivity* are developed. The *HWInterface abstractActivity* is used by a robot to imitate *HWInterface activity's* behavior and interface functions. As it is basically the same as the one described in Chapter 5, readers can refer to Chapter 5 for more details about it. The *Environment* model is used to model robots' movement time, and to return sensor data (whisker sensor and infrared sensor) to the control models whenever a robot moves. The model is shown below:

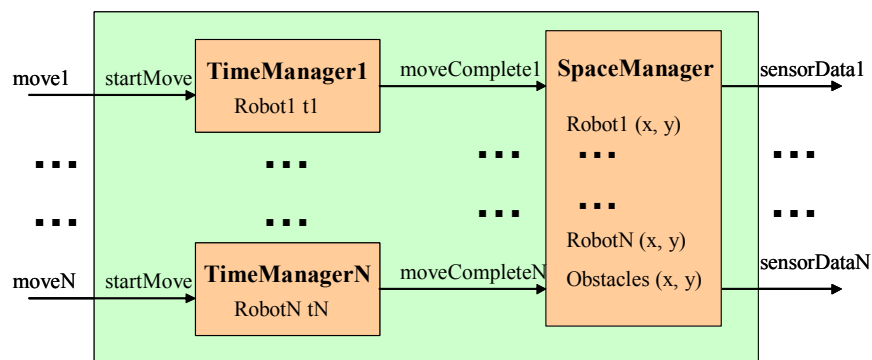


Figure 6.4 Environment Model

As can be seen, the *Environment* model includes the *TimeManager* models and the *SpaceManager* model. For each robot, there is a *TimeManager* corresponding to it. This

TimeManager models the time for a robot to finish a movement. This time is calculated based on the move distance and move speed. To account for variability in the real motion, a random number generator provides a source of additive noise. As each robot has its own *TimeManager*, a group of heterogeneous robots with different moving characteristics can be easily modeled.

The *SpaceManager* models the experimental floor space, including the dimension, shape and location of the field and moving and/or static objects. In this example, the robots are mobile so the *SpaceManager* needs to keep track of their (x,y) positions and moving directions during simulation. It does so whenever receives a *moveComplete* message from *TimeManager*. Such tracking is needed to predict when robots are in the line-of-sight or whisker-based collision relationships and thereby to supply them with the correct sensor data. Note that, when transferred to reality, such tracking is not necessary since robots encounter sensory situations in “situated” fashion. Similarly, to account for variability in the real motion, a random number generator provides a source of additive noise. In this example we have ignored the dynamic processes of a movement as we treat each movement as an atomic action so the positions and directions of robots are updated discretely.

With these models, step-wise simulation methods are applied to test the robot models. These step-wise simulation methods include central simulation (fast-mode and real-time), distributed simulation, hardware-in-the-loop simulation, and real system test. More description about how these simulation methods are conducted can be found in Chapter 3 and Chapter 5. Note that in central simulation, to account for network latency between

distributed robots, we use *addCouplingWithDelay()* to add couplings between robots. This method automatically inserts network delay models on robots' coupling paths. In distributed simulation or real execution, this method is replaced by the regular *addCoupling()* method. By applying the *addCouplingWithDelay()* method, the model shown in Figure 6.1 is transformed into the model as shown in Figure 6.5. In this figure, the models with label *D* are the inserted network delay models.

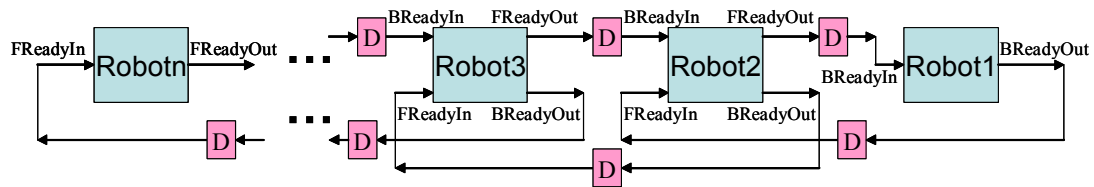
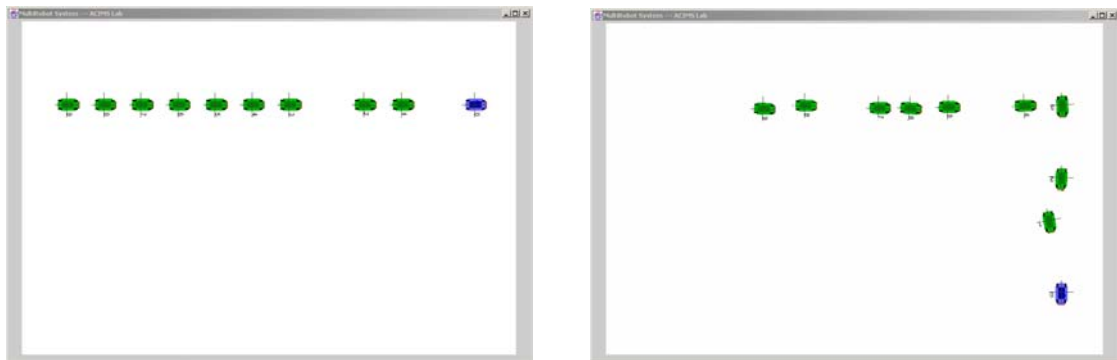


Figure 6.5. Using network delay model in central simulation

To facilitate users to see how robots move in real time, a graphic user interface is provided so that users can easily see how the changes of the control logic may affect a robot's behavior. Figure 6.6 shows two sample pictures captured from the graphic user interface during a central real time simulation. In this example, 10 robots are included.



(a) (b)
Figure 6.6: 10 robots in central real time simulation

After test, the *Robot* models are deployed to the corresponding robots and executed by DEVS real time execution engines. Using the robots as shown in Figure 6.7, we have successfully developed and demonstrated a system with three robots (the fourth robot doesn't work well). We expect this system can be easily scaled up to include more robots.



Figure 6.7: Real robots for demonstration

6.4 A Study of Formation Coherence

One of the advantages of a simulation-based virtual testing environment is that not only the correctness of a robot model can be tested, but also the performance of a system can be evaluated, and different control schemas can be tested and experimented easily. Using this simulation-based testing environment, we have studied the formation coherence of this convoy system.

As mentioned before, due to the variance in real execution, a robot will not move the same distance and direction as it is asked to move. Thus after a movement, a robot will not reach the same position and direction (angle) as the desired position and direction. The difference between a robot's real position and its desired position is affected by the variance of movement in real execution, which is modeled by the noises in simulation.

On the other hand, even though variance exists, the system can still conduct the convoy with some level of formation coherence. This is because an “adjust process” has been implemented that allows robots to adjust their positions/directions based on the feedback from its infrared sensors. Apparently the level of formation coherence is affected by the noises. Thus one interesting problem of this convoy system is to evaluate the formation coherence under the condition of a given set of noise factors.

To study this problem, we added noises in robots’ movement during simulation. Specifically, we define the distance noise factor as the ratio of the maximum distance variance divided by the desired distance a robot is supposed to move; and the angle noise factor as the maximum angle variance for each movement. For example, a distance noise factor being 0.2 means there will be maximum plus minus $0.1 * MovingDist$ variance if a robot moves *MovingDist* unit; an angle noise factor being 10 means a robot will have the maximum plus minus 5 degrees variance from its desired direction. These noise factors allow us to run simulations in a realistic way. Figure 6.8 shows four robots’ moving trails in an example system with distance noise factor set to 0.16 and angle noise factor set to 4. In this example, robots were put in an empty rectangle field surrounded by walls. There is no other object within this field. For analytic purpose, we set *robot0*’s noise factor and angle factor to 0, so *robot0* (the leader) always moves in a perfect way. This allows us to easily calculate other robots’ desired positions for each moving step.

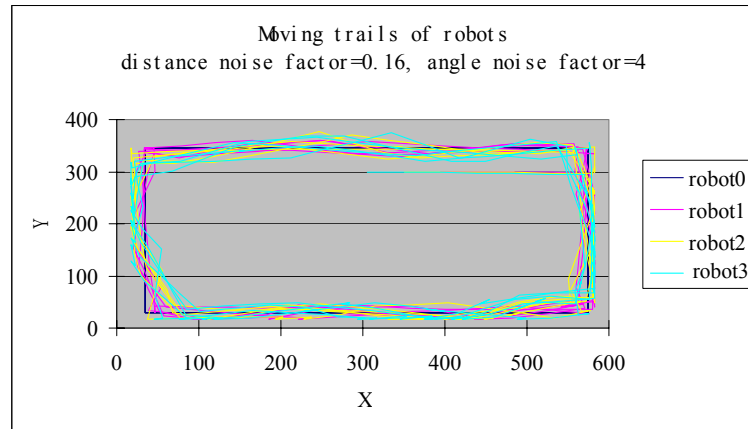


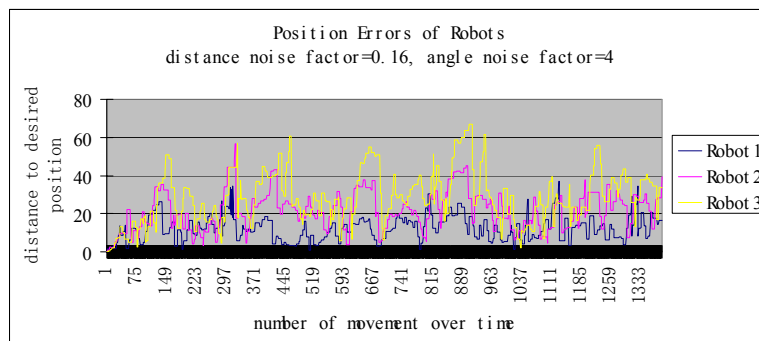
Figure 6.8: Moving trails of robots during a simulation

We define the formation coherence of this convoy system as the fidelity of robots' real positions to their desired positions (the positions where robots are supposed to be in a perfect environment). For this purpose, we define error $e_i(t)$ as the distance error (between real position and desired position) of robot i at time t . The total error $E(t)$ is the sum of $e_i(t)$ of all robots. This total error $E(t)$ is an indicator for the convoy system's formation coherence (a precise mathematic formula may be defined such as defining formation coherence $C(t) = 1 - E(t)/K$, where K is a constant reflecting the designer's expectation). Figure 6.9 shows each robot's error and their total error for the example system in Figure 6.8.

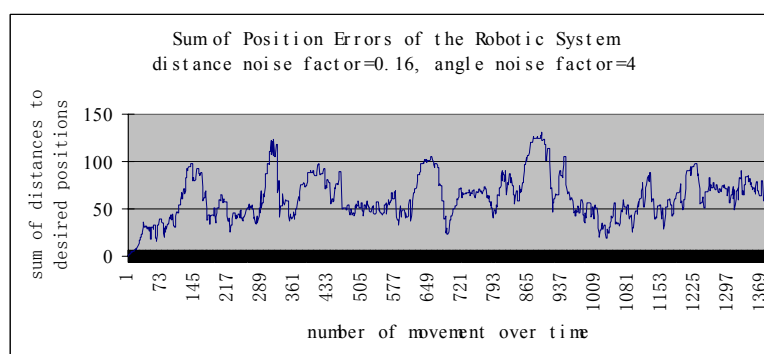
Our study shows that under the current control schema (passing moving parameters backward, and applying "adjust process" after every movement), the more "behind" a robot's position is, the more distance error that robot will have. For example, Figure 6.9(a) clearly shows that *robot3* has more distance error than *robot1*. On the other hand, the study also shows that, for a system with limited number of robots and small noise

factors, the formation of this convoy system is “coherent” in the sense that the total distance error of robots is always kept within a boundary. This is also showed by Figure 6.9(b) to some extent.

To further study how different control schemas may affect this convoy system’s formation coherence, we designed several other schemas and applied them to the system. For example, we noticed that a robot would sometimes miss its direct “front” robot (because the robot is not in its line-of-sight) and “see” its “front” robot’s “front” robot. In that case, under the first control schema the robot will move aggressively ahead trying to catch that robot. This causes disorder situations between robots. To solve this problem, we revised the control logic of the first schema and designed a new schema. In this new schema, if a robot “sees” a “front” robot far away, it will first scan around to check if there is other robot nearby. If there is, that robot should be the “front” robot. This schema had been tested in the virtual testing environment and showed improvement in system’s formation coherence. Other schemas have also been tested and experimented. For example, we tested a schema in which a robot will pass not only its regular moving parameters but also its adjustment moving parameters to its “back” robots. This allows the “back” robot to predict its “front” robot’s position more precisely and thus being able to make according pre-adjustment.



(a)



(b)

Figure 6.9: Position errors of the robotic convoy system

Using simulations to study the problem of formation coherence also gives us the insight that from the system design point of view, the formation coherence obtained via simulation can actually act as a criterion to help designers to design or to choose the sensor/actuators hardware of the system. For example, by running multiple simulations with different configurations, we detected that the system's formation coherence is not sensitive to robots' infrared sensor data (due to the "adjust process" after every movement). Even we added large noises into the infrared sensor data, the system can still conduct a "coherent" convoy (this is true for a system with a small number of robots). This feature that is obtained via simulation-based study implies that there is no need to

equip robots with advanced (thus costly) infrared sensors, because robots will conduct “coherent” convoy even with “bad” infrared sensor data.

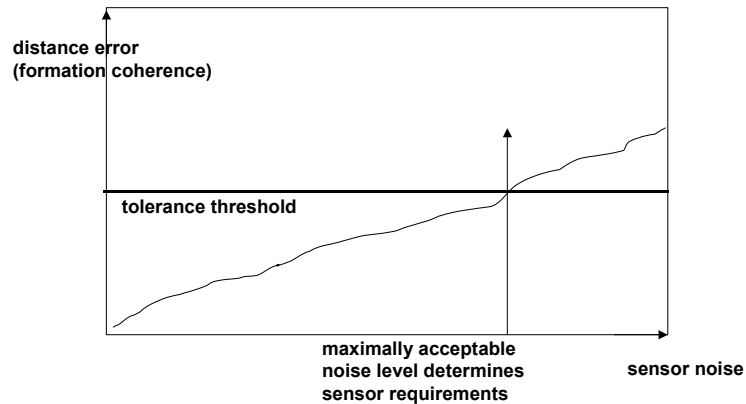


Figure 6.10: Formation coherence as criteria for sensor capabilities obtained via simulation

Following this idea, we can further use simulation to study the relationship between formation coherence and sensor’s noises, and then use formation coherence as a criteria to determine sensors’ requirements. This is shown in Figure 6.10. As shown in the figure, the error of formation coherence increases when the noise factor of sensors increases. As the error of formation coherence reaches the tolerance threshold (which is a design criteria provided by the users), this is the point where the maximum sensor noise level can be accepted. Thus with simulation-based study, we can use formation coherence of this convoy system as a criterion to determine sensors’ requirements. Similar study can also be conducted to determine actuators or other components of the system to be designed.

CHAPTER 7

A HIGH PERFORMANCE SIMULATION ENGINE FOR LARGE-SCALE SYSTEMS

7.1 Simulation of large-scale systems

Today's real-time embedded systems are more and more networked together to form large-scale networked systems. For example, the Warfighter Information Network (WIN) in DOD would include thousands or even tens of thousands communication devices to provides command, control, communications, computers, intelligence, surveillance and reconnaissance (C4ISR) support capabilities [FAS]. For this kind of large-scale systems, the multitude of proposed solutions at each software and hardware layer has led to explosions in possible design choices. The size of these systems makes experimentation and measurement prior to deployment impossible, yet the risk of deploying the new technologies in critical situations require assurance that they will work. To handle the scalability of these systems and to provide insights to help the designers make appropriate design choices, simulation technologies are frequently applied.

Simulation-based study of large-scale systems requires high performance simulation environments. Usually parallel or distributed simulation technologies are applied to achieve this [Man03], [Lok99] [Par03]. In the meantime, there is also considerable research work to enhance the implementation of simulation environments to achieve performance improvement. One of such work is recently presented in [Ste03] that is based on the Joint MEASURE simulation environment developed at Lockheed Martin.

This chapter presents our work of a high performance simulation engine for large-scale cellular models. The cellular automata paradigm defines a grid of cells using discrete variables for time, space and system states [Wol86]. The cells are updated according with a local rule function that uses a finite set of nearby cells (called the neighborhood of the cell). Cellular DEVS models have been developed to model and simulate various phenomena such as fire spreading [Muz02, Bit03], traffic control [Dav00], etc.

This new simulation engine that we developed improves simulation performance for large-scale cellular models from two sources that are based on the qualities of cellular space models. First, based on the observation that usually only a small portion of cells in a cell-space model is active (performing state changing) at any given time, this new simulation engine considers only those active cells during simulation. This enhances simulation performance compared with simulations that are based on cellular automata in which all cells perform computations and message exchange at every time step. This is also the approach taken by [Ame01] and [Muz02]. Second, in a cell space model, the active cells are typically locally clustered. This is because cells are coupled to their neighbors so the state change of one cell will first directly affect its neighbors. Based on this observation, a new data structure that retains cells' spatial information and thus takes advantage of the localized activities of cellular space models is developed to increase simulation performance. With this data structure, search of the active cells can be arbitrarily faster in cellular space models where the number of cells increases but the number of active cells remains the same.

The following chapter describes this new simulation engine and the data structure it employs. To set the stage, we first review the standard DEVS simulation protocol as implemented by the *coordinator* in DEVSJAVA [DEVJ]. Then we propose an improved simulation engine that is based on the standard *coordinator*. With this background, we proceed to describe the new simulation engine, *oneDCoord* as implemented in DEVSJAVA, and its *minSelTree* data structure. After that, we analyze the performance of these simulation engines. Finally, two examples are presented to demonstrate the performance improvement of the new simulation engine as compared to the standard *coordinator*.

It is worthy to mention that although this simulation engine is developed for cellular space models, it can also be applied to other large-scale simulations such as the simulation of swarm intelligence that could include thousands of mini robots; or the simulation of distributed networks that contain a large number of network nodes.

7.2 The Standard DEVS Simulation Protocol

In a DEVS-based simulation environment such as DEVSJAVA, a coordinator is assigned to a coupled model and simulators are assigned to each component. Figure 7.1 shows the simulation of a coupled model with three components.

The simulation of DEVS models moves forward cyclically based on the time of next event, denoted by tN , which is updated by component models' state transition functions. In the standard DEVS simulation protocol, the coordinator is responsible for stepping simulators through the cycle of activities as shown in Figure 7.1. Specifically in the

DEVSJAVA simulation environment, the simulation protocol in each simulation cycle looks similar as below:

```

simulators.AskAll("nextTN")
tN = compareAndFindTN();
simulators.tellAll("computeOut",tN)
simulators.tellAll("sendOut")
simulators.tellAll("ApplyDelt",tN)

```

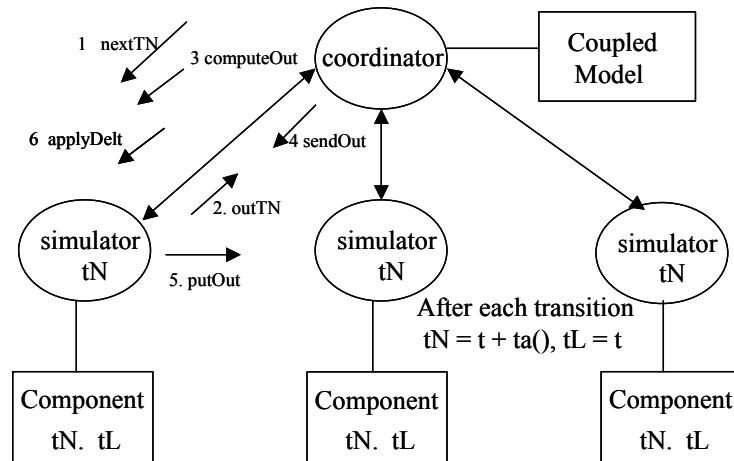


Figure 7.1: The Standard DEVS Simulation Protocol

A detailed description of how the coordinator and simulators step through a simulation cycle is given below:

1. Coordinator sends *nextTN* to request *tN* from each of the simulators.
2. All the simulators reply with their *tNs* in the *outTN* message to the coordinator. The coordinator compares these *tNs* and finds the minimum.
3. Coordinator sends to each simulator a *computeOut* message containing the global *tN* (the minimum of the *tNs*). Each simulator checks if it is imminent (its *tN* = global *tN*) and if so, computes the output message of its model. Otherwise an empty message is generated.

4. Coordinator sends to each simulator a *sendOut* message.
5. Based on the coupling specification, each simulator responds by putting its output message (if it is not empty) to the destination simulators.
6. Coordinator sends to each simulator an *applyDelt* message containing the global tN . Each simulator reacts to the *applyDelt* message as below:
 - If it is imminent and its input message is empty, then it invokes its model's internal transition function
 - If it is imminent and its input message is not empty, it invokes its model's confluence transition function
 - If is not imminent and its input message is not empty, it invokes its model's external transition function
 - If is not imminent and its input message is empty then nothing happens.

This simulation protocol follows closely with the semantic of DEVS models. Thus it is easy to be understood and implemented. However, it tends to result in slow simulation speed for models that have a large number of components. This is because in every simulation cycle, all the simulators, no matter they are imminent or not, have to go through the simulation steps as described above. To overcome this shortcoming, an improved simulation engine can be proposed. This simulation engine uses a *Heap* data structure to sort and find imminents and then only those imminents are asked to go through the simulation cycle.

7.3 A Proposed Improved Simulation Engine

This proposed simulation engine (not implemented by DEVJSJAVA) implements a heap to keep track of the smallest tNs of its component simulators. During simulation, each active simulator removes and inserts its tN in the heap. So the smallest tN and imminents can be found from the root of the heap.

Using a heap to keep track of the tNs and imminents, the simulation protocol of this proposed simulation engine in each simulation cycle is shown below:

```

tN = Heap.getMin()
imminents = Heap.getImms()
imminents.tellAll("computeOut",tN)
imminents.tellAll("sendOut")
imminents = imminents.addAll(influencees)
imminents.tellAll("ApplyDelt",tN)
imminents.tellAll("updateHeap")

```

In every simulation cycle, the simulation engine first gets the smallest tN and the imminents from the heap. With the smallest tN and imminents in hand, the simulation engine then sends out the *computeOut* and *sendOut* messages to imminents. The *sendOut* message will trigger imminents to put their output messages to their destination simulators, which are called *influencees*. The *influencees*, like the *imminents*, need to execute their state transition functions. Thus before executing *imminents.tellAll("ApplyDelt",tN)*, the coordinator adds those influencees into imminents by executing *imminents = imminents.addAll(influencees)*. At the end of the iteration, the coordinator asks all imminents to update their newest tNs in the heap to prepare for the next simulation iteration.

Generally speaking, the update process where each active simulator removes and inserts its tN in the heap has computation complexity $O(n*\log_2N)$ (n is the number of imminents in the simulation cycle; N is the total number of cells). For large-scale cellular models with small number of imminents ($n \ll N$), this proposed simulation engine has computation complexity at the magnitude of \log_2N , thus resulting in considerable performance improvement.

Based on this proposed simulation engine, a new simulation engine, the *oneDCoord* in DEVSJAVA environment, is developed. This new simulation engine not only keeps track of the imminents and asks only the imminents to go through a simulation cycle, but also implements a the *minSelTree* data structure to allow efficient search of the imminents.

7.4 The New Simulation Engine and Its Data Structure

7.4.1 The Simulation Protocol

The simulation protocol of this new simulation engine is similar to the proposed simulation engine as described above. The only difference is that a new data structure *minSelTree* is developed to replaces the heap for keeping track of tNs and imminents.

Below is the simulation protocol of this simulation engine.

```

tN = minSelTree.getMin()
imminents = minSelTree.getImms()
imminents.tellAll("computeOut",tN)
imminents.tellAll("sendOut")
imminents = imminents.addAll(influencees)
imminents.tellAll("ApplyDelt",tN)
imminents.tellAll("sendTNUp")

```

As can be seen, in every simulation cycle, the new simulation engine first gets the smallest tN and *imminents* by executing *minSelTree.getMin()* and *minSelTree.getImms()* respectively. Then similar to the description above, only the *imminents* (and *influences*) go through the simulation cycle. The last step in the simulation cycle is to ask all *imminents* to send their newest tNs to the *minSelTree* so the information kept there is updated timely. This is to prepare for the next simulation iteration.

7.4.2 The *minSelTree* Data Structure

The *minSelTree* data structure is the essential part of this new simulation engine to keep track of tNs and *imminents*. It is a complete tree, which means each leaf node of this tree data structure has the same “distance” from the root. The *minSelTree* is constructed in such a way that for each cell in the model, there is a leaf node of *minSelTree* corresponding to it. To retain a cell’s spatial information in *minSelTree*, the cell’s ID is used as a reference to assign a leaf node to the cell. As adjacent cells have adjacent IDs, their corresponding *minSelTree* nodes will sit adjacently in the *minSelTree* too. The *minSelTree* is set up during initialization of the simulation based on the total number of cells and the base of the tree, which means the number of children of each internal node and is provided by the user. The formula below shows the relationship among the number of cells N , the base b and the height h (distance from leaf to the root) of the *minSelTree*.

$$h = \text{ceiling}(\log_b N)$$

Figure 7.2 shows the relationship among the models, simulators, and the *minSelTree* data structure. Here we assume the model to be simulated is a one-dimension cellular

space model with n cells. These cells have IDs from 1 to n based on their positions in the cellular space. From Figure 7.2 we can see that for each cell, there is a simulator *oneDSim* assigned to it; for each simulator, there is a leaf node of *minSelTree* corresponding to it. Thus a cell, its simulator, and the *minSelTree* leaf node form one to one relationship to each other. Among them, the simulator has access to both the cell and the leaf node. Figure 7.2 also shows that the coordinator *oneDCoord* has access to the root node of *minSelTree*.

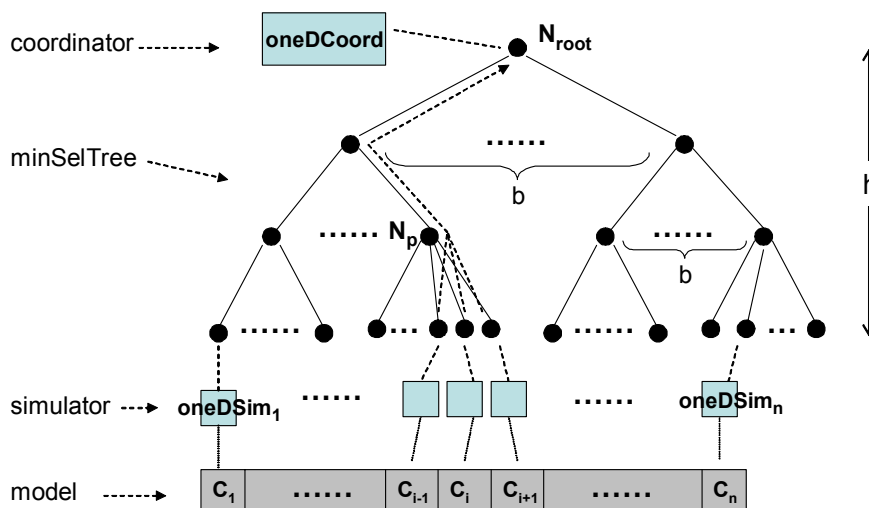


Figure 7.2. Model, simulator, and the *minSelTree*

During simulation, a simulator *oneDSim* is responsible to drive the simulation of its cell model and to update the new tN to the corresponding leaf node of *minSelTree*. The leaf node then sends this tN up to its parent node, which compares this tN with the tNs of other children and selects the smallest ones to send up. This “send up” process continues until the root node is reached. Note that the information that is sent up includes not only the smallest tN , but also the references of those simulators which hold that tN . Because each node selects the smallest tN and the imminent simulators to send up, after this

recursive “send up” process ends, the root node has the global minimum of tN s and the references for all the imminent simulators. The coordinator *oneDCoord* can then access the root node of *minSelTree* and easily gets that information.

To make this process feasible, each node of the *minSelTree* keeps track of its children’s tN and imminent simulators (A leaf node keeps track of its own tN and itself as the imminent simulator). Specifically, each node of *minSelTree* has a variable *minEnvironment*, which is a table storing information in the following format: (*childName*, *Pair(imminents, tN)*). The *childName* is the name of a child node. The tN is the child node’s tN and the *imminents* is a set containing the references for all the simulators holding that tN . The *imminents* and tN are encapsulated into a *Pair*. With this information stored in *minEnvironment*, a node can find its tN (the smallest tN among its children) and the references for the simulators holding that tN by executing the following *whichMin()* method:

```
public Pair whichMin() { // find the imminents and tN
    double timeGranule = .000001;
    ensembleSet imminents = new ensembleSet();
    double min = POSITIVE_INFINITY;
    while(minEnvironment.hasNext()) {
        Pair p = (Pair) minEnvironment.next(); //name, Pair
        Pair pp = (Pair) p.getValue(); //imms, tN
        double tN = pp.getValue(); // get tN
        if (Math.abs(tN - min) < timeGranule) imminents.addAll((ensembleSet) pp.getKey());
        else if (tN < min) {
            imminents = new ensembleSet();
            min = tN;
            imminents.addAll((ensembleSet) pp.getKey());
        }
    }
    return new Pair(imminents, min);
}
```

This method compares all tNs stored in *minEnvironment* and selects the smallest one. In the meantime, it adds the simulators that hold the smallest tN into the *imminents* set. The method returns a new pair that contains the *imminents* and the smallest tN . Notice that a variable *timeGranule* is used to specify the smallest time unit in simulation. Events that happen inside the same *timeGranule* are considered happened at the same time.

The information stored in *minSelTree* needs to be updated continuously when simulation proceeds. This is accomplished by executing *imminents.tellAll("sendTNUp")* at the end of each simulation iteration. This step asks all imminent simulators to update their new tNs to their corresponding leaf nodes. As mentioned before, the update of a new tN triggers a recursive “*send up*” process until the root node is reached. During this process, each node finds the smallest tN among its children (by executing the *whichMin()* method) and sends that information up.

In a cellular space model, cells are coupled to their neighbors. Thus it's typical that a cell and its neighbors change their states at the same time. For example, in Figure 7.2, cell c_{i-1} , c_i , and c_{i+1} may all change their states at the same time and have new tNs . If the leaf nodes for these cells send their tNs up independently, each of them will trigger a “*send up*” path to the root node N_{root} . Apparently this is inefficient as these nodes actually share the same parent N_p (because the cells are adjacent in the cellular space model). Thus the “*send up*” paths behind node N_p can be combined into one path. By combining several “*send up*” paths into one path, the *minSelTree* nodes on that path only need to execute the *whichMin()* once instead of multiple times. Notice that this improvement actually shows how the new coordinator takes advantage of the fact that activities of

cellular models usually happen locally. Because the spatial information of cells is retained in *minSelTree*, a cell and its neighbors' nodes will share the same parent in *minSelTree*. Thus their “send-up” paths can be bundled together, which results in performance improvement. The code listed below shows how a *minSelTree* node implements the *sendUp()* method.

```
public void sendUp (String nm,Pair p){
    minEnvironment.setPair(nm,p);
    receivedImmi++;
    if(receivedImmi== expectedImmi){
        receivedImmi=0; //reset the value for the next cycle
        expectedImmi =0; //reset the value for the next cycle
        whichMin = whichMin();
        if (!root) parent.sendUp(myName,whichMin);
    }
}
```

As can be seen, this *sendUp()* process is a recursive process. It continues until the root node is reached. When receiving an update from a child node, the method first updates the node's *minEnvironment* variable. Then it increases *receivedImmi* and checks if *receivedImmi* equals *expectedImmi*. The two variables *expectedImmi* and *receivedImmi* are used to guarantee that only one “send up” path is invoked. Only when *receivedImmi* equals *expectedImmi*, which means the node has got all expected update from its children, does the method execute the *whichMin()* to find the smallest *tN* and then calls *sendUp()* to send this information up. The variable *expectedImmi*, meaning how many update a node expects from its children, is set by the *informChange()* method as shown below.

```
public void informChange() {
    if(expectedImmi ==0&&!root)
        parent.informChange(); // inform change only once
    expectedImmi ++;
}
```

Whenever a cell changes its state and has a new tN , its simulator calls the corresponding leaf node's *informChange()* method, which will increase the *expectedImmi* of that node. This method also makes sure that the parent's *informChange()* method will only be called once. Using the system shown in Figure 7.2 as an example, if cell c_{i-1} , c_i , and c_{i+1} all change their states, the *expectedImmi* of node N_p is 3; the *expectedImmi* of node N_p 's parent node is 1 (assuming there are no other cells changing their states).

7.4.3 Building *minSelTree* for two-dimension cellular space models

The model given in Figure 7.2 is a one-dimension cellular space model. In a two-dimension cellular space model, cells have neighbors not only along the x dimension but also along the y dimension as shown in Figure 7.3.

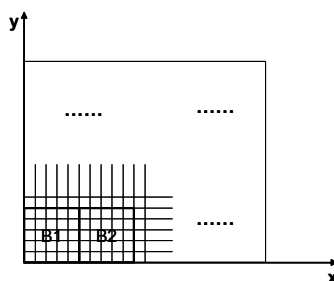


Figure 7.3. A two-dimension cellular space model

To construct *minSelTree* for a two-dimension cellular space model, one way is to assign consecutive IDs to all the cells row by row, thus treating the two-dimension cellular space model as a one-dimension model. However, this approach results in the situation that cells are clustered (having nearby IDs) in only one dimension. To take good advantage of localized activities of cellular models, it is desirable that cells are clustered

in two dimensions. Thus in our implementation, we assign cells' IDs based on blocks as shown in Figure 7.3. All cells inside one block belong to one parent in the *minSelTree*. The resulting *minSelTree* is a little bit different from the one discussed before. In the tree for one-dimension cellular space models, each internal node of the tree has b children (b is the base of the tree). For the new *minSelTree*, it is the same situation except that the bottom nodes have $BXs*BYs$ children (BXs and BYs denote the size of a block). Notice that the previous situation is actually a special case of this new one with $BXs=base$ and $BYs=1$.

7.5 Performance Analysis

Compared to the original coordinator with the basic simulation protocol, there are two sources of speed up of the new simulation engines. One is the keeping track of imminents so only the imminent cells will be considered in every simulation cycle. The other is the efficient smallest tN search. Below we analyze the performance of these simulation engines by considering two extreme cases: only one cell is imminent in a simulation iteration, and all cells are imminent in a simulation iteration. For simplicity, we only consider the computation complexity of finding the smallest tN . Let us assume there are N cells in the cellular space model.

For the coordinator with the basic simulation protocol, all simulators are asked to send their tNs to the coordinator. Then the coordinator compares and selects the minimum among these N tNs . Thus the computation complexity is $O(N)$, which is independent of the number of imminent cells.

For the coordinator with heap implementation, each imminent simulator removes and inserts its tN in the heap. Then the smallest tN is found by getting the value of the top of the heap. If only one cell is imminent, only one simulator needs to update its tN in the heap, which takes computation complexity $O(\log_2 N)$. If all cells are imminent, all simulators need to remove and insert their tNs in the heap, resulting in computation complexity $O(N * \log_2 N)$.

For the new coordinator with *minSelTree*, the height of the tree $h = \text{ceil}(\log_b N)$ (b is the base of this tree). If only one cell is imminent, only one simulator will update its new tN on the leaf node of *minSelTree*. Thus only one “send-up” path, which has h nodes in the path, will be generated. Along this path, each node executes *whichMin()* method to compare and select the minimum tN among b children. This results in computation complexity $O(h * b) = O(b * \log_b N)$. If all cells are imminent, all simulators will update their new tNs on their leaf nodes of *minSelTree*. As a result, all nodes in the *minSelTree* will be involved in the “send up” process. On the other hand, even a node’s *sendUp()* method will be called multiple times, the *whichMin()* method will only be executed once. Assume the total number of nodes of *minSelTree* is T (exclude the leaves of the tree), then the computation complexity is $O(T * b)$. For a complete tree with N leaves, $T = (N - 1) / (b - 1)$. This results in computation complexity $O(T * b) = O((N - 1) * b / (b - 1)) = O(N)$.

The above analysis shows that when all simulators are imminent, both the new simulation engine and the *coordinator* have computation complexity $O(N)$, which is better than $O(N * \log_2 N)$, the complexity of the proposed simulation engine. However, when only one simulator is imminent, the proposed simulation engine has computation

complexity $O(\log_2 N)$, which is close to the performance of the new simulation engine $O(\log_b N)$ and better than the performance of *coordinator*. To further compare the new simulation engine with *minSelTree* and the proposed simulation engine with heap implementation, let's consider another example. *E.g.*, let $N = b*b$ for a two-level *minSelTree* and let there be b imminent cells within one block (this means the leaf nodes of these cells share the same parent). Considering only the computation required by comparisons, the new coordinator takes $2b$. And the heap implementation takes $b \log_2 N = b \log_2 (b*b) = 2b \log_2 b$, while the old coordinator takes $b*b$. The utilization of localized activity is clear here: $2b$ vs $2b \log_2 b$. The extra $\log_2 b$ part is due to the full reordering that the heap implementation does for every replacement.

7.6 Examples and Test Data

This section describes the simulation of two examples to compare the simulation performance of the new simulation engine and the standard *coordinator*. The first example is a one-dimension cell-space model that models the phenomenon of diffusion. The second example is two-dimension cell-space model that models the phenomenon of fire spreading. For each example, a brief description of the model is given and then the simulation data of the two simulators is provided.

7.6.1 Simulation of a Diffusion Model

Diffusion is a common phenomenon that has been studied using simulation methods. In this example (the *diffuse2ndOrdCellSpace* model in DEVSJAVA environment), a heat

diffusion problem is modeled as a one-dimensional cell-space model with each cell coupled to its left and right neighbors (except the boundary cells). Each cell holds its current temperature and will gradually change to the desired temperature, which is defined as the average of the left and right cells' temperatures. The speed of temperature change is determined by the difference between a cell's desired temperature and its current one. For each cell, a quantum is provided so a cell will update itself and inform its neighbors only when the change of temperature reaches the quantum size. With this approach, each cell can calculate its time advance for the next update based on the quantum and the speed of temperature change.

Table 7.1: Comparison of the two simulation engines when simulating a one-dimensional diffusion model

number of cells	50	100	500	1000
coordinator	29.592	54.448	272.712	542.721
oneDCoord (base = 6)	7.221	7.24	8.222	8.432
speedup (coordinator time/oneDCoord time)	4.098047	7.520442	33.16857	64.36444

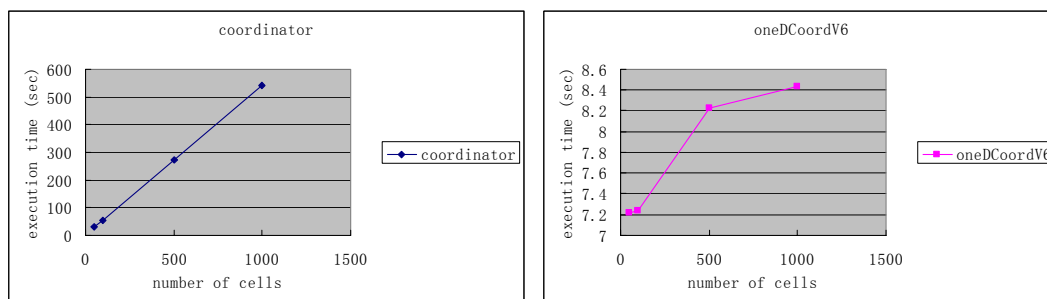


Figure 7.4: Simulation time of coordinator and oneDCoord

Table 7.1 shows the time (in seconds) of simulating this diffusion model with different number of cells using the standard coordinator and the new simulation engine

respectively. For each simulation running, the quantum is defined as 0.1, and the number of simulation iterations is 10000. The simulations were run on a laptop with Intel Pentium IV 1.7GHZ processor, 256M memory, and Windows 2000 OS.

Table 7.1 clearly shows that the simulation speed of the new simulation engine is better than that of the *coordinator*. It also shows that as the number of cells increase, the simulation time of *coordinator* linearly increases too. However, the simulation time of *oneDCoord* only increases slightly (as shown in Figure 7.4). This is because in this example, the increase of cells does not affect the number of imminents in every simulation cycle. As a matter of fact, the number of imminents remains 1 or 2 in this example as the total number of cells increases.

7.6.2 Simulation of a Fire Spreading Model

This example describes a dynamic forest fire spread model, which is based on the work of [Bit03]. In this model, a forest is modeled as a two-dimensional cell-space composed of individual forest cells coupled together according to their relative physical geometric locations. Each cell is modeled in the same way as that of [Vas93], [Ame01]. Specifically, each cell has the following six states: *unburned*, *burning*, *burned*, *unburned-wet*, *burning-wet*, and *burned-wet*. Conditions and rules are defined to govern the state transition of a cell. In the two-dimensional cell space model, each cell has eight neighbor cells *N*, *NE*, *E*, *SE*, *S*, *SW*, *W*, and *NW* except the boundary cells. Accordingly, for each cell, fixed fire spreading directions along these eight directions are defined. Fire spread in each cell is modeled using Rothermel's [Rot72] stationary model. During simulation, the

behavior of a burning cell can be influenced by external inputs from neighboring cells as well as changes in weather conditions. In addition, uncertainty is incorporated in the model by allowing certain critical parameters to be sampled from arbitrary probability distributions during the simulation run. A detailed description of this model can be found in [Bit03].

Table 7.2 shows the time (in seconds) of simulating the fire spread model with different number of cells using the standard coordinator and the new simulation engine respectively. For each simulation running, the number of simulation iterations is 7200. The simulations were run on a laptop with Intel Pentium IV 1.7GHZ processor, 256M memory, and Windows 2000 OS. Again, Table 2 clearly shows the advantage of the new simulation engine (with average speedup between 18 and 20 as shown in Table 7.2).

Table 7.2: Comparison of the two simulation engines when simulating a two-dimensional fire spread model

Number of cells	30x30 (900)	34x34 (1156)	40x40 (1600)
coordinator	356.293	471.258	681.951
oneDCoord (base = 6)	19.388	23.924	36.002
speedup (coordinator time/oneDCoord time)	18.37699	19.69813	18.94203

CHAPTER 8

CONCLUSION AND FUTURE WORKS

8.1 Conclusions

Powered by the rapid advance of computer, network, and sensor/actuator technologies, real-time embedded systems are more and more networked together and have increasingly complicated behavior and structures. The combination of temporal requirements, concurrent environmental entities, and high reliability requirements, together with distributed processing make the software to control these systems extremely hard to design and difficult to verify. As a result, the software development of these systems, which mainly focused on low-level coding and programming in the past, is being rapidly shifted to involve high-level modeling techniques and software design and verification methods.

In this dissertation we have developed a simulation-based software development methodology to manage the complexity of developing distributed real-time software. This methodology, based on the discrete event system specification (DEVS), overcomes the “incoherence problem” between the design and implementation stages by emphasizing “model continuity” through the development process. Specifically, the methodology allows the same designed control models to be tested and analyzed by simulation methods and then easily deployed to the distributed target system for execution. To achieve this, we clearly separate a system’s sensor/actuators interfaces

from its control model, which is the main design and test interest. During the process, virtual sensors/actuators are developed for simulation-based test, and then replaced by real sensors/actuators during real system execution. By restricting virtual sensors/actuators to sharing the same interface functions with their corresponding real sensors/actuators, the continuity of the control models is supported from simulation-based design to real system execution.

The methodology employs simulation-based methods to test the software under development. Specifically, to improve the traditional software testing process where real-time embedded software needs to be hooked up with real sensor/actuators and placed in a physical environment for system-level test and analysis, we developed a virtual testing environment that allows software to be effectively simulated and tested in a virtual environment, using virtual sensor/actuators. Within this environment, we developed a stepwise simulation-based test process so that different aspects of a real-time software system can be tested and analyzed incrementally.

One important aspect of developing real-time software is to capture a system's behavior, structure, and timeliness in an effective way. Our research shows that DEVS models, which are based on formal systems theory, provide a natural and effective way to model distributed real-time systems' structure, dynamic behavior, and timeliness. For real-time systems, *activity* has been developed to allow models to interact with an external environment. Furthermore, variable structure modeling capability is developed and implemented so that dynamic reconfiguration of real-time systems can be modeled naturally. Within this DEVS modeling, simulation, and real-time execution framework,

models can be developed, simulated/tested by simulation methods, and then executed in a distributed environment.

Based on the proposed methodology, we have developed a design and test environment for distributed autonomous robotic systems, which form an interesting class of real-time systems. In particular, our work on “robot-in-the-loop” simulation allows real and virtual subsystems to work together for a meaningful system-wide test. For example, when developing a robotic system that includes hundreds of mini mobile robots, one or several real robots can be tested and experimented with other hundreds of virtual robots that are simulated on computers. With the help of this environment, we have successfully developed and investigated several distributed robotic systems. One of them is a “dynamic team formation” system in which mobile robots search for each other, and then form a team dynamically through self-organization. Another system is a scalable robot convoy system in which robots convoy and maintain a line formation in a coordinated way.

Another issue that arises when applying simulation methods to study and test large-scale real-time systems is the performance of the underline simulation engine. By studying this problem in the context of cellular space models, we developed a new simulation engine. Compared to the standard *coordinator*, this simulation engine speedups the simulation from two sources. First, it only considers the active cells during simulation. This is based on the observation that usually only a small number of cells are active (performing state changing) at any given time, even though the total number of cells may be very large. Second, it implements a data structure that allows efficient

search of the active cells which can be arbitrarily faster in cellular space models where the number of cells increases but the number of active cells remains the same. Performance analysis and two examples are provided in the dissertation to demonstrate the efficiency of the new simulation engine.

8.2 Future Work

This research has established a framework for distributed real-time software development. In the meantime, it also opens up several future research directions. Some of these directions are listed below.

First, while the current research has mainly focused on the continuity from simulation-based design to real software execution, it does not specifically address the problem of how to start from a system's requirements in a methodological way and then go to the design stage, where DEVS models are developed. This can be enhanced by integrating techniques from object-oriented development such as UML use case analysis, sequence diagrams, *etc.* A more interesting work would be to integrate these techniques, together with systems theory concepts that DEVS supports, into the model continuity process that we have developed. The importance of having a systems theory-based design process has been documented in [Pau03], [Pre01]. For example, [Pre01] presented a system-centered use cases-driven design approach. Integrating these systems concepts such as system specification at different abstraction levels, hierarchical decomposition, system-centered use cases, *etc.* into software design holds the potential to reach more systematic and effective design approaches for complex software-intensive systems.

Another future research that will benefit system test is to conduct further research on the simulation-based virtual testing environment. For example, our current research has adopted hardware-in-the-loop simulation as a test step. While hardware-in-the-loop simulation focuses on including a piece of hardware into a testing loop, this idea can be further extended to the system level to form system-in-the-loop simulation (such as robot-in-the-loop simulation), which allow a real system to interact with a virtual environment that is simulated by computers. In fact, our work on the virtual testing environment and stepwise simulation-based methods has obscured the boundary between a real system and the virtual environment. This kind of seamless integration of a real system an a virtual environment will find more and more future applications as simulation technologies advance.

Following the direction of system test, more research can be conducted for simulation-based test. We know that for simulation-based test, the quality of input test data is very important. A good set of test data should provide a complete functional coverage of the system. Thus one of a future research topic is to study automatic test case generation. For example, given the state-space of a model, how to generate test cases automatically for simulation-based test. This may also imply that there is a need to integrate simulation-based test methods with formal methods such as model checking, etc.

From the robotic application point of view, our current work on distributed robotic design and test can be extended into a more advanced integrated development environment where a generic repository of models for robot control, robot hardware, and the real physical world can be used for specifying autonomous mobile robots. This

developing environment should include an advanced world model that allows virtual environment to be dynamically generated for simulation. Furthermore, it should support the development of robotic systems at all development phases. For example, it may support non-embodied simulation for the purpose of analysis and prototyping; embodied sensor-based simulation for the purpose of design and control logic test; and “robot-in-the-loop” simulation for the purpose of real system test.

Finally, for dynamic reconfiguration of software systems, we have developed variable structure modeling capability. Future research can be conducted to study dynamic reconfiguration in a more general view, and especially, to investigate how software reconfiguration may affect system dependability and reliability.

REFERENCES

- [Alu94] R. Alur and D.L. Dill, A theory of timed automata, *Theoretical Computer Science* 126, 183-235, 1994
- [Ame01] J. Ameghino, A. Tróccoli, G. Wainer. "Models of Complex Physical Systems using Cell-DEVS", *Proceedings of the Annual Simulation Symposium*, Seattle, Washington, 2001.
- [Ant00] G. Antoniol, B. Caprile, A. Potrich, P. Tonella, "Design-code traceability for object-oriented systems". *Annals of Software Engineering* vol. 9: 35-58 (2000)
- [Avi75] Avizienis A., Fault Tolerance and Fault Intolerance: Complementary Approaches to Reliable Computing, *ACM SIGPLAN Notices*, Vol. 10, No. 6, pp. 458-464, June 1975
- [Avi76] Avizienis A., Fault tolerant systems, *IEEE Transactions on Computers*, Vol. C-25, No.12, pp. 1304-1312, 1976
- [Bag91] R. L. Bagrodia, C. Shen, "MIDAS: integrated design and simulation of distributed systems", *Software Engineering*, *IEEE Transactions on*, Volume: 17 , Issue: 10 , Oct. 1991
- [Bal00] Balch, T.; Hybinette, M.: Social potentials for scalable multi-robot formations. *Robotics and Automation*, 2000. *Proceedings. ICRA '00. IEEE International Conference on* , Volume: 1 , 2000 Page(s): 73 -80 vol.1
- [Bal98] Balch, T.; Arkin, R.C.: Behavior-based formation control for multirobot teams. *Robotics and Automation*, *IEEE Transactions on*, Volume: 14 Issue: 6, Dec. 1998 Page(s): 926 -939
- [Bar94] Barros, F.J.; Mendes, M.T.; Zeigler, B.P., "Variable DEVS-variable structure modeling formalism: an adaptive computer architecture application". 'Distributed Interactive Simulation Environments', *Proceedings of the Fifth Annual Conference on* , 7-9 Dec 1994 Page(s): 185 -191
- [Bar97a] Barros, F.J. and B.P. Zeigler, "Adaptive Queueing: A Case Study Using Dynamic Structure DEVS". *International Trans. in Oper. Res.*, 1997. Vol. 4, No. 2, pp 87-98
- [Bar97b] Barros. F.J. "Modeling Formalisms for Dynamic Structure Systems". *ACM Transactions on Modeling and Computer Simulation*, Vol. 7, No. 4, 501-515, 1997
- [Ben91] Benveniste, A.; Berry, G.; The synchronous approach to reactive and real-time systems *Proceedings of the IEEE* , Volume: 79 Issue: 9 , Sept. 1991
- [Bit03] Bithika Khargharia¹, Salim Hariri¹, Manish Parashar², Lewis Ntamo¹, Byoung uk Kim, vGrid: A Framework For Building Autonomic Applications, *Proceedings of the International Workshop on Challenges of Large Applications in Distributed Environments (CLADE'03)*

- [Blair01] Gordon Blair et Al., "The Design and Implementation of Open ORB v2", DS Online Vol. 2, No. 6 2001.
- [Boy93] J. L. Boyd, Designing reactive systems for strong traceability, Carleton University, Ottawa, Ont., Canada, 1993
- [Bro86] R. A. Brooks (1986) "A Robust Layered Control System For A Mobile Robot", IEEE Journal Of Robotics And Automation, RA-2, April. pp. 14-23, March 1986
- [Bro90] R. A. Brooks (1990) "The Behavior Language; User's Guide", M. I. T. Artificial Intelligence Laboratory, AI Memo 1227, April.
- [Bro98] Brown, A.W.; Wallnau, K.C.;"The current state of CBSE", IEEE Software , Volume: 15 Issue: 5, Sep/Oct 1998 Page(s): 37 -46
- [Bru00] B. Bruegge & a. H. Dutoit, Object-Oriented Software Engineering - Conquering Complex and Changing Systems, Prentice Hall, 2000
- [Cao97] Cao, Y. U., A. S. Fukunaga, and A. B. Kahng, Cooperative Mobile Robotics: Antecedents and Directions, Autonomous Robots 4(1): 7-27, 1997
- [Car02] Carpin, S.; Parker, L.E.: Cooperative leader following in a distributed multi-robot system. Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on , Volume: 3 , 2002 Page(s): 2994 -3001
- [Che02] Chen, X., "Dependence management for dynamic reconfiguration of component-based distributed systems", Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE
- [Cho01] Y. K. Cho, "RTDEVS/CORBA: A Distributed Object Computing Environment For Simulation-Based Design Of Real-Time Discrete Event Systems." Ph.D. thesis, University of Arizona, Tucson, AZ 2001
- [Cho03] Y. K. Cho, X. Hu, and B. P. Zeigler: The RTDEVS/CORBA Environment for Simulation-Based Design Of Distributed Real-Time Systems, Simulation: Transactions of The Society for Modeling and Simulation International, 2003, Volume 79, Number 4
- [Cou99] Couretas, J., B. P. Zeigler, U. Patel, "Automatic Generation of System Entity Structure Alternatives: Application to Initial Manufacturing Facility Design." Transactions of the SCS, 1999,16(4), pp. 173-185.
- [Dav00]]A. Davidson, G. Wainer. "Specifying truck movement in traffic models using Cell-DEVS". In Proceedings of the 33rd Annual Symposium on Computer Simulation. Washington, D.C. U.S.A. 2000
- [Der89] Michael L. Dertouzos and Aloysius K. Mok. Multiprocessor on line scheduling of hard real time tasks, IEEE Transactions on Software Engineering, 15(12):1497-1506, December 1989
- [DEVJ] DEVS-Java Reference Guide, www.acims.arizona.edu

- [Dix98] A. Dix, J. Finlay, G. Abowd, R. Beale, Human-Computer Interaction 2nd Edition, Prentice Hall, 1998
- [Dow00] Jim Dowling and Vinny Cahill, "Building a Dynamically Reconfigurable minimumCORBA Platform with Components, Connectors and Language-Level Support", In IFIP/ACM Middleware'2000 Workshop on Reflective Middleware, New York, USA, April 2000.
- [Dow01] Jim Dowling and Vinny Cahill, Dynamic Software Evolution and The K-Component Model, Workshop on Software Evolution, OOPSLA 2001
- [Eri] Eric E. Allen, Diagnosing Java code: Assertions and temporal logic in Java programming <http://www-106.ibm.com/developerworks/java/>
- [FAS] Warfighter Information Network-Tactical (WIN-T): <http://www.fas.org/man/dod-101/sys/land/win-t.htm>
- [Fin88] Finkelstein, L.; Land, F.; Carson, E.R.; Westcott, J.H., Systems theory and systems engineering, Science, Measurement and Technology, IEE Proceedings A, Volume: 135 Issue: 6, July 1988
- [Fro95] J. Frossl, J. Gerlach and T. Kropf. - An efficient algorithm for real-time symbolic model checking. Pros. Europ. Design & test Conf. (ED&TC'95), 1995, pp. 15-20
- [Ger02] E. Gery, D. Harel, and E. Palachi, Rhapsody, "A Complete Life-Cycle Model-Based Development System", Integrated Formal Methods, Third International Conference, IFM 2002
- [Gho94] K. Ghosh, B. Mukherjee, K. Schwan, "A Survey of Real-Time Operating Systems", Technical report, Atlanta, Georgia 30332-0280, College of Computing, Georgia Institute of Technology, 1994
- [Gil62] Gill, Arthur, Introduction to the theory of finite-state machines, New York, McGraw-Hill 1962
- [Gom00] Hassan Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with Uml, Addison-Wesley Longman Publishing Co. 2000
- [Gom01] M. Gomez., "Hardware-in-the-Loop Simulation", Embedded Systems Programming, December, 2001
- [Gom93] Hassan Gomaa, Software Design Methods for Concurrent and Real-Time Systems, Addison-Wesley Longman Publishing Co. 1993
- [Gon02] F. G. Gonzalez, W. J. Davis, "A New Simulation Tool for the Modeling and Control of Distributed Systems", SIMULATION: Transactions of the Society for Modeling and Simulation International, Volume 78, Number 9, 2002
- [Gor02] J. Gorinsek, S. Van Baelen, Y. Berbers and K. De Vlaminc, EMPRESS: Component Based Evolution for Embedded Systems, ECOOP 2002 Workshop on Unanticipated Software Evolution (USE2002), G. Kniesel, P. Costanza, M.

Dimitriev (eds.) Malaga, Spain, June 2002

- [Hal93] N.Halbwachs. - Synchronous Programming of Reactive Systems. - IMAG Institute, Grenoble, France, Kluwer Academic Publishers, 1993, The Kluwer international series in engineering and computer science.
- [Hon97] J.S. Hong, and T.G. Kim, "Real-time Discrete Event System Specification Formalism for Seamless Real-time Software Development," Discrete Event Dynamic Systems: Theory and Applications, vol. 7, pp.355-375, 1997.
- [Hu01] X. Hu, B.P. Zeigler, J. Couretas. "DEVS-On -A-Chip: Implementing DEVS In Real-Time Java On A Tiny Internet Interface For Scalable Factory Automation", IEEE International Conference on Systems, Man, And Cybernetics, October, 2001
- [Hu02] X. Hu, and B. P. Zeigler: An Integrated Modeling and Simulation Methodology for Intelligent Systems Design and Testing. Performance Metrics for Intelligent Systems Workshop, August, 2002
- [Hu03a] X. Hu, and B.P. Zeigler, " Model Continuity in the Design of Dynamic Distributed Real-Time Systems", submitted to IEEE Transactions On Systems, Man And Cybernetics- Part A: Systems And Humans
- [Hu03b] X. Hu, and B. P. Zeigler: Model Continuity to Support Software Development for Distributed Robotic Systems: A Team Formation Example, accepted in June 2003, Journal of Intelligent & Robotic Systems, Theory & Application
- [Hu03c] X. Hu, B. P. Zeigler, and S. Mittal, "Variable Structure in DEVS Component-based Modeling and Simulation", accepted and to be published, Simulation: Transactions of The Society for Modeling and Simulation International, November 2003
- [Ioc01] Luca Iocchi, Daniele Nardi, Massimiliano Salerno, Reactivity and Deliberation: a survey on Multi-robot systems, Lecture Notes in Computer Science, Volume 2103, Springer-Verlag Heidelberg, 2001
- [Jan02] R. S. Janka, L. M. Wills, L. B. Baumstark, "Virtual Benchmarking and Model Continuity in Prototyping Embedded Multiprocessor Signal Processing Systems", IEEE Transactions on Software Engineering, September 2002 (Vol. 28, No. 9)
- [Kim97] Kim, K.H., "Object Structures for Real-Time Systems and Simulators", IEEE Computer, August 1997, pp.62-70
- [Kim99] D. Kim, S.J. Buckley, and B.P. Zeigler. "Distributed Supply Chain Simulation in a DEVS/CORBA Execution Environment," in Proceeding of WSC. Phoenix, Arizona, 1999
- [Kon01] Fabio Kon, Tomonori Yamane, Christopher K. Hess, Roy H. Campbell and M. Dennis Mickunas, "Dynamic Resource Management and Automatic Configuration of Distributed Component Systems", USENIX COOTS'2001.

- [Kow01] Kowalczyk, W.: Multi-robot coordination. Robot Motion and Control, 2001 Proceedings of the Second International Workshop on, 2001 Page(s): 219 -223
- [Kri97] C.M. Krishna and K.G. Shin, Real-time systems, McGraw-Hill, New York, 1997.
- [Lap92] Laprie J.C. (ed.), Dependability: Basic Concepts and Terminology, Dependable Computing and Fault-Tolerant Systems Series, Vol. 5, Springer Verlag, 1992
- [Lee01] Edward A. Lee, etc. OVERVIEW OF THE PTOLEMY PROJECT MARCH 6, 2001 Technical Memorandum UCB/ERL M01/11
- [Lei80] D. W. Leinbaugh Guaranteed response time in a hard realtime environment. IEEE Transactions on Software Engineering, January 1980
- [Lok99] Lokesh Bajaj, Mineo Takai, Rajat Ahuja, and Rajive Bagrodia. Simulation of large-scale heterogeneous communication systems. In Proceedings of IEEE Military Communications Conference (MILCOM '99), November 1999
- [Man03] M. Mathure, V. Jonnalagadda, J. Zalewski, A Heterogeneous Architecture and Testbed for Simulation of Large-Scale Real-Time Systems, 7th IEEE Int'l Symposium on Distributed Simulation and Real-Time Applications, 2003
- [Mar01] Martin, G.; Lavagno, L.; Louis-Guerin, J., Embedded UML: a merger of real-time UML and co-design, Hardware/Software Codesign, 2001. CODES 2001. Proceedings of the Ninth International Symposium on , 25-27 April 2001
- [Mic98] Michael Puttré, Simulation-based design puts the virtual world to work Design News February 16, 1998
- [Mor96] J. Morse and S. Hargrave, The increasing importance of software. Electronic Design, Vol. 44 (1), Jan. 1996
- [MOV] <http://www.acims.arizona.edu/PROJECTS/MultiRobot.mpg>
- [Muz02] Muzy, A., G. Wainer, E. Innocenti, A. and Aiello, J.F. Santucci. "Comparing simulation methods for fire spreading across a fuel bed", In Proceedings of AIS'2002, Lisbon, Portugal.
- [Nat95] Natarajan, Swaminathan, Imprecise and approximate computation, Kluwer international series in engineering and computer science Kluwer Academic Publishers, 1995
- [Nee99] Sandeep Neema, Ted Bapty, Jason Scott, Adaptive Computing and Run-time Reconfiguration, 1999
- [OMG1] OMG web site URL: <http://www.omg.org/>
- [OMG2] OMG Request for Proposal: UML 2.0 Superstructure RFP, OMG document: ad/2000-08-09
- [OMG3] OMG Request for Proposal: Action Semantics for the UML RFP, OMG

document: ad/98-11-01

- [OMG4] OMG Request for Proposal: UML profile for Scheduling, Performance and Time, OMG document: ad/99-03-13
- [OMG5] OMG Unified Modeling Language Specification, Version 1.4. <http://www.omg.org>.
- [Pal01] N Noël De Palma, Philippe Laumay and Luc Bellissard. Ensuring Dynamic Reconfiguration Consistency. Sixth International Workshop on Component-Oriented Programming (WCOP 2001) at ECOOP 2001, Budapest (Hungary), 2001
- [Par00] Parker, L. E.: Current State of the Art in Distributed Autonomous Mobile Robots. In L. E. Parker, G. Beker, J. Barhen (Eds.), Distributed Autonomous Robotics Systems 4, pp.3-12, Springer, 2000
- [Par03] Sunwoo Park, "Hierarchical Model Partitioning for Distributed Simulation of Hierarchical and Modular DEVS Models", Ph. D. Dissertation, Univ. of Arizona, May 2003.
- [Pau03] Paul K. Davis and Robert H. Anderson, Improving the Composability of Department of Defense Models and Simulations", report for Defense Modeling and Simulation Office, 2003
- [Pau96] P. Paulin, M. Cornero, C. Liem, F. Nacabal, C. Donawa, S. Sutarwala, T. May and C. Valderrama, Trends in embedded systems technology. Hardware/software co-design, 1996
- [Pau97] P. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Coossens, Embedded software in real-time signal processing systems: Application and architectural trends, Proc. of IEEE, vol. 85(3), Mar. 1997, pp. 419-435
- [Paw02] Pawletta T., B. Lampe, "A DEVS-Based Approach for Modeling and Simulation of Hybrid Variable Structure Systems", Lecture Notes in Control and Information Sciences, No. 279, Springer Pub. 2002, pp. 107-129.
- [Pei02] Peipelman. J., N. Alvarez, K. Galinet, R. Olmos.: 498 A & B Technical Report. Department of Electrical and Computer Engineering, University of Arizona, 2002
- [Phi97] Phillip A. Laplante, Real-time systems: Design and Analysis, 2nd Ed. IEEE Press, Piscataway, NJ, 1997.
- [Pra01] H. Praehofer, "Towards a systems methodology for object-oriented software analysis", Discrete Event Modeling and Simulation Technologies: A Tapestry of Systems and AI-Based Theories and Methodologies, Springer, 2001
- [Pre97] R.S. Pressman, Software Engineering: A Practitioner's Approach, fourth ed. New York: McGraw-Hill, 1997.
- [Pri57] Prior, A. N., 1957, Time and Modality, Oxford: Clarendon Press.

- [Pri67] Prior, A. N., 1967, Past, Present and Future, Oxford: Clarendon Press.
- [Pri69] Prior, A. N., 1969, Papers on Time and Tense, Oxford: Clarendon Press.
- [Pto] <http://ptolemy.eecs.berkeley.edu/>
- [Ram01] B. Ramesh, M. Jarke, "Toward reference models for requirements traceability", Software Engineering, IEEE Transactions on , Volume: 27, Issue: 1 , Jan. 2001
- [Ran02] Randall S. Janka, Linda M. Wills, Lewis B. Baumstark, Jr., Virtual Benchmarking and Model Continuity in Prototyping Embedded Multiprocessor Signal Processing Systems, IEEE Transactions on Software Engineering, September 2002 (Vol. 28, No. 9)
- [Ras01] Rastofer, U.; Bellosa, F., Component-based software engineering for distributed embedded real-time systems, Software, IEE Proceedings, Volume: 148 Issue: 3, June 2001
- [Rem93] REMBOLD, U., B.O. NNAJI, A. STORR, Computer Integrated Manufacturing and Engineering, Addison-Wesley Publishing Company, Wokingham, England, 1993.
- [Rob00] Paul Robertson, Robert Laddaga, and Howie Shrobe, "Introduction: the first international workshop on self-adaptive software", Lecture Notes in Computer Science, 2000, pp. 1-10
- [Rot72] Rothermel, R., "A mathematical model for predicting fire spread in wildland fuels". Research Paper INT-115. Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station, 1972
- [Roz90] Rozenblit, J. W., J. Hu, B.P. Zeigler, and T.G.Kim, "Knowledge-Based Design and Simulation Environment (KBDSE): Foundational Concepts and Implementation," J. Operations Research Society 41(6), 475-489, 1990
- [Sak00] M. Saksena and P. Karvelas, Designing for Schedulability: Integrating Schedulability Analysis with Object-Oriented Design, In Proceedings, 12th Euromicro Conference on Real-Time Systems, June 2000.
- [Sak99] M. Saksena and Y. Wang., Scheduling Fixed-Priority Tasks with Preemption Threshold, In Proceedings, IEEE International Conference on Real-Time Computing Systems and Applications, December 1999.
- [SBA] U.S. Navy simulation-based acquisition website: <http://www.ntsc.navy.mil/Resources/Library/Acqguide/sba.htm>
- [Sch00] W. Schulte, "Why Doesn't Anyone Use Formal Methods?", Integrated Formal Methods, Second International Conference, IFM 2000
- [Sch02] Schulz, S.; Buchenrieder, K.J.; Rozenblit, J.W.: Multilevel testing for design verification of embedded systems. IEEE Design & Test of Computers Volume: 19 Issue: 2 , March-April 2002

- [Sel94] B. Selic, G. Gullekson, P. T. Ward. Real-Time Object-Oriented Modeling. Wiley. 1994.
- [Sel98] Bran Selic, Using UML for Modeling Complex Real-time Systems, white paper, March 11, 1998
- [Sgr00] Sgroi, M.; Lavagno, L.; Sangiovanni-Vincentelli, A., "Formal models for embedded system design", Design & Test of Computers, IEEE , Volume: 17 Issue: 2 , April-June 2000
- [Sha01] Slan C. Shaw: Real-time Systems and Software, 2001, John Wiley & Sons
- [Sha01] Slan C. Shaw: Real-time Systems and Software, 2001, John Wiley & Sons
- [Son01] Feijun Song; Folleco, A.; An, E.: High fidelity hardware-in-the-loop simulation development for an autonomous underwater vehicle. OCEANS, 2001. MTS/IEEE Conference and Exhibition, Volume: 1 , 2001
- [Ste03] Steven B. Hall, Shankar M. Venkatesan, Donald B. Wood, "A Faster Implementation of DEVS in the Joint MEASURE Simulation Environment", in Proc. of Summer Computer Simulation Conference, Montreal, July 2003.
- [Ste91] D. B. Stewart and P. K. Khosla, Real time scheduling of sensor based control systems, In Eighth IEEE Workshop on Real-Time Operating Systems and Software, May 1991
- [Tho00] Filip Thoen and Francky Catthoor, "Modeling, Verification, and Exploration of task-level concurrency in real-time embedded systems. Kluwer Academic Publishers, 2000, pp.46
- [Uhr01] Uhrmacher, A.M., "Dynamic Structures in Modeling and Simulation - A Reflective Approach". ACM Transactions on Modeling and Simulation, Vol.11. No.2 , p. 206-232, April 2001.
- [Uhr93] Uhrmacher, A.M. "Variable Structure Models: Autonomy and Control - Answers from Two Different Modeling Approaches". Proc. AI, Simulation, and Planning in High Autonomy Systems. IEEE Computer Society Press, 1993, 133-139
- [Vas93] Vasconcelos, J. M, Modeling Spatial Dynamic Ecological Processes with DEVS-Scheme and Geographical Information Systems, Ph.D. Dissertation, Dept. of Renewable and Natural Resources, University of Arizona, Tucson, U.S.A., 1993
- [Vie03] Vieri Del Bianco, Luigi Lavazza, Marco Mauri, et al., "Towards UML-based formal specifications of component based real-time software" pp. 118 - 134 Lecture Notes in Computer Science Publisher: Springer-Verlag Heidelberg Volume: Volume 2621 / 2003
- [Vil97] J. Villasenor, W. H. Mangione-Smith, "Configurable Computing," Scientific American, pp.66-71, June 1997.
- [Wan97] Wang. J.: Methodology and design principles for a generic simulation platform

for distributed robotic system experimentation and development. *Systems, Man, and Cybernetics*, 1997. *Computational Cybernetics and Simulation.*, 1997 IEEE International Conference on , Volume: 2, 1997 Page(s): 1245 -1250 vol.2

- [Wel01] Wells, R.B.; Fisher, J.; Ying Zhou; Johnson, B.K.; Kyte, M.: Hardware and software considerations for implementing hardware-in-the-loop traffic simulation. *Industrial Electronics Society*, 2001. *IECON '01. The 27th Annual Conference of the IEEE* , Volume: 3 , 2001
- [Wol86] S. Wolfram, *Theory and Applications of Cellular Automata*, World Scientific, Singapore, 1986
- [Zei00] Zeigler, B.P., T.G. Kim, and H. Praehofer.: *Theory of Modeling and Simulation*. 2 ed. 2000, New York, NY: Academic Press
- [Zei76] Zeigler, B.P., *Theory of Modelling and Simulation*, Wiley, N.Y., 1976
- [Zei89] Zeigler, B. P. Concepts for distributed knowledge maintenance in variable structure models. In *Modelling and Simulation Methodology - Knowledge Systems Paradigm*, B. Zeigler, M. Elzas, and T. Oeren, Eds. Elsevier North-Holland, Inc., Amsterdam, The Netherlands, pp.45-54, 1989.
- [Zei90] Zeigler, B.P. 1990. *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*. Academic Press.
- [Zei96] Bernard P. Zeigler, Yoonkeon Moon, Doohwan Kim, Jeong Geun Kim: *DEVS-C++: A High Performance Modelling and Simulation Environment*. *HICSS (1)* 1996: 350-359
- [Zei97] Zeigler, B.P., H. Sarjoughian, and W. Au. "Object-Oriented DEVS", *Proc. Enabling Technology for Simulation Science*, SPIE AeoroSense 97. 1997. Orlando, FL.
- [Zei99] Zeigler, B.P., H.S. Sarjoughian, "Support for Hierarchical Modular Component-based Model Construction in DEVS/HLA", *Simulation Interoperability Workshop*, March, Orlando, FL., 1999.
- [Zha87] Wei Zhao, Krithi Ramamritham, and J. A. Stankovic, Preemptive scheduling under time and resource constraints, *IEEE Transactions on Computers*, C-36(8):949-960, August 1987
- [Zho93] Zhou, MengChu, and Frank DiCesare, *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*, Kluwer Academic Publishers, Boston, 1993