Relational Specification of Interface Modules for Real-time Systems

By

©Yingzi Wang, B. Eng.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of

Master of Engineering

Faculty of Engineering and Applied Science
Memorial University of Newfoundland

May 2006

St. John's Newfoundland

Abstract

Documentation plays a key role as a component of design process, and a preview of a task before it comes to be executed. A well-specified task might not take less implementation time than one without documents, but one of the obvious advantages is that misunderstandings are avoided and readable specification makes it easy for the successive developers to exploit or modify the software or hardware design. Interface Modules (IM) are modules that encapsulate input or output device hardware and the related software, so that the application software can be written without specific knowledge of the particular devices used.

In this work, we present a technique to IM specification that very few researchers pay attention to in the formal specification area. The technique is an extension of the System Requirements Documentation technique presented in [58], which is based on the Software Cost Reduction (SCR) method. Since an IM interacts with both the external environment and other software modules, the technique is used to specify a hybrid of software and corresponding hardware devices. The interface quantities are modeled as functions of time and the behavior is described in terms of conditions, events and mode classes.

The contributions of this work to the field of formal specification in general, consists in extending SCR method with introducing *access programs* and *parameterized modes* to specify Interface Modules for real-time systems. In the SCR method, conditions are defined as boolean functions of monitored or controlled variables. Such definitions are limited to address the relationship to the environment. For interface modules, we use access programs as conditions so that the relationship of the IMs to other software modules can be expressed. The *parameterized modes* simplify the specification by grouping a set of modes with particular values in the same mode name. This technique facilitates concise and formal description of the module behav-

ior, including tolerances and delays.

Acknowledgements

I sincerely thank my supervisor, Dr. Dennis K. Peters, for his great guidance, thoughtful critisizm and constant support. I am thankful to our colleagues in Electrical and Computer Engineering at Memorial University, and in particular to An Zhiwei for many insightful discussions and technical support. Support from the Natural Sciences and Engineering Research Council and the Faculty of Engineering at Memorial University are greatly acknowledged.

Contents

Abstract i

Acknowledgements iii

List of Acronyms ix

1 Introduction 1

1.1 Interface Modules 2

1.2 Computer System Documentation 3

1.2.1 The Four Variable Requirements Model 3

1.2.2 System Requirements Document 6

1.2.3 Module Interface Specification 6

1.3 Interface Module Specifications 7

1.4 Scope 8

1.5 Outline 8

2 Related Work **10**

2.1 System Requirements Specification 10

2.1.1 Automata Based Methods 10

2.1.2 Abstract Model Based Methods 12

2.1.3 Predicate Logic Based Methods 12

2.1.4 SCR Requirements Method 14

2.2 Interface Specification 14

2.3 Tabular Expression 16

2.3.1 Normal Tables 18

2.3.2 Vector Tables 18
2.3.3 Decision Tables 19

3 SCR Requirements Documentation Introduction 22
3.1 SCR Requirements Documentation 22
3.1.1 Identifier Annotations 23
3.1.2 Conditions 23
3.1.3 Events and Event Classes 24
3.1.4 Mode and Mode Classes 28

4 Specifying Interface Modules 31
4.1 SCR Extensions 31
4.1.1 Using Access Programs as Conditions 32
4.1.2 Public Variables 33
4.1.3 Parameterized modes 33
4.1.4 Callback functions in the User Interface 36
4.2 Interface Modules Specification 39
4.2.1 Monitored and Controlled Quantities 39
4.2.2 Mode Classes 40
4.2.3 Controlled Value Functions 41
4.2.4 Timing Requirements 43
4.2.5 Environmental Constraints 43
4.3 Specification of Human-Machine Interface Modules 44
4.4 Concurrent Applications 44
4.5 Discussion 46

5 Sample Applications 49
5.1 A Robot Arm Control System 49
5.1.1 Robot Interface Module Specification 49
5.1.2 Mode Class $^{Cl}\mathsf{Motion}$ 50
5.1.3 Mode Class $^{Cl}\mathsf{Gripper}$ 53
5.1.4 Conditions 53
5.1.5 Controlled Value Functions 54
5.1.6 Constants 55

5.1.7 Environmental Constraints 56
5.2 Public Variable Interface of Robot Arm Control System 57
5.2.1 Public Variables 57
5.2.2 Mode Class ^{Cl}Motion 58
5.2.3 Controlled Value Functions 59
5.2.4 Conditions 60
5.3 Callback Function Specification 61
5.3.1 Environmental Quantities 62
5.3.2 Mode Class ^{Cl}MouseListener 63
5.3.3 Mode Class ^{Cl}mouseMotionListener 64
5.3.4 Controlled Value Functions 66
5.3.5 Conditions 67
5.3.6 Dictionary 67
5.4 Automated Teller Machine 68
5.4.1 Card Reader 69
5.4.2 KeyboardAdaptor 73
5.5 Discussion 75

6 Conclusions 77
6.1 Contributions 77
6.2 Applicability of This Work 78
6.3 Limitations of the Method 78
6.4 Future Work 79
6.5 Conclusions 80

List of Figures

1.1 Fire Alarm Control System 2
1.2 Modularized Fire Alarm Control System 3
1.3 Four Variable Model 4

2.1 Raw Table Skeleton of Table 17

4.1 Robot system 40

List of Tables

2.1 An example of Normal Table . . . 17
2.2 An example of Vector Table . . . 19
2.3 An example of Decision Table . . . 21

3.1 Identifier Annotations . . . 24
3.2 Event Notation . . . 25
3.3 Event Class Notation . . . 26
3.4 Mode Transition Table 1 . . . 29
3.5 Mode Transition Table 2 . . . 29

4.1 Conditions . . . 32
4.2 Access Programs . . . 33
4.3 Transition Relation ^{Cl}Motion . . . 34
4.4 Public Variables . . . 34
4.5 Accessing Public Variables . . . 34
4.6 Transition Relation for Non-parameterized mode class . . . 35
4.7 Transition Relation for Parameterized mode class . . . 35
4.8 Controlled Variable Function . . . 38
4.9 Environmental Quantities . . . 40
4.10 Mode Transition Relation ^{Cl}Gripper . . . 42
4.11 Control Value Function . . . 42

List of Acronyms

Acronym	Description	Page (definition)
SRD	System Requirement Documentation	6
IM	Interface Module	2
IMS	Interface Module Specification	7
SDL	Specification and Description Language	11
UML	Unified Modelling Language	11
VDM	Vienna Development Method	12
SCR	Software Cost Reduction	14

Chapter 1

Introduction

Documentation plays a key role as a component of software and hardware design process, and a preview of a task before it comes to be executed. Complete and precise system requirements documentation can be used to verify the feasibility of a project before it is implemented. Faults can be found in the early stages, if there are any, and thus the cost of maintenance for the project can be reduced.

To reduce complexity, a system can be decomposed into a set of modules, each of which performs a certain task in the system [55]. As a part of system modules, Interface Modules form the bridge between system software and the environment. Often, interface modules will encapsulate input and output hardware devices (e.g., actuators or sensors) and related software. If the devices change, the interface module will ideally be the only component that is required to change. The other modules in the system are "protected" by the interface modules in that little or no change is required.

The unique characteristics of Interface Modules require the capability of a technique to specify a relation of a combination of software and corresponding hardware devices. As a part of the system behavioral documentation, interface module specification describes the visible behavior of a practical interface module. A few authors have researched on Interface Modules specification, e.g. see [33, 39]. This work investigates techniques for using techniques based on [58], including mode classes and conditions to represent the behavior of Interface Modules for real-time systems.

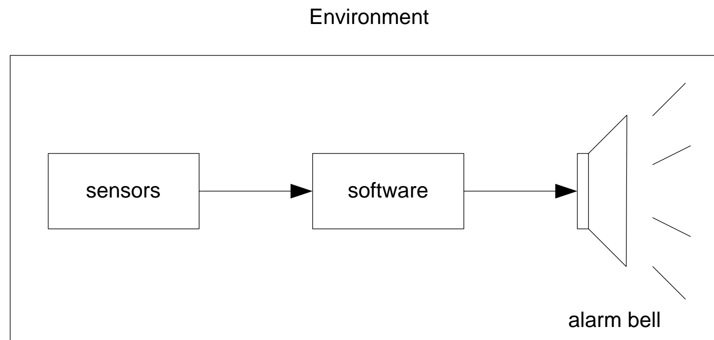

Figure 1.1: Fire Alarm Control System

1.1 Interface Modules

Interface Modules (IM) are modules that communicate between system software and the environment external to the system, encapsulating input or output device hardware and the related software, so that the application software can be written without specific knowledge of the particular devices used [54, 14]. An IM reduces the complexity of the system design by isolating the interface details from the rest of the system software. This is particularly important in embedded systems, where the IM will often contain special purpose hardware devices (e.g., actuators or sensors): replacing or modifying a device should only lead to changes in the IM, rather than requiring changes to other modules in the system. If the interface hardware is not explicitly encapsulated, when a device changes, software depending on it will also need to change, so the change could have surprising and widespread ramifications.

A fire-alarm system, for example, as illustrated in Figure 1.1, illustrates the basic concept of Interface Modules. The sensors measure the current temperature in the room once every time interval and then send it to the software which processes the data under some constraints. When temperature is above the threshold, the software will send out a signal to start the fire-alarms. If the whole system is decomposed into a set of modules, as shown in Figure 1.2, both Input Interface and Output Interface are examples of Interface Modules. Changes to the sensor or alarm hardware will not effect other system software, but the interface modules only.

An ideal interface module will:

- be the only component that needs to change if the devices change;

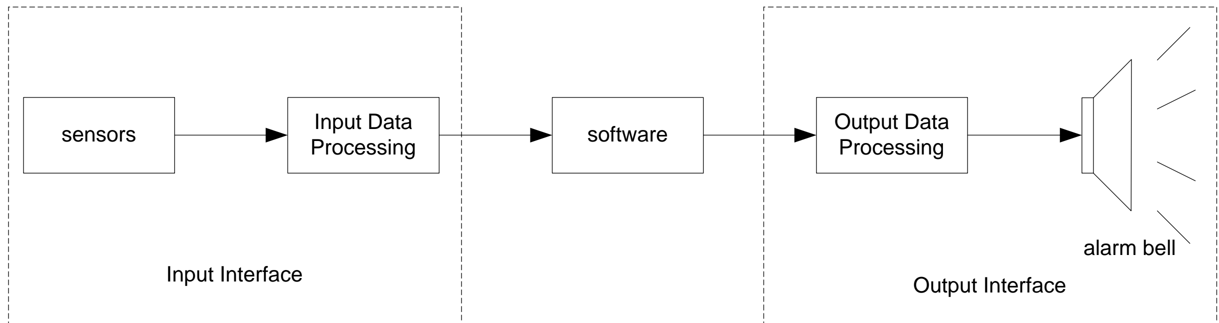

Figure 1.2: Modularized Fire Alarm Control System

- not need to change unless the devices change;
- be relatively small and simple in structure so that it can be easily changed if necessary.

1.2 Computer System Documentation

By *specification*, we mean a description of the *acceptable behavior* of an entire system, sub-system, or component. A specification should describe *what* is to be built, omitting details of *how* this will be achieved. A system or component that satisfies the specification can be implemented in hardware, software, or combination of both. An important goal is to avoid both overspecification and underspecification. Thus a specification must characterize *every* black-box behavior acceptable to the customer. Further, it should be free of implementation bias since the behavior and the interface of the module are clearly specified [37].

1.2.1 The Four Variable Requirements Model

When specifying system and software requirements, it is important to distinguish quantities that are external to the system (environmental quantities) from those that are internal to the system [68, 69]. The Four Variable Model [56, 68, 69], addresses this issue and is adopted as a framework of this work. According to this model, environmental quantities will include physical properties(e.g., temperatures, pressure, location of objects), values of images on output display devices, settings of input

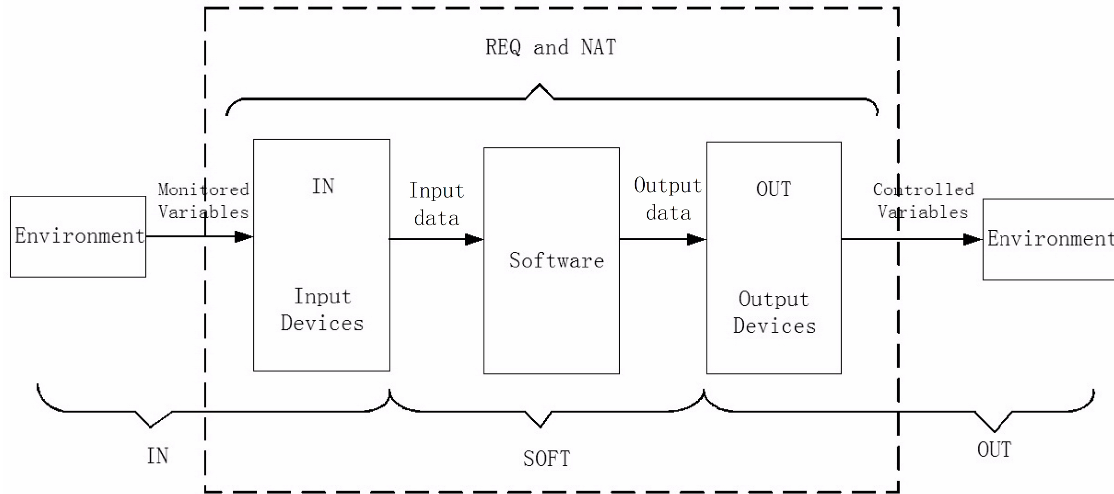

Figure 1.3: Four Variable Model

switches, and settings of controlled devices. They are independent of the chosen solution and are apparent to the "customer". Also, in real-time systems, environmental quantities can be modeled by functions of time [35, 56, 63].

Environmental quantities can be classified into two sets: the *controlled* quantities and *monitored* quantities. A monitored variable represents an environmental quantity that influences system behavior, a controlled variable denotes an environmental quantity that might be changed due to the operation of the system. In Four Variable Model, as illustrated in Figure 1.3, the required system behavior is described as a set of mathematical relations on four sets of variables — monitored and controlled variables and input and output data items. Input and output data items, which are the input to and output from these devices, are treated as resources for other system modules.

Input devices (e.g., sensors) measure the monitored quantities and output devices set the controlled quantities. The variables that the devices read and write are called input and output data items.

The definition of four relations in the Four Variable Model — REQ, NAT, IN and OUT is given in [69], which is the framework for describing system requirements. NAT defines the natural constraints on system behavior, such as those imposed by physical laws and by the system environment. REQ describes requirements by giving

the relation that the system must maintain between the monitored and the controlled quantities. The relation IN specifies the accuracy with which the input devices measure the monitored quantities and the relation OUT specifies the accuracy with which the output devices set the controlled quantities. The software requirements specification, called SOFT, defines the required relation between the input and output data items. In the original Four Variable Model, IN and OUT describe behavior of devices (i.e., hardware only). In this work, we take a slightly different view of IN and OUT in that they may contain hardware and software. In this view IN and OUT can be determined from the combination of all of the IM in the system. Accuracy and tolerance are used for the purpose of modeling the ideal system to describe errors and delays that may be introduced anywhere in the system.

Since IMs interact with both other software modules and the environment external to the system, they are examples of hybrid systems, which contain both discrete and continuous components. Among the quantities that interact with IM, some of these quantities are continuous, while others are discrete. For example, the change of the environmental quantities like temperature, pressure, and the position of a moving car is continuous; the values of variables measured in the system corresponding to these environmental quantities are discrete. Thus, such combination of continuous and discrete viriables presents new challenges for IM specification.

Writing specifications for interface modules is different from that for software modules in that:

- IM interact with both environment and other software modules, so that discrete and continuous valued variables are integrated in IM, while Software Modules (SM) contain only discrete valued variables and discrete time. The Interface Module Specification (IMS) must provide a suitable technique to clearly specify such combinations.
- IM is the medium between system software and the environment, so that the device changes might require modification in IM, but ideally no other SM changes occur. IMS must explicitly specify all environmental quantities relative to the IM and give the acceptable behavior of the IM.

Interface modules are modules in the system which provide interface between the environment and other modules in the system. When specifying interface modules,

they can be viewed as "systems", which are sub-systems in the target system. The monitored variables for the IM are either the monitored variables or output data items for the target system, and the controlled variables for the IM are either the controlled variables or the input data items for the target system.

1.2.2 System Requirements Document

The goal of the System Requirements Documentation (SRD) is to precisely describe a set of acceptable system behaviors. The idea is to make the "what decisions" explicitly up front, not implicitly during design and implementation [37]. The SRD supports the system development process in a number of important ways; it:

- Serves as a contract between the users and the developers;
- Ensures that developers need not decide what is the best for users;
- Provides essential support for independent verification;
- Supports estimates of time and resources;
- Provides protection against personnel turnover;
- Supports the maintainers.

While sometimes the SRD for an industrial system is large and complex so that it does not make easy reading, it provides precise answers to important questions about what must be built. Also important, it provides a framework in which to ask precise questions. To avoid overspecification, the SRD should describe the system behavior as a mathematical relation between entities in the system's environment.

1.2.3 Module Interface Specification

Module Interface Specification (MIS) describes the observable behavior of the module in the system. A module is the basic unit of development and change in the design process. It could be a portion of a program that carries out a specific function, or possibly with related hardware. Each module is defined according to the information-hiding principle, so that module users (e.g., software developers) can use the module without knowing how it is built.

The module interface is the set of assumptions that the developers of external programs may make about the module. It includes restrictions on the way that modules may be used. Modules communicate either by one module using access programs from other modules, or by one module being notified of an event that was signaled by the other modules. The interface consists of assumptions about the availability of the access programs, the syntax of the calls on the access programs, the behavior of the access programs, and the meaning of events [37].

1.3 Interface Module Specifications

Interface Module Specifications (IMS) are components of the System Design Specification (SDS), as described in [58, 56]. Each treats a module as a "black box", identifying those programs that can be invoked from outside of the module (access programs), and describing the externally-visible effects of using them. Like other module specifications, the IMS perform a key role in modular system development to four groups of people: designers, developers, verifiers, and end users. An IMS illustrates to the module designer what behavior is required of the module for design and review. It provides the developer with a clear statement of the required task and allows it to be implemented without communicating with other module designers. In addition, the IMS can be used to verify that the module internal design is correct or that a module implementation obeys the convention. Designers of other modules in the system can use the IMS to understand what behavior they can expect from the module. Also the developers are freed from having to know implementation details about module internals. As a part of the system design process, the IMS and the system architecture can be used to verify that the design satisfies the system requirements [37].

Interface Module Specifications (IMS) describe the visible behavior of a particular module — an Interface Module. Like other modules in the system, the visible behavior of IM can be described by specifying the Module Interface. Each module interface document describes the aspects of module behavior visible to other developers using the module.

As mentioned in Section 1.2.1, the IN and OUT relation used here are slightly different from that in [56] in that they may contain hardware and software, rather

than the pure hardware devices. Unlike other module specifications (e.g., for pure software or hardware modules), IMS must describe a combination of software and corresponding hardware devices.

1.4 Scope

This thesis reports the results of an investigation of techniques for using a readable form of system design documentation to specify the observed behavior of Interface Modules. The following issues are addressed:

- The observable behavior of the Interface Modules.
- How the Interface Modules connect with environment and software modules.

The Interface Modules discussed in this work are assumed to be components in the system that are decomposed in the ideal way according to the information hiding principle, so that the relationship between Interface Modules and other Software Modules can be specified explicitly. In practice, the system might not always be divided in such an ideal way. Therefore the Interface Modules might be of large size with complex structure. If the system is not well decomposed, a device change might require modification to other components in the system, i.e., software modules.

The application of the technique that is presented in this thesis is not limited to specifying interface modules, but also can be used to specify modules with relationship to software modules, i.e., interface of a software module. This work is not suitable to the electronic systems (e.g., Very Large Scale Integration circuit (VLSI)).

The assumption of our method for specifying Interface Modules is that System Requirement Documentation (SRD) has been given. Some of the parameters, related to Interface Modules are provided by the SRD, i.e. monitored and controlled variables.

1.5 Outline

Chapter 2 surveys some research work of formal specification that is related to interface module specifications.

Chapter 3 presents Software Cost Reduction (SCR) method in detail as a framework of this thesis. The notations and definitions of the method that are adopted to specify interface modules is addressed.

Chapter 4 describes the techniques that are extended from Peters' work in [58]. The main contribution of this work is shown in this chapter.

Chapter 5 provides two examples for the Interface Module specification method — a robot arm control system and an Automated Teller Machine (ATM). The specification of the user interface module is also discussed in this chapter.

Chapter 6 discusses the results of applying this work, suggests some future research in this area and draws some conclusions.

Chapter 2

Related Work

Two areas of research are most closely related to this work: Specifying system requirements and Interface Specification. Some of the most relevant work in these areas is as follows.

2.1 System Requirements Specification

The formal techniques that are mainly applied in System Requirements Specification are divided into Finite State Automata (FSA) based methods, Abstract Model based methods and Predicate Logic based methods. Some of the most relevant work in these areas is as follows. SCR Requirement Method, one of the FSA based methods, is emphasized in Section 2.1.4 as the foundation of this thesis and discussed in detail in Chapter 3.

2.1.1 Automata Based Methods

Most of the "popular" formal specification techniques, (e.g., SDL [38], statecharts [25], Hybrid Automata [3, 47], and Petri Nets [50]) are based on automata theory. They model the target system and its environment as one or more FSA, and describe the required behavior of the system in terms of operations on that model.

One of the most widely used techniques in specifying the requirement of real-time systems is Statecharts [25, 26]. It extends traditional FSA with nested states, parallel (AND) or choice (OR) composition of state machines. Its notation allows

relatively complex systems to be described using multiple levels of nesting so that the specification is still understandable. Real-time requirements are described using an implicit clock variable and timeout events.

Statecharts ideas have been adopted by several other techniques, for example RSML [29, 46] and Modecharts [41]. In addition, [66] shows that the expressive power of Statecharts increases when it is combined with temporal logic.

Specification and Description Language (SDL) [38] is mainly a language to specify and describe the logic of processes independently of implementation techniques, and is also based on a FSA approach. In SDL, which allows textual and graphical representations, a system is viewed as consisting of interconnected blocks. A system communicates with the environment and its parts (blocks) communicate with each other by channels which carry signals.

The representation of real-time systems can be realized in timed automata as an abstract model by defining timed state sequences [18, 4]. Each state in the time sequence includes an observation variable that satisfies the propositional constraint and a clock interpretation that satisfies the timing constraint. As a different approach, timed-transition systems can be employed in real-time systems [27, 48, 36]. By restricting the time at which transitions may occur, time is incorporated into the transition system model, which includes a set of propositions, a set of initial observations and a final set of transitions. A minimal and maximal delay is assigned for each transition.

The Unified Modeling Language (UML) [62] is the *de facto* industry standard language for specifying, visualizing, constructing, and documenting the artifacts of a system, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems [24]. Although UML is popular in industrial practice due to its strong expressive ability and several supporting tools, like other informal methods, UML defines the syntax of a given notation rigorously but leaves the notation's semantics defined informally. Therefore, it does not meet the needs of this work since it cannot be used to unambiguously determine if any system behavior is acceptable or not.

2.1.2 Abstract Model Based Methods

The Vienna Development Method (VDM) [22] and Z [40] are two popular abstract model based specification techniques. When specifying a software module in VDM or Z, the required behavior of the module is described by constructing a model of the system and defining the system behavior in terms of this model.

In VDM, for module interface specifications, the syntactic domains consist of module access programs; the semantic domains are usually some well understood data models (e.g., sequence, tuple, set, map, tree) used for denoting the states of the modules and meaning of objects in the syntactic domains; and the interpretation functions map the elements in the syntactic domains (module access programs) into the semantic domains.

Z is a specification language based on the concepts and notations of first-order logic and set theory. Sets are the only data model in Z: all the specifications are written in terms of pre-defined set manipulation notation.

The basic specification unit in Z is the schema, which can be used to describe both static and dynamic aspects of software modules in a style similar to VDM implicit specifications. However, Z specifications are less readable because of the complicated notations, and it provides no notation for real-time description. Since VDM and Z are techniques for specifying software systems, they cannot specify quantities in continuous value range.

2.1.3 Predicate Logic Based Methods

There are a number of other logic based methods using various forms of logical notation and document structure. For example, in Real-Time Logic (RTL) [7, 42] the behavior is described in terms of *events* and *actions*. In Albert II [12, 13] the system is described as a collection of co-operating agents, using a variation of Real-Time Temporal Logic to describe each of them [52]. Although like this work, it is using a notation from the software requirements for the A-7E aircraft [2, 35], it is less readable without using tabular expression.

Temporal logics employ special operators to denote that a condition is true always ($\Box$) or eventually ($\Diamond$). It was first used in specifying reactive systems over time by Pnueli [59]. Ever since, it has been studied extensively as a means of describing the

temporal behavior of computer systems that do not have real-time requirements [20].

Linear-time and Branching-time temporal logics are extended from temporal logics (e.g., PTL [23],UB [10], CTL [20], and CTL* [17]). Linear-time logics are interpreted over linear structures of states, each of which represents an execution sequence of a reactive system. The classical example of linear-time logics is PTL [23]. Branching-time temporal logics, on the other hand, are denoted as a set of states in tree structures. Each tree represents a reactive system and each path of the tree denotes the possible execution sequences in the system. Classical examples of branching-time logics include UB [10], CTL [20], and CTL* [17].

A variety of approaches have been developed for adding the time constraints to temporal logics. Bounded temporal operators is a common way of introducing real-time in the syntax by assigning an upper bound and a lower bound to the specific operator. For example, the bounded operator $\Diamond_{2,4}$ is interpreted as "eventually within 2 to 4 time units". However, the bounded-operator notation is limited within the adjacent temporal contexts. Its shortcoming can be remedied by use of freeze quantification that binds a variable to the corresponding time. The idea was first introduced and analyzed in [6]. As the third method to write real-time requirements, an explicit clock variable is used based on standard first-order temporal logic. A dynamic state variable T is used as a clock variable to describe the values of the corresponding time in each state. For instance, the time-bounded response property can be specified by the formula

$$\forall x \cdot \Box((p \wedge T = x) \rightarrow \Diamond(q \wedge T \leq x + 3))$$

where the global variable x is bound to the time of every state in which p is observed (p and q each represents an event). The time constraint for p is $T = x$. q is restricted with $T \leq x + 3$.

Examples of expressing timing constraints in this method can be found in [60, 19, 27, 61]; it has been studied for its expressiveness and complexity in [5, 28]. The temporal operators do not, however, increase the expressiveness of a logic since "always" and " eventually" can be expressed simply as quantification over time ($\forall t$ and $\exists t$, respectively).

2.1.4 SCR Requirements Method

The "Software Cost Reduction" (SCR) requirements method [32] is a formal method based on tables for the specification and analysis of the behavior of complex systems. The SCR method has been applied to several practical systems such as avionics systems, telephone networks, and safety-critical components of nuclear-power plants and so on. Designed for use by engineers, it was introduced originally in a project at the US Naval Research Laboratory (NRL) to specify the requirements for the operational flight program of the A-7E aircraft [2, 34, 35], and is a forerunner of the approach presented in this thesis. In the SCR method, the behavior of the system is described by a set of mode classes, in which each mode represents a state in the concurrently executing FSA. A condition is a predicate defined on one or more system entities (a system entity is an input or output variables mode, term) at some point in time. An event is the instant when a condition changes value [32].

Several authors have applied and extended SCR for requirements documentation [8, 9, 58]. Also the use of SCR for hybrid systems is discussed in [30]. The effectiveness of this approach is shown in a number of real examples (e.g., [11, 34, 45, 71]) and it is also shown to satisfy some industrial expectations of requirements documentation. D. Peters extends some notations from SCR in [58], deriving a monitor for real-time systems from given system requirement documentation. He gives an interpretation of this notation on behaviors as functions of continuous time, and defines a mode as a set of possible histories instead of a unique state as in the SCR approach. This simplifies the specification and clarifies the system behavior. As a previous work and foundation of this thesis, SCR approach and its applications are discussed in more detail in Chapter 3.

2.2 Interface Specification

There are many techniques in Module Interface Specification, whereas very few researchers have concentrated on the issues of Interface Modules. One of the challenges of Interface Module Specification is to specify variables in both continuous and discrete range. Some research work uses hybrid system specification in formal method [30]; their specifications are for system requirements rather than that for Interface

Modules. In [49], N. Lynch, R. Segala and F. Vaandrager propose a hybrid automaton model that is capable of describing both continuous and discrete behavior in the system. The model, which extends the timed automaton model, allows communication among components using both shared variables and shared actions. Like other requirements documentation techniques, interface module specifications can not be fully expressed with these techniques.

A few of the module interface specification techniques that are most well suited to IMS are listed below.

Several languages have been used in the Interface Specification. Bornea [65] is a language designed for low-level specification of message behaviors using ADL framework [64]. The main problem with Bornea is that it specifies the interface behavior with auxiliary definitions, akin to coming up with an implementation. This blurs any distinction between a specification and an implementation, thus making the specification vulnerable to having faults as is the implementation.

IBDL is provided as a language for interface behavior specification and testing based on the message passing paradigm [70]. In this method, formulas are given to disambiguate termination from abnormal termination of a message using the return values and exception to reflect whether the pre-condition associated with the message is satisfied or not. State changes caused by a message invocation are specified by enumerating subsequent messages that a message invocation enables (and/or) disables, by establishing their pre-conditions. However, like other informal specification techniques, the structured English adopted in ISDL [70] leads to its lack of precisely defined behavioral semantics. Thus it does not meet the requirement of this work since it cannot determine whether any behavior is acceptable or not.

Some other research regarding Interface Specification Languages can be found in [72, 16, 67].

Britton *et al* propose an abstract interface to describe interface modules [14]. Such an abstract interface is an abstraction that represents more than one interface; it consists of the assumptions that are included in all of the interfaces that it represents. The technique includes assumption lists, access function tables and event tables. Although the technique is similar with the format adopted in this work, the abstract interface cannot fully specify the interface modules since it only provides the interface between the user programs and the device interface modules.

The Trace Assertion method [57] is a software module specification method based on the state machine model where the states of the machine are denoted by its history. In this method, each module is considered as providing a number of programs that a user program can invoke. Some programs (O(for operation)-programs) can cause changes in state of the module, and other programs (V(for value)-programs) can give to a user program the values of the variables making up that state.

In [57], Wang and Parnas present the Trace Assertion Method to specify module interfaces with examples of specifying operations on some typical data structure modules — stack, queue, and binary tree. These are all examples for software modules. As discussed in [51], Trace Assertion Method has the ability to specify nondeterministic cases. Janicki and Sekerinski apply the Trace Assertion Method in Module Interface Specification in [44]. However, it is restricted to software-based specification and is not suitable for hybrid systems containing both hardware and software (i.e., it fails to represent variables in the real domain in the interface module). Like Z, Trace Assertion Method does not provide explicit and complete specification of real-time aspects.

2.3 Tabular Expression

Tabular expressions are used in this work to denote the module behaviors. As described in [1], which is based on [43, 53], they are designed for denoting software behaviors. Using traditional mathematical notation to document real software products often generates large and complicated expressions. If such complex documentation is considered unreadable, it will not be used by maintainers and will become less valuable. The wide application of tabular expressions can be found in several industrial projects and research efforts [31, 35, 46].

Tabular notation has been found to be useful for improving the readability of complicated mathematical expressions, and is particularly well-suited to software documentation. The structure provided by tabular notation makes it easier for a person to consider every case separately while writing or reading a design document.

An n-dimensional table contains of n *headers* (denoted as H_1, H_2,..., H_n), and an n-dimensional *main grid*, G. As illustrated in Table 2.1, H_1 is the row header containing the expressions $h_{1,1}$, $h_{1,2}$ and $h_{1,3}$. Also, H_2 contains the expressions $h_{2,1}$

H2

	$h_{2,1}$	$h_{2,2}$	$h_{2,3}$
$h_{1,1}$	$g_{1,1}$	$g_{1,2}$	$g_{1,3}$
$h_{1,2}$	$g_{2,1}$	$g_{2,2}$	$g_{2,3}$
$h_{1,3}$	$g_{3,1}$	$g_{3,2}$	$g_{3,3}$

H1 G

Figure 2.1: Raw Table Skeleton of Table

$f(x, y) \stackrel{df}{=}$

$p_T : H_1 \wedge H_2$ $r_T : G$ Normal	$x < 0$	$x = 0$	$x > 0$
$y < 0$	$-x - y$	$-y$	$x - y$
$y = 0$	$-x$	0	x
$y > 0$	$y - x$	y	$x + y$

Table 2.1: An example of Normal Table

and $h_{2,2}$. G contains the expressions $g_{1,1}$, $g_{1,2}$, $g_{2,1}$, $g_{2,2}$, $g_{3,1}$ and $g_{3,2}$. An *index*, α, is a tuple of length n such that $\forall i, 1 \leq i \leq n \Rightarrow 1 \leq \alpha[i] \leq length(H_i)$, where $\alpha[i]$ represents the element at position i of index α. An index locates a unique cell in each header H and in the main grid G. For example, in Table 2.1, we define $f(x, y) = |x + y|$. The tuple (1, 2) is an index for the table in Table 2.1 and locates the first cell in H_1, $y < 0$, the second cell in H_2, $x = 0$, and $-y$ in G.

In Tabular Expressions, the semantics of each form are described by the *cell connection graph (CCG) case* and two "table rules": the *table predicate rule*, P_T, which defines the domain of the expression, and the *table relation rule*, r_T, which defines the value of the expression.

The cells of headers mentioned in the table predicate rule (H_1 and H_2 in Table 2.1) are called guard cells, combined according to P_T to form a guard expression $P_T[\alpha]$ for a particular α. The cells of main grids in the table relation rule (G in Table 2.1) are

called value cells, forming the value expression according to headers. For example in Table 2.1, for $\alpha = (1,1), P_T[\alpha] = y < 0 \wedge x < 0$. The conjunction of $P_T[\alpha]$ with the value expression for that index, $r_T[\alpha]$, forms a *raw element relation*, R_α, for example $R_{1,1}$ is defined as $\{(x,y,v) \mid (y < 0) \wedge (x < 0) \wedge v = (-x - y)\}$, where v is the value of the function $f(x,y)$.

The cell connection graph determines how the raw element relations are to be combined to construct the table relation, R_T. R_T for Normal Tables, Vector Tables and Decision Tables are defined in Section 2.3.1, Section 2.3.2 and Section 2.3.3 respectively.

2.3.1 Normal Tables

Normal Tables evaluate the value expression for the index that makes the guard expression true. The table relation of Normal Table can be expressed as

$$R_T = \bigcup_{\alpha \in I} R_\alpha$$

where I is the index set of G and $\alpha \in I$.

As an example of interpreting a Normal Table in Table 2.1, expression of P_T : $H_1 \wedge H_2, r_T$: G and CCG case "Normal" specifies that the table is interpreted by choosing i and j such that $H_1[i] \wedge H_2[j]$ is true. The value of the tabular expression is given by $G[i,j]$, giving the table relation,

$R_T = \bigcup_{j=1}^{3} \bigcup_{i=1}^{3} R_{(i,j)} = R_{1,1} \cup R_{2,1} \cup R_{3,1} \cup R_{1,2} \cup R_{2,2} \cup R_{3,2} \cup R_{1,3} \cup R_{2,3} \cup R_{3,3}$

Thus, in Table 2.1, if $x < 0$ and $y > 0$, for example, the value of the tabular expression is $y - x$.

2.3.2 Vector Tables

Vector tables "join" the value expressions for all indices (a row or column) that make the guard expression true. Vector tables are useful when describing a function whose range is a tuple of elements, because one dimension of the table is dedicated to separating the different elements. The relation for Vector tables is

$$R_T = \bigotimes_{i=1}^{n} \left(\bigcup_{\substack{\alpha_v = i \\ \alpha \in I}} R_\alpha \right)$$

$\mathrm{p_T} : H_2$ $\mathrm{r_T} : H_1 = G$ Vector	P	$\neg P$
x_1	a	b
x_2	c	d

Table 2.2: An example of Vector Table

The operator "$\bigotimes$", is a variation of "join" from relational databases, and is used to merge relations to form a single 'vector' relation. For example, if A $\subseteq U_0 \times U_1$ and $B \subseteq U_0 \times U_2$, then

$A \otimes B = \{(x_0, x_1, x_2) | (x_0, x_1) \in A \wedge (x_0, x_2) \in B\}$.

For example, Table 2.2 specifies values for the tuple (x_1, x_2). The first row of the main grid specifies the value of x_1 only, while the second row gives values for x_2 only. In Table 2.2, the table predicate rule is "H_2", therefore the expressions in header H_2 are used to select a column. The table relation rule is "$H_1 = G$", but this rule must be applied to both rows of the table. The equivalent concurrent-assignment statement for the first column of the table is "$x_1, x_2 := a, c$".

The relation of Table 2.2 can be explicitly defined in terms of the table's four raw element relations as

$$
\begin{aligned}
R_T &= (R_{1,1} \otimes R_{1,2}) \cup (R_{2,1} \otimes R_{2,2}) \\
&= (\{(x_1, P) \mid P \wedge (x_1 = a)\} \otimes \{(x_2, P) \mid P \wedge (x_2 = c)\}) \cup \\
&\quad (\{(x_1, P) \mid \neg P \wedge (x_1 = b)\} \otimes \{(x_2, P) \mid \neg P \wedge (x_2 = d)\}) \\
&= \left\{ (x_1, x_2, P) \;\middle|\; \begin{array}{l} (P \wedge (x_1 = a) \wedge (x_2 = c)) \vee \\ (\neg P \wedge (x_1 = b) \wedge (x_2 = d)) \end{array} \right\}
\end{aligned}
$$

2.3.3 Decision Tables

Decision tables are useful when describing a function whose domain is a tuple of elements, and also when the conditional expressions do not follow regular rules. A normal table header often contains conditions on only one element of the domain. In the case of many elements, the table may contain too many dimensions and become unreadable. Therefore, decision tables can be applied to separate the different elements of the domain. For example, Table 2.3 specifies a function whose input is the tuple (x_1, x_2).

The value that x_1 and x_2 may take are given in the main grid G. The first row of G gives possible values of x_1, while the second row gives possible value for x_2. Cells from both rows of Table 2.3 contribute to the selection of the correct column, therefore, the table is interpreted by taking the intersection of raw element relations from each row of the table. The overall relation of the decision table is defined by the union of these aggregate relations.

The relation R_T of decision tables can be expressed as

$$R_T = \bigcup_{\beta \in I^D} \left(\bigcap_{\substack{\alpha_v = 1 \\ \alpha|v=\beta}}^{k} R_\alpha \right)$$

I^D is the index set for the table with the vector header index removed, and $\alpha|\nu$ is the index formed by deleting the νth element from α. ν is the length of the vector header.

To be a standard notation, α can be defined as being an element of a set, which is defined itself in terms of β. So we define $DT(\beta, v, k) = \{(\beta_1, \beta_2, \ldots \beta_{v-1}, j, \beta_v, \ldots \beta_n) \mid 1 \leq j \leq k\}$ (that is, the set of tuples formed by inserting the values from 1 to k at the vth position in β), and thus the expression is

$$R_T = \bigcup_{\beta \in I^D} \left(\bigcap_{\alpha \in DT(\beta,v,k)} R_\alpha \right)$$

where v is the vector header number and k is the length of the vector header.

For example, $R_{1,1}$ can be expressed as $R_{1,1} = \{(y_1, y_2, y_3, y_4, x_1, x_2) \mid y_1 \wedge (x_1 = \mathit{True})\}$. The relation of Table 2.3 can be explicitly defined in terms of the table's eight raw element relations as follows.

$$\begin{aligned} R_T &= (R_{1,1} \cap R_{2,1}) \cup (R_{1,2} \cap R_{2,2}) \cup (R_{1,3} \cap R_{2,3}) \cup (R_{1,4} \cap R_{2,4}) \\ &= \left\{ (y_1, y_2, y_3, y_4, x_1, x_2) \;\middle|\; \begin{array}{l} (y_1 \wedge (x_1 = \mathit{True}) \wedge (x_2 = \mathit{On}))\vee \\ (y_2 \wedge (x_1 = \mathit{True}) \wedge (x_2 = \mathit{Off}))\vee \\ (y_3 \wedge (x_1 = \mathit{False}) \wedge (x_2 = \mathit{On}))\vee \\ (y_4 \wedge (x_1 = \mathit{False}) \wedge (x_2 = \mathit{Off}) \end{array} \right\} \end{aligned}$$

Informally, Table 2.3 is to be read as follows. The table predicate rule is "H_1 = G", therefore the values in header H_1 and the main grid G are used to select a

$\mathrm{p_T} : H_1 = G$ $\mathrm{r_T} : H_2$ Decision	y_1	y_2	y_3	y_4
x_1	*True*	*True*	*False*	*False*
x_2	*On*	*Off*	*On*	*Off*

Table 2.3: An example of Decision Table

column that satisfies this predicate. However, the column must be chosen such that both elements of the tuple (x_1, x_2) are associated with the correct value. The table relation rule is "H_2", therefore the result of the function is the value of the expression in the selected column of header H_2.

Chapter 3

SCR Requirements Documentation Introduction

The notation and terminology used to describe the software requirements in the Software Cost Reduction (SCR) requirements method is introduced in this chapter. Some of the notations in this work are extended from [58], which is based on the SCR requirements method.

3.1 SCR Requirements Documentation

The SCR requirements method was introduced more than two decades ago to specify software requirements for real-time embedded systems concisely [34]. More recently, the method has been extended and applied to system requirements, rather than simple software, to specify the functional requirements (the values that the system assigns to the output) and non-functional requirements (e.g., timing and accuracy) [56].

The SCR requirement method consists of tabular notations, conditions, events, input and output data items, mode classes, and terms [32]. The Four Variable Model of Parnas and Madey [56], which is illustrated in Chapter 1, provides a formal framework for the SCR method. To specify the relations of the Four Variable Model in a practical and concise manner, the SCR method introduces modes, terms, conditions and events.

The SCR requirements method describes the required system behavior as a set of

mode classes, each of which is represented as a finite state machine. Complex systems are defined by several mode classes operating in parallel.

In [58], the SCR method is extended in the definition of *event* and *mode*. Instead of the SCR definition of events as changes in the value of conditions, this method defines the events as instants when one or more conditions change value, together with the status of all conditions at the same time. Thus the need for special conditioned events is avoided and the elimination of "simultaneous events" simplifies the specification of requirements. The extension method defines a mode as an equivalence class of system histories, whereas SCR defines a mode as a state of a finite state machine [58].

In the SCR method, requirements specifications use tabular expressions method — condition tables, event tables, and mode transition tables, to present the required system behavior precisely and concisely. Each table defines a mathematical relation or function. A condition table describes a controlled variable or a term as a function of a mode and a condition; an event table describes a controlled variable or term as a function of a mode and an event. A mode transition table describes how a mode transits to a new mode according to events [32].

The notations expressed below are adopted from SCR as well as [58] and applied to specify Interface Modules.

3.1.1 Identifier Annotations

To make the specification concise, throughout this thesis we use prefix "c", "m","mc" "Md", "Cl", "C" and "p", where their annotations are described in Table 3.1 to help clarify the meaning of identifiers. The *type* of a variable indicates the range of values that may be assigned to that variable.

3.1.2 Conditions

The values of controlled quantities are changed by the system in response to changes in the monitored quantities, e.g., a user pushing a button, or the value of a quantity exceeding some threshold. Such relevant properties of the monitored and controlled quantities can often be succinctly characterized by predicates, called *conditions*, which are Boolean functions of time defined in terms of the monitored and controlled quantities. These conditions can be expressed by using constants, the environmental

Form	Meaning	Example
^{c}x	Controlled variable	^{c}cvarx
^{m}x	Monitored variable	^{m}mvarx
^{mc}x	Monitored and Controlled variable	^{mc}mcvarx
^{Md}x	Mode	^{Md}modex
^{Cl}x	Mode class	^{Cl}mclassx
^{C}x	Constant	^{C}constx
^{p}x	Condition	^{p}condx

Table 3.1: Identifier Annotations

quantities, and functions of them, together with standard relational (e.g., $<, >$) and logic (e.g., $\wedge, \vee, \neg$) operators, and tabular expressions.

For a particular system, we assume a finite set of conditions, $p_1, p_2, \ldots, p_n$. For the purpose of simplifying the specification, we assign the conditions in a fixed order and refer to them simply by their index in that order (i.e., p_2, etc.) [58].

In the specification for a real-time system, time elapsed from the initial state may affect the module behavior. Hence conditions specified in terms of time are often needed. If this is the case, time is a monitored quantity and no special notation is required [58]. In a real-time system, the acceptable behavior of the system not only must be functionally correct, but also must be temporally correct — satisfying some timing constraints [37]. Such real-time constraints are often the important issue in designing safety- or mission-critical real-time systems, for example in aviation and military applications.

The representation of timing constraints in real-time systems can be achieved by the time elapsed since some fixed time prior to the start time of the system — when the system is turned on.

3.1.3 Events and Event Classes

The instants when one or more conditions change value are significant to the behavior of the system, and these instants are referred to as *events*. Formally, an event e, is a pair, (t, c), where $e.t \in \Re$ is a time at which one or more conditions change value and $e.c$ denotes the status (i.e., true, false, becoming true, becoming false — denoted $\mathrm{T}, \mathrm{F}, @\mathrm{T}, @\mathrm{F}$, respectively) of all conditions at $e.t$, as defined in Table 3.2.

e.c[i]	p_i
T	$`p_i(e.t) \wedge p_i(e.t)'$
F	$\neg `p_i(e.t) \wedge \neg p_i(e.t)'$
@T	$\neg p_i(e.t)' \wedge p_i(e.t)'$
@F	$`p_i(e.t) \wedge \neg p_i(e.t)'$

Table 3.2: Event Notation

The notations $`p_i(e.t)$ and $p_i(e.t)'$ are used to denote the value of $p_i(e.t)$ immediately before and after $e.t$, respectively. The notations "@T" and "@F" characterize the event becoming _true_ or _false_, respectively. The notations "T" and "F" denote that the condition is remaining "true" or "false" since there is no event of conditions changing [58].

The type EvSp, which is defined as Real $\times\{\mathrm{T}, \mathrm{F}, @\mathrm{T}, @\mathrm{F}\}^n$, is the *event space* — the set of all possible events relevant to a particular system. A finite set of events Ev $\subset$ EvSp denotes any particular finite behavior of the system operation.

In many cases the description of system behavior can be stated concisely by considering sets of similar changes in conditions. Such sets of events are referred to as *event classes*. An event class, EC, is a subset of the events relevant to the system: EC $\subseteq$ Ev. Note that since all changes at an instant are described in one event, there is no need to consider "simultaneous events" as a special case. Instants, when two or more relevant changes occur at the same time are cases where an event is in two or more event classes [58].

Some simple event class expressions are defined in Table 3.3. The notations "@T", "@F", "T" and "F" are defined in Table 3.2. The notation "t" denotes that the condition is either remaining true (T) or becoming false (@F), and "f" denotes remaining false (F) or becoming true (@T). t$'$ denotes remaining or becoming _true_, and f$'$ denotes remaining or becoming _false_. The notation "$*$" indicates that the system is not affected by the condition. The event argument is omitted from event class expressions, i.e., @T(p_i) is defined as $\{$ e $\in$ Ev $|$ e.c[i] $=$ @T$\}$. The juxtaposition of two or more event class expressions denotes the conjunction of the expressions, e.g., "@T (p_1) WHEN(p_2)" denotes @T (p_1) $\wedge$ WHEN(p_2).

The following standard functions, which are adopted from [58], are used in interface module specifications. Implicitly, all the functions describe a particular behavior

Notation		Event Class Expression
scalar	tabular	
	p_i	
@T(p_i)	@T	$e.c[i] = @T$
@F(p_i)	@F	$e.c[i] = @F$
WHILE(p_i)	T	$e.c[i] = T$
WHILE($\neg p_i$)	F	$e.c[i] = F$
WHEN(p_i)	t	$e.c[i] = T \vee e.c[i] = @F$
WHEN($\neg p_i$)	f	$e.c[i] = F \vee e.c[i] = @T$
	t′	$e.c[i] = T \vee e.c[i] = @T$
	f′	$e.c[i] = F \vee e.c[i] = @F$
CONT(p_i)		$e.c[i] = F \vee e.c[i] = T$
	$*$	<u>*true*</u>
	$\varnothing$	<u>*false*</u>

Table 3.3: Event Class Notation

on the finite period of system operation $[t_i, t_f]$, in which t_i is the system initial time, and t_f refers to the "current" time, i.e., the final point of the behavior being considered. By convention, when referring to the "current" time (i.e., t_f) the time argument can often be omitted.

Definition 3.1 *For an event class, e, and time, t, such that $t_i \leq t \leq t_f$, Prev(e,t) is the set of events in e that occur prior to t, i.e.,*

$$Prev(e,t) \stackrel{\text{df}}{=} \{x \in e | x.t < t\}$$

Definition 3.2 *For an event class, e, and time, t, $t_i \leq t \leq t_f$, Last(e,t) is the time of the latest event from e before t.*

$Last(e,t) \stackrel{\text{df}}{=}$

$Prev(e,t) \neq \varnothing$	$Prev(e,t) = \varnothing$
$max(\{x \mid \exists y \in Prev(e,t), y.t = x\})$	0

Definition 3.3 *For an event class, e, and time, t, $t_i \leq t \leq t_f$, First(e, t) is the time of the earliest event from e before t.*

$First(e,t)$

$\stackrel{\text{df}}{=}$

$Prev(e,t) \neq \varnothing$	$Prev(e,t) = \varnothing$
$min(\{x \mid \exists y \in Prev(e,t), y.t = x\})$	0

Definition 3.4 *For a condition, p_i, and time, t, such that $t_i < t < t_f$, Drtn(p_i, t) is the duration of time that p_i has been continuously true if $p_i(t)$ is true, otherwise, if $p_i(t)$ is false, then Drtn(p_i,t) = 0.*

$Drtn(p_i, t)$

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_1 \wedge H_2$ $\mathrm{r_T} : G$ Normal	$p_i(t)$	$\neg p_i(t)$
$Prev(@\mathrm{T}(p_i), t) \neq \varnothing$	$t - Last(@\mathrm{T}(p_i), t)$	0
$Prev(@\mathrm{T}(p_i), t) = \varnothing$	$t - t_i$	0

Definition 3.5 *For a condition, p_i, and times, t_1 and t_2 such that $t_i \leq t_1 \leq t_2 \leq t_f$, totalDrtn($p_i, t_1, t_2$) is the total amount of time that p_i has been true between t_1 and t_2.*

$totalDrtn(p_i, t_1, t_2)$

$\stackrel{\text{df}}{=} \int_{t_1}^{t_2} \mathrm{onTime}(p_i, t)dt$

where $onTime(p_i, t)$

$\stackrel{\text{df}}{=}$

$\mathrm{p_T} : H_1$ $\mathrm{r_T} : G$ Normal	
$p_i(t)$	1
$\neg p_i(t)$	0

Definition 3.6 *For an event class, e, and time, t, $t_i \leq t \leq t_f$, Since(e,t) is the time elapsed since the latest event e before t.*

$Since(e,t)$

$\stackrel{\text{df}}{=}$

$Prev(e,t) \neq \varnothing$	$Prev(e,t) = \varnothing$
$t \; - \; Last(e, t_e)$	0

3.1.4 Mode and Mode Classes

The behavior of the system can be described as a sequence of events with respect to condition changes since some initial state. Such a sequence of events is denoted as *history* in some time interval, $[t_i, t_f]$, concisely describing the system behavior by giving the value of the relevant conditions at t_i (initial conditions at the initial time) and listing the sequence of events between t_i and t_f [58].

Similarly, the history that is relevant to the behavior of a module can thus be described by the initial conditions and the sequence of events that have occurred since the initial state. It is noted in [34, 35, 58] that it is often the case that many histories are equivalent with respect to their impact on future behavior. Since many histories are the same with respect to current and future behavior, they are grouped together into a *mode*. A set of modes that partition the possible histories — forming an equivalence relation on the set of histories — is known as a *mode class*.

Definition 3.7 *An environmental mode class (or simply mode class) is an equivalence relation on possible histories, MC $\subseteq$ Hist $\times$ Hist, such that, if MC(H_1, H_2), and $\hat{H}_1$ and $\hat{H}_2$ are the extensions of H_1 and H_2 by the same event, then MC($\hat{H}_1, \hat{H}_2$).*

A mode class consists of the set of mode names, $\{{}^{Md}\mathsf{m}_1,\ {}^{Md}\mathsf{m}_2,\ \cdots,\ {}^{Md}\mathsf{m}_\mathsf{k}\}$, where k $\in$ int, and the function, M: Hist $\rightarrow$ $\{{}^{Md}\mathsf{m}_1,\ {}^{Md}\mathsf{m}_2,\ \cdots,\ {}^{Md}\mathsf{m}_\mathsf{k}\}$, mapping each possible history to a mode in the mode class. The mode name is used to represent the characteristic predicate of the mode. For example for a mode ${}^{Md}\mathsf{m}$, and a time t, ${}^{Md}\mathsf{m}(t)$ is a condition that is true if and only if the history on $[t_i, t]$ is in ${}^{Md}\mathsf{m}$. Note that by convention t is implicitly t_f, so this condition is denoted by "${}^{Md}\mathsf{m}$".

A mode transition function specifies the next mode for any combination of current mode and event. In this work, as illustrated in Table 3.4 and Table 3.5, two forms of mode transition table are applied. In Table 3.4, the original mode is ${}^{Md}\mathsf{m}_1$. When the condition ${}^{p}\mathsf{condition}_1$ becomes true, the system moves to the mode ${}^{Md}\mathsf{m}_2$. Table 3.5 moves the status of conditions to the grid cell of the table, which allows more conditions that are relevant to the mode class in the H_2 cell.

In general, a mode is a set of states that are related in the system response to future events. The state describes the status of the system in the specific time, whereas a mode can be referred to a relatively broad description of the system response that

Mode	Event	New mode
$^{Md}\mathsf{m_1}$	@T ($^{p}\mathsf{condition_1}$)	$^{Md}\mathsf{m_2}$
$^{Md}\mathsf{m_2}$	@T ($^{p}\mathsf{condition_2}$)	$^{Md}\mathsf{m_1}$

Table 3.4: Mode Transition Table 1

$p_T : H_1 \wedge G(H_2)$ $r_T : H_3$ Decision	$^{p}\mathsf{condition_1}$	$^{p}\mathsf{condition_2}$	New Mode
$^{Md}\mathsf{m_1}$	@T	*	$^{Md}\mathsf{m_2}$
$^{Md}\mathsf{m_2}$	*	@T	$^{Md}\mathsf{m_1}$

Table 3.5: Mode Transition Table 2

relates to the occurrence of future events. The same change in conditions occurring at different times is different events, thus the corresponding states are different from each other. However they could be in the same mode — we group these states into a mode — a set of states that is equivalent with current or future behavior.

In addition, the difference between modes and states has more to do with other mode classes than events at different times. The SCR work defines "state" to be the values of all variables relevant to the system, and terms like "environmental state" and "system state" to refer to the values of all environmental quantities or system variables, respectively. The term "mode" is used to be distinct from this, but is essentially the same as what is called "state" in many other techniques (e.g., Statecharts). In SCR, the system is in only one state at any given instant, whereas it will be in many modes (one in each class). In Statecharts, for example, the system could be in several states at the same time (if they were nested or parallel). Note, however, that there is no notion of 'nested' modes in SCR as there is for states in StateCharts.

For example, in describing the motion of a robot arm, it is difficult to describe a particular position of the arm when it is moving. Since the arm position is changing with time, the system will contain several distinct states during the period of motion of the robot arm, e.g., a state set of when the arm is in the position of (10, 10), (10.5, 10), (10.6, 10.1), (10.7, 10.2), $\cdots$. Such distinct but analogous states bring complexity and difficulty to the specification. With mode representation method, these states

can be grouped together as a mode in responding to the event that causes the robot arm motion. Thus, when the arm is moving toward to the destination, wherever it is, the system will remain in one mode — $^{Md}\mathsf{movingTo}(\mathsf{x}, \mathsf{y})$. Therefore the problem addressed above could be solved.

Chapter 4

Specifying Interface Modules

In this chapter, we will introduce some extension techniques from the previous work of SCR method [58] and how we apply SCR technique to specify Interface Modules. These extensions are the main contribution of this work for Interface Module Specification.

4.1 SCR Extensions

System requirement documentation focuses on specifying the observable behavior of the system, which is treated as a "black box" and which only interacts with the environment. Since Interface Modules connect not only the environment but also other system modules, SCR method, as a requirement based formal technique, is not suitable for IMS. Therefore some extensions are needed to make it suitable for Interface Module Specification.

In this chapter, we will use a Robot Arm Control System as a basis for explanation. The computer-controlled robot arm can grasp an object and move it to another position. The robot has five motors to grasp or release the "hand" and position the tip. As illustrated in Figure 4.1, the system contains a robot arm, a PC and the interface hardware.

Name	Condition
^{p}low	^{m}Vtip < 0.8V
^{p}float	0.8V $<$ ^{m}Vtip < 2.0V
^{p}high	^{m}Vtip > 2.0V

Table 4.1: Conditions

4.1.1 Using Access Programs as Conditions

In the SCR method, conditions are boolean funtions of the monitored or controlled variables that have some scope of the value and some time period restriction in the real-time systems. Table 4.1 illustrates some examples of condition definitions. Complete condition definitions for the robot example are detailed in Chpater 5.

Such a specification of conditions is suitable for system requirement documentation. However, an IMS needs to specify not only the relationship to the environment, but also the relationship to other modules in the system. For interface modules, we extend [58] by using access programs as conditions — using the access program name and parameters to denote a condition that is true only when the access program is executing.

Access programs are programs that may be called by programs outside of the module to which they belong. A module interacts with other system modules by calling their access programs. They form a "window" to communicate with other modules. Access programs can be called by other modules and used to access these modules. Thus, they form the interface to the other system modules.

For example, if `foo` is an access program of an IM, then `foo(x)` is true if and only if `foo` is executing, and $\texttt{foo}(\texttt{x}) \wedge \texttt{x} < 0$ is true if and only if `foo` is executing and its parameter x was less than 0 when it was called.

The Access Programs table provides syntax descriptions of programs that may be called by modules in the system. In the robot arm control system as illustrated in the Table 4.2, there are four access programs in the interface module.

An event occurs when one or more conditions change value. Thus an event might occur when an access program condition becomes true — the program is called. For the robot example in Table 4.3, $@\mathrm{T}\,(\texttt{moveInitialPos}())$ denotes the event class of instants when the access program `moveInitialPos()` starts to execute. Similarly, $@\mathrm{F}\,(\texttt{moveLinear}(x, y))$ denotes the event class of instants when $\texttt{moveLinear}(x, y)$

Name	**Descriptions**	**Parameter Types**
`moveInitialPos`	move the arm to the initial position ($^{C}\mathsf{X}$, $^{C}\mathsf{Y}$)	
`moveLinear`	move the arm to the destination (x,y)	**float**, **float**
`graspGripper`	grasp the gripper	
`releaseGripper`	loose the gripper	

Table 4.2: Access Programs

finishes.

4.1.2 Public Variables

Access programs are not the only approach for software modules to access the interface module in the system. They can interact with the module by changing the value of the public variables in the module. The public variables form a different interface to the interface module that also can be accessed and modified directly by software modules in the system. An event occurs when the value of a public variable is changed. For example, we could replace the access programs in the robot system with public variables, as illustrated in Table 4.4.

The three variables form a different interface by replacing the access programs `moveLinear`(x, y) and `moveInitialPos`$()$. Software modules directly assign new values to these public variables of the destination for the next movement instead of calling the access programs. The internal design remains the same as before except for the changed interface. In this sort of interface, as shown in Table 4.5, $^{m}\mathsf{x}$, $^{m}\mathsf{y}$, and $^{m}\mathsf{armHeight}$ are public variables that can be accessed directly and changed by programs in other modules (or by the users directly) in the system.

4.1.3 Parameterized modes

Parameterized modes denote a set of modes with one name — one for each element of the Cartesian product of domains of the parameters. For example, $^{Md}\mathsf{movingTo}(x, y)$ denotes the set of modes containing one mode for each element of the domains of x and y — the possible destination positions for the robot arm. Thus $^{Md}\mathsf{movingTo}(0, 0)$ is a mode in this set, as are $^{Md}\mathsf{movingTo}(0, 1.5)$, $^{Md}\mathsf{movingTo}(1, 1)$ and $^{Md}\mathsf{movingTo}(1, 0.1)$.

Modes : ^{Md}uninitialized, ^{Md}movingTo(x, y), ^{Md}stopped

Initial Mode : ^{Md}uninitialized

Transition Relation :

$p_T : H_1 \wedge G(H_2)$ $r_T : H_3$ Decision	moveInitialPos()	^{p}onPosition(x, y)	moveLinear(x, y)	
^{Md}uninitialized	@T	*	*	^{Md}movingTo $({}^{C}\mathsf{X_i}, {}^{C}\mathsf{Y_i})$
^{Md}movingTo(x, y)	*	@T	*	^{Md}stopped
^{Md}stopped	*	*	@T	^{Md}movingTo (x, y)

Maximum Delay : ^{c}RT_MOVING

Table 4.3: Transition Relation ^{Cl}Motion

Public Variable	Description
^{m}x	x coodinate of the arm position
^{m}y	y coodinate of the arm position
^{m}armHeight	height of the arm tip above the surface

Table 4.4: Public Variables

Modes : ^{Md}movingTo(x, y), ^{Md}stopped

Initial Mode : ^{Md}stopped

Transition Relation :

Mode	Event	New mode
^{Md}stopped	@F (^{p}onPosition(^{m}x, ^{m}y))	^{Md}movingTo (^{m}x, ^{m}y)
^{Md}movingTo(x, y)	@T (^{p}onPosition(x, y))	^{Md}stopped

Maximum Delay : ^{c}RESPONSE_TIME_MOVING

Table 4.5: Accessing Public Variables

$p_T : H_1 \wedge G(H_2)$ $r_T : H_3$ Decision	${}^{p}\mathsf{a}$	${}^{p}\mathsf{b}$	new mode
${}^{Md}\mathsf{A}$	@T	*	${}^{Md}\mathsf{B}$
${}^{Md}\mathsf{B}$	*	@T	${}^{Md}\mathsf{A}$

Table 4.6: Transition Relation for Non-parameterized mode class

$p_T : H_1 \wedge G(H_2)$ $r_T : H_3$ Decision	${}^{p}\mathsf{a}$	${}^{p}\mathsf{b}$(x)	new mode
${}^{Md}\mathsf{A}$(x)	@T	*	${}^{Md}\mathsf{B}$
${}^{Md}\mathsf{B}$	*	@T	${}^{Md}\mathsf{A}$(x)

Table 4.7: Transition Relation for Parameterized mode class

The domain of each parameter in the parameterized modes is defined in the specification.

Parameterized modes simplify the specification by providing a set of modes with particular values. As illustrated in Table 4.5, when the values of the monitored variables (${}^{m}\mathsf{x}, {}^{m}\mathsf{y}$) change such that they are different from the actual position, (${}^{c}\mathsf{x}, {}^{c}\mathsf{y}$), the system enters the mode ${}^{Md}\mathsf{movingTo}({}^{m}\mathsf{x}, {}^{m}\mathsf{y})$, indicating that it is moving towards the new destination position. It remains in this mode until the actual position is within some tolerance of the destination, at which time the event @T (${}^{p}\mathsf{onPosition}(x, y)$) occurs and the arm stops. This table thus describes transitions to and from a large set of modes $\left\{{}^{Md}\mathsf{movingTo}(x, y) \mid x \in \mathbf{float}, y \in \mathbf{float}\right\}$ representing each of the possible destination positions for the arm.

However, the parameterized modes would allow a mode class to be infinite (i.e., an infinite domain for a parameter would result in an infinite mode class). The solution is to restrict the parameters to finite domains, which will limit the mode class to finite set of modes. This is a significant restriction. Since specifying a finite type for the parameters in a parameterized mode ensures that the mode class is finite – there is one mode for each possible value of the parameter, so there is a finite number of modes. If, on the other hand, an infinite type were used then the mode class would be infinite.

When discussing parameterized modes, it is necessary to explain the semantics

of decision tables clearly. Consider a decision table as illustrated in Table 4.6. Its semantics can be expressed as

$$((`m = {}^{Md}\mathsf{A} \wedge @\mathrm{T}\,({}^{p}\mathsf{a})) \Rightarrow m' = {}^{Md}\mathsf{B}) \wedge ((`m = {}^{Md}\mathsf{A} \wedge \neg @\mathrm{T}\,({}^{p}\mathsf{a})) \Rightarrow m' = {}^{Md}\mathsf{A}) \wedge ((`m = {}^{Md}\mathsf{B} \wedge @\mathrm{T}\,({}^{p}\mathsf{b})) \Rightarrow m' = {}^{Md}\mathsf{A}) \wedge ((`m = {}^{Md}\mathsf{B} \wedge \neg @\mathrm{T}\,({}^{p}\mathsf{b})) \Rightarrow m' = {}^{Md}\mathsf{B})$$

With parameterized mode and condition, we have Table 4.7. The semantics can be expressed as

$$\forall x \cdot (((`m = {}^{Md}\mathsf{A}(x) \wedge @\mathrm{T}\,({}^{p}\mathsf{a})) \Rightarrow m' = {}^{Md}\mathsf{B}) \wedge ((`m = {}^{Md}\mathsf{A}(x) \wedge \neg @\mathrm{T}\,({}^{p}\mathsf{a})) \Rightarrow m' = {}^{Md}\mathsf{A}(x))) \wedge (`m = {}^{Md}\mathsf{B} \wedge ((\exists y \cdot @\mathrm{T}\,({}^{p}\mathsf{b}(y)) \wedge m' = {}^{Md}\mathsf{A}(y)) \vee (\neg \exists y \cdot @\mathrm{T}\,({}^{p}\mathsf{b}(y)) \wedge m' = {}^{Md}\mathsf{B})))$$

When consider an instant where $`m = {}^{Md}\mathsf{B}$ and $@\mathrm{T}({}^{Md}\mathsf{b}(0))$, while ${}^{p}\mathsf{a}$ remains constant, we can show that $m' = {}^{Md}\mathsf{A}(0)$.

4.1.4 Callback functions in the User Interface

It is often the case that events may not only cause changes within the module, but refer to cooperation with other modules. If a modification on a variable monitored in module M_1 requires to call an access program (e.g., foo()) in another module M_2, such access program (foo()) in the M_2 can be considered to be a controlled condition of module M_1. In another word, change in module M_1 invokes an access program foo() in module M_2; thus foo() is controlled by module M_1 since its status is changed due to M_1. In this work, we treat access programs as conditions. Therefore, foo() is a controlled condition which is controlled by M_1.

Since an interface module encapsulates all of what is necessary to connect the application software to the external world, the boundary between the IM and the application software is that the IM must relate external quantities to software quantities/access programs. In the case where a module is implemented in the environment of a support tool, this support system (i.e., UI environment) itself is the interface module. The callback function is one of the interacting formats between the developing environment and the IM.

The interface of a robot control system is implemented in Java AWT. Here we specify a small, but functionally complete part of java.awt.Component, as an example

of our specification of callback functions.

The behavior of the GUI of the robot control system is that the user can control the motion of the robot arm by dragging the mouse of the computer. When the user clicks and moves the mouse, the robot arm moves simultaneously according to the position of the cursor on the screen of the computer.

The specification given in Section 5.3 is only for the part of the behavior of java.awt.Component that relates to the IM in our example. In this example, we use abstract state variables (**mouseListener** list and **mouseMotionListener** list) and specify how the callback events are related to these variables. The mouseListener list is a sequence of references to **MouseListener** and mouseMotionListener list is a sequence of references to **MouseMotionListener**.

The method mouseDragged() is invoked when the mouse button is pressed on a component and then dragged. It is one of the components in **mouseMotionListener** that provides human-machine interface by Java GUI. The behavior of the `addMouseListener()` and `addMouseMotionListener()` methods is to add the argument to the end of the given sequence. When an external mouse motion event occurs, the appropriate member functions of the objects in the **mouseListener** or **mouseMotionListener** lists are called to implement the interface.

For example, as a member function of each of the objects in the mouseMotionListener list, the mouseDragged() method is invoked when a user drags the mouse.

Technically, we denote the motion of the mouse as a monitored quantity $^{m}\mathsf{mouseMotion}$. The change of $^{m}\mathsf{mouseMotion}$ invokes mouseDragged() as a registered listener object. It is called in response to change of certain environmental conditions. When we use it as a condition, it will be a controlled condition because the interface module will make it true when the appropriate event (i.e., @T ($^{m}\mathsf{mouseMotion}$) or @F ($^{m}\mathsf{mouseMotion}$)) occurs. It may also be a monitored condition, since the interface module may not be able to respond to other events until it becomes false (i.e., the callback has completed).

The IMS states the relationship of callback events and the monitored events by the controlled variable function. Table 4.8 illustrates that the method mouseDragged() is invoked in the mode $^{Md}\mathsf{mouseDragging}$. `l` denotes the ith object in the **mouseMotionListenerList**. The `mouseDragged` or `mouseMoved` method will be invoked on this object as appropriate.

l

$p_T : H_2$ $r_T : H_1 = G$ Vector	^{Md}idle	^{Md}processingEvent(^{C}MouseMoved, i)	^{Md}processingEvent(^{C}MouseDragged, i)
l.mouseDragged(^{C}mouseDragged)	*false*	*true*	*false*
l.mouseMoved(^{C}mouseMoved)	*false*	*false*	*true*

where ^{c}l = $\mathbf{mouseMotionListenerList}_i$ for all i : **int**, $0 \leq i < |\mathbf{mouseMotionListenerList}|$

Table 4.8: Controlled Variable Function

4.2 Interface Modules Specification

This section illustrates how the SCR method is applied in Interface Module Specification. The technique of specifying Interface Modules views the Interface Modules as “systems” in the sense of [58]. We use access programs as conditions and events are triggered when the status of these conditions is changed. Quantities related to the Interface Module interface are represented as monitored quantities and controlled quantities. Module behavior is described in terms of abstract state variables and one or more mode classes.

State variables define the state space in terms of a collection of typed variables [37]. Each variable is declared by providing its name and type. In the IMS, abstract state variables are introduced to specify the module behavior. The word “abstract” is used to distinguish these from “concrete” state variables, which would be used in the module internal design documentation. When it is clear from the context that we are using abstract state variables then it is safe to leave off “abstract”.

4.2.1 Monitored and Controlled Quantities

Since Interface Modules interact with both the environment and the other software modules in the system, quantities related to IM are environmental quantities and software quantities. IMS combines software and environmental variables in one document.

As stated in [58] *environmental quantities* are quantities that are external to the system, “independent of the chosen solution and are apparent to the ‘customer’.” From the point of view of the IM, the ‘customer’ is the designer of the software that will use the IM to communicate with the external environment, and the quantities of interest are both internal (software) and external quantities. The internal quantities are software quantities that form the interface between the IM and other system modules, including, for example, parameters to access programs. The external quantities are the environmental quantities relevant to the system and represent such things as temperature, switch settings, or the position of a robot arm. All these quantities can be represented by functions of time. Note that for real-time systems, time, itself, is a relevant environmental quantity.

The IMS must describe the behavior of the IM in terms of these quantities. It must

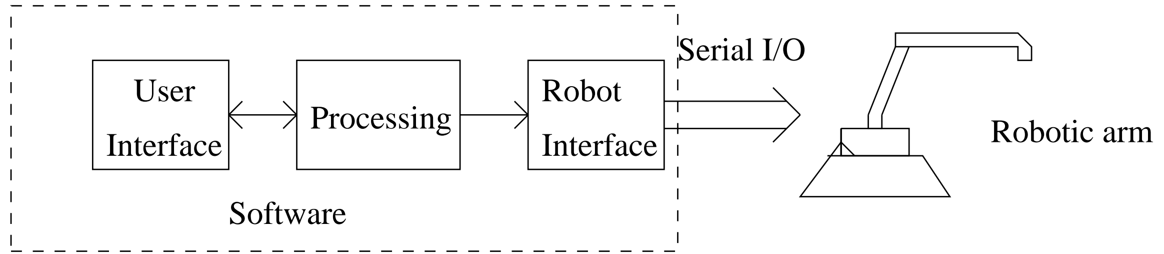

Figure 4.1: Robot system

Variable	**Description**	**Value Set**
$^{m}\mathsf{t}$	current time	**Real**
$^{c}\mathsf{armPos}$	position of the arm tip(x, y) in mm	**Real** $\times$ **Real**
$^{c}\mathsf{armHeight}$	up position of the arm tip in mm	**Real**
$^{c}\mathsf{gripPres}$	pressure applied by the gripper (Pa)	**Real**

Table 4.9: Environmental Quantities

give the value of the controlled quantities depending on the current and past values of the monitored quantities. Consider, for example, a system for making signs that uses a robot arm as illustrated in Figure 4.1. The system consists of three modules (each of which may be further divided into smaller modules): User Interface, Processing, and Robot Interface. The User Interface and Robot Interface are both examples of interface modules, which isolate the processing software from the specific details of the input or output of software and hardware device. If, for example, the mechanical properties of the robot arm were to be modified, it would probably require that the software controlling it also change. The Robot Interface module limits the impact of these changes to the module itself.

The Environmental Quantities table defines the syntax of environmental quantities relative to the system. Monitored and controlled quantities are distinguished by the prefix "m" and "c". Also, time is always a monitored quantity in real-time systems, as shown in Table 4.9.

4.2.2 Mode Classes

For IMS, we take the mode definition to specify how the module behavior depends on previous events, and the controlled state functions can be specified in terms of the

current mode in one or more mode classes. It is often possible that there would be more than one mode class in the IM. If the behavior is specified for every mode in a mode class, then it is fully specified.

For example, as illustrated for the robot control system in Table 4.3 and Table 4.10, there are two mode classes $^{Cl}\mathsf{Motion}$ and $^{Cl}\mathsf{Gripper}$ in the Robot Interface Module. $^{Cl}\mathsf{Motion}$ represents the motion of robot arm — moving the arm to another position, and $^{Cl}\mathsf{Gripper}$ describes the motion of the gripper — grasping or releasing the "hand". The Transition Relation Table gives the next mode for any current mode and event combination. In $^{Cl}\mathsf{Motion}$, the initial mode is $^{Md}\mathsf{uninitialized}$. When the event @T (`moveInitialPos`()) occurs (i.e., the access program `moveInitialPos`() is called), the IM enters the mode $^{Md}\mathsf{movingTo}(^{C}\mathsf{X_i}, ^{C}\mathsf{Y_i})$ — the arm is moving toward its initial position. Similarly, @T (`moveLinear`(x, y)) initiates the movement towards the position (x, y). When the arm reaches its destination, the condition $^{p}\mathsf{onPosition}(x, y)$ becomes <u>*true*</u> (i.e., @T ($^{p}\mathsf{onPosition}(x, y)$)) and the IM enters the mode $^{Md}\mathsf{stopped}$ — it stops at that position.

Mode class $^{Cl}\mathsf{Gripper}$ relates to the opening and closing of the gripper. The mode change is initiated by either @T (`graspGripper`()) or @T (`releaseGripper`()), to close or open the gripper, respectively. The mode changes to $^{Md}\mathsf{grasped}$ or $^{Md}\mathsf{released}$ when the access program returns (@F (`graspGripper`()) or @F (`releaseGripper`())) — indicating that the call will not return until the gripper has completed the operation.

$^{Cl}\mathsf{Motion}$ and $^{Cl}\mathsf{Gripper}$ describe the mode classes in the robotic arm module and each mode in the mode class is clearly specified. If there is any mode in the mode class that is left unspecified, then the specification is incomplete. In other words, if the behavior is specified for every mode in the mode class, then the module is fully specified.

4.2.3 Controlled Value Functions

Since the values of the controlled quantities are changed by the system in response to changes in the monitored quantities, the status for each controlled quantity in each mode in the mode class needs to be specified clearly. The Control Value Function expresses the acceptable values of controlled quantities in terms of the previous

Modes : ^{Md}grasping, ^{Md}grasped, ^{Md}releasing, ^{Md}released

Initial Mode : ^{Md}released

Transition Relation :

Mode	Event	New mode
^{Md}released	@T (graspGripper())	^{Md}grasping
^{Md}grasping	@F (graspGripper())	^{Md}grasped
^{Md}grasped	@T (releaseGripper())	^{Md}releasing
^{Md}releasing	@F (releaseGripper())	^{Md}released

Maximum Delay : ^{c}RT_GRIPPER

Table 4.10: Mode Transition Relation ^{Cl}Gripper

(^{c}armPos)
|

$p_T : H_1$ $r_T : H_2\ G$ Vector	^{c}armPos |
^{Md}uninitialized	$\frac{d}{dt}({}^{c}\text{armPos}) = (0,0)$
^{Md}stopped	$\frac{d}{dt}({}^{c}\text{armPos}) = (0,0)$
^{Md}movingTo(x, y)	$\frac{d}{dt}(\lvert{}^{c}\text{armPos} - (x,y)\rvert) < 0 \wedge$ $Drtn({}^{Md}\text{movingTo}(x,y)) \leq {}^{C}\text{MOVE_TIME}$

Table 4.11: Control Value Function

behavior, current mode in one or more mode classes, abstract states and condition values. To be concise and precise, these quantities are expressed by mathematical expressions.

For example, as illustrated in Table 4.11, the controlled quantity ^{c}armPos is defined in terms of every mode in the mode class ^{Cl}Motion. In the mode ^{Md}uninitialized and ^{Md}stopped, the value is constant, expressed as ${}^{c}\text{armPos} \mid \frac{d}{dt}({}^{c}\text{armPos}) = (0,0)$, where "|" denotes "such that"; while in the mode ^{Md}movingTo(x, y), its value is changing such that the arm is moving closer to the destination (x, y), which is expressed as $\frac{d}{dt}(\lvert{}^{c}\text{armPos} - (x,y)\rvert) < 0$. In addition, the duration of a particular movement must be less than timing constraint ^{C}MOVE_TIME.

4.2.4 Timing Requirements

The mode changes depend on the occurrence of events (e.g., the user pressing a button, receiving a response from the application devices or an access program being called by the software modules). In many cases the maximum duration of the mode transition is important. Specification can be made by giving either a single maximum delay for all transitions as part of the mode class definition, as is done in Table 4.10 or specifying the maximum delay for different transitions in a tabular form. Other formats of timing constraint can be expressed by adopting Definition 3.1, Definition 3.2, $\cdots$, and Definition 3.6 in the Section 3.1.3.

In the system, the event occurs instantly. But it will take a while for the system to respond to that change. The Maximum Delay expresses that the correspondence of the system according to the occurrence of an event is required to be no later than the defined maximum delay time. For example in the $^{Cl}\mathsf{Gripper}$ illustrated in Table 4.10, when condition @T (`graspGripper`()), the system must respond no later than maximum delay time $^{c}\mathsf{RT_GRIPPER}$.

For real-time systems, the amount of time between events may be relevant to the module behavior. This amount of time can be expressed using the functions defined in Section 3.1.2. For example, if a robot arm is required to move the arm to another position within a maximum amount of time $^{C}\mathsf{MOVE_TIME}$, the specification can be expressed as : $Drtn(^{Md}\mathsf{movingTo}(x, y)) \leq {}^{C}\mathsf{MOVE_TIME}$, where $^{Md}\mathsf{movingTo}(\mathsf{x}, \mathsf{y})$ is the mode that the arm is moving to the destination (x, y). $Drtn(^{Md}\mathsf{movingTo}(x, y))$ denotes the amount of time when the mode $^{Md}\mathsf{movingTo}(\mathsf{x}, \mathsf{y})$ is continuously true until the time that it becomes false (when the arm has reached the position (x, y)).

4.2.5 Environmental Constraints

Exclusive conditions and possible simultaneous changes are discussed in [58]. Often some of the conditions used in an SRD are mutually exclusive, i.e., one being *true* implies that the other is *false*.

"Knowledge of possibility and impossibility of simultaneous changes can be essential for checking that all possibilities are addressed in SRD" [21]. Such simultaneous changes do exist in IM, for example the users are not allowed to press "ON" and "OFF" buttons of the machine at the same time; or if they do, there must be some

priority addressed in the IMS for system protection. Besides, due to the mechanical limitations of application devices, IMS must provide these constraints clearly to the implementors. For example in the robot arm system, due to the physical limitation of the robot arm, there is maximum speed at which it can move. Such constraints can be described by constraints on environmental quantities in addition to the range limits. For example, $\left|\frac{d}{dt}({}^{c}\mathsf{armPos})\right| \leq {}^{C}\mathsf{ARM_RATE}$ describes that the rate of tip motion is bounded by the physical capability of robot mechanism.

4.3 Specification of Human-Machine Interface Modules

A human-machine or user interface is any aspect of a system that impacts a user's interaction with that system. The basic function of a user interface however is to provide the user with the available controls, a presentation of the control options and feedback of the actions taken [37].

For a software system, the importance of the user interface specification is to clearly address a set of possible user operations (i.e., clicking the mouse or keyboard to execute the software) and the corresponding access programs with possible feedbacks. For the robot arm example in Section 5.1.1, the user inputs the destination of the robotic arm from the keyboard. Such an interface can be specified by using the access programs as conditions. Section 5.3 presents a different interface that the user can control the motion of the robotic arm by dragging the mouse, and the domain of the arm position is expanded from the x-y plane to three dimension x-y-z. Although the interface is not more accurate with respect to Section 5.1.1, it is much more convenient to the user. As the interface is implemented in Java AWT, a small, but functionally complete part of java.awt.Component is specified as an example of our specification on callback functions.

4.4 Concurrent Applications

The purpose of the modularization for interface modules is to make them simple and practical, so that they can be easily changed if needed. In the concurrent and

distributed systems, the interface modules can be simplified by avoiding concurrent and distributed issues. For example in the banking system, the operations in the banking machine (i.e., the users key in the password) belong to the interface modules. For the individual banking machine, there is no concurrent event occurring in the interface module.

However, since in general, monitored variables can change values at any time, changes that might occur in a very short period of time could be treated as concurrent events. The problem arises — how to specify the case when the monitored variable is changing its value while the access program is executing. Since the value of a monitored variable is monitored and restricted by a condition, an event will occur if the status of the condition changes. The problem is solved by ensuring that the response due to any event is specified for all modes.

As for the whole system, it will be handled by other system software modules. If the user presses two or more numbers at one time, which is a kind of concurrent event, or simultaneous event, as mentioned in [58], we can avoid the confusion in the IM by using environmental constraints in the IMS.

For access programs, we assume that an IM acts as a "monitor" (in the concurrent programming sense). Since a monitor is a module that restricts access such that, for a particular instance, only one thread can be executing any of the access programs at a given instance. Therefore, for a particular instance, only one process can be executing an access program at a time, so the concurrency issues can safely be neglected. For example, in the robot arm motion specification, the destination (${}^{m}\mathtt{x}$, ${}^{m}\mathtt{y}$) of the arm tip are monitored variables. The access program $\mathtt{moveLinear}(\mathtt{x}, \mathtt{y}, \mathtt{true})$ takes the coordinate from the user input and drives the robot arm moving toward it. While the access program $\mathtt{moveLinear}(\mathtt{x}, \mathtt{y}, \mathtt{true})$ is executing, the (${}^{m}\mathtt{x}$, ${}^{m}\mathtt{y}$) will not allowed to change until the execution of the access program ends.

However, concurrency issues can only be avoid as long as no access program blocks waiting for an external event. If an IM were intended to be used in a multi-threaded environment and not restricted in this way, then some extensions are needed to define to denote the possibility that more than one thread could be executing an access program at one time. The same issue exists in the callback function specification when multiple callbacks are allowed to happen in parallel. The research on such issue would be our future work.

4.5 Discussion

In this chapter, we introduce an extension of the System Requirements Documentation technique presented in [58], which is based on the Software Cost Reduction (SCR) method, and illustrates how we apply these techniques to IMS. An IM is specified as a "sub-system" that interacts with both the external environment and other software modules in the system. The interface quantities are modeled as functions of time and the behavior is described in terms of conditions, events and mode classes. This technique facilitates concise and formal description of the module behavior, including tolerance and delay.

In the SCR method [56] and its extension [58], the techniques provided are mainly for specifying system requirements. The technique of interface module specification has not been discussed. As illustrated in Section 4.1, in this work, we apply the SCR method to specify the behavior of interface modules and make contributions in using access programs as conditions to trigger the occurrence of the events, specifying public variables that can be accessed by software modules and defining parameterized modes. Also the monitored and controlled quantities and mode class are adopted in the specification.

In the work of [58], terms or logic expressions are used as conditions to check the status. Such a technique is suitable for system requirement documentation, i.e., system mode will change with the change of the value of some environmental quantities, since the interface of the system is the environment. For the interface module, the interface not only relates to the environment, but also the system software. Using access programs as conditions in IMS, the interface between modules and the system software is clearly specified.

As defined in SCR approach [58, 32], a condition can be both monitored and controlled. Since, in this work we apply the access programs as conditions, the access programs can also be monitored and controlled.

When a condition is both monitored and controlled, it is only controlled in a very limited sense. Since, a monitored and controlled variable must be related to the environment; thus it is monitored by the system. Therefore, in many cases for such a variable, the control of a quantity will be limited by nature. For example, the cruise-control function of a car maintains the speed in a certain scope. So a driver

does not need to control the speed if the car is set to be cruise-controlled. In this case, the speed of the car is a monitored and controlled variable, but clearly the speed and changes in it are limited by the physics of the car, road etc.

In the same way, such environmental limitation also applies to the access program when it is monitored and controlled. A module can make the condition false (i.e., by returning from the call), but cannot normally prevent it from becoming true. Namely, in most cases a module can call its own access programs, so it can make the condition true. However a module cannot normally restrict the condition when the condition will become true (i.e., when another module will call the access program). An exception here would be if the module is a "monitor" (in the concurrent programming sense) and so the condition being true with respect to one thread will constrain it from becoming true with respect to another thread. A module might also be able to prevent a condition from becoming true (e.g., if the module is a monitor and another access program is executing).

For example in the user interface of robot arm system, the access program mouseDragged() is monitored and controlled. The interface module makes it true when the position of physical mouse of a computer is changed (i.e., a user is moving the mouse) — the interface module calls mouseDragged(). Since an interface module is implied as a "monitor", in the multi-threaded environment, one thread calls mouseDagged() when other threads in the interface module are prevented from calling it due to the mutual exclusive characteristics of the "monitor". In such a case, the interface module has little control effect to the access program mouseDragged(). Also, if the interface module is not a "monitor", other modules are allowed to call mouseDagged() simultaneously. The interface module will have weak control effect to make it true in such a case.

Accessing public variables is another kind of interface to the system software. Often, it is called shared variable in software systems. The application of such a technique does not hinder specifying interface modules, but also can be used to specify modules with relationship to software modules, i.e., interface of a software module.

Section 4.1.4 and Section 5.3 show a functionally complete callback functions specification of part of java.awt.Component. The behavior of the GUI of the robot control system is specified in terms of abstract state variables and mode classes. However, the specification does not include the external objects that are referenced by

the module being specified, i.e., mouseEntered() in the mouseListener interface. One possible solution is to treat them as monitored quantities. However, the problem is that it would be difficult to be complete because there is actually a set of each listener interface. Although further improvement is needed (i.e., how to deal with external objects that are referenced by the module being specified), the callbacks interface specification shows that the technique is promising in specifying user interfaces.

Chapter 5 illustrates the interface module specification in two examples — a robot arm control system and an ATM Banking Machine control system. Both are simple applications but contain typical features of interface modules. In practice, the interface modules could be much more complex. In this case, they can be decomposed into sub-modules, each of which can be specified in this approach.

Chapter 5

Sample Applications

This chapter illustrates the applicability of our IMS methods by presenting sample specifications for interface modules such as might be used in two sample applications: a robot arm control system and an automated teller banking machine. Together these systems represent most of the characteristics of interface modules.

5.1 A Robot Arm Control System

The system is a robot arm that can be controlled by a computer to grasp and move objects from one place to another. In the robot arm control system illustrated in Figure 4.1, the system contains a robot arm, a PC and the interface hardware. The robot arm has five motors to position the tip and open or close the "hand", and the motion of the robot arm is controlled by software on the PC via a serial link. There are three modules in the system, both User Interface and Robot Interface are examples of interface modules.

5.1.1 Robot Interface Module Specification

The specification of IM contains access program tables, environmental quantities tables, mode transition tables, control value function tables, condition function definitions, constant tables, and environmental constraints.

Access programs are defined in a table, describing the function and the parameter type of those access programs. The blank cell in the Parameter Types column

indicates that there is no parameter for this access program. For example, the access program `moveInitialPos` drives the arm to the initial position $({}^{C}\mathsf{X_i}, {}^{C}\mathsf{Y_i})$, which is set in the program and cannot be changed by the user. There is no parameter in this program since the coordinate of the original position is set in the program and cannot be changed by other modules.

The quantities relevant to the modules are specified explicitly in the environmental quantity table. In this illustration we will consider the tip position in two dimensions only, ${}^{c}\mathsf{armPos}$ representing the robot arm's position on a drawing surface, and two real valued variables, ${}^{c}\mathsf{armHeight}$, to represent the height of the arm tip, and ${}^{c}\mathsf{gripPres}$ to denote the pressure between the fingers of the gripper.

Access Programs

Name	Descriptions	Parameter Types
`moveInitialPos`	moves the arm to the initial position $({}^{C}\mathsf{X_i}, {}^{C}\mathsf{Y_i})$	
`moveLinear`	moves the arm to the destination (x, y) with arm up/down (*true*/*false*)	**float**, **float**, **Boolean**
`graspGripper`	closes the gripper	
`releaseGripper`	opens the gripper	

Environmental Quantities

Variable	Description	Value Set
${}^{m}\mathsf{t}$	current time	**Real**
${}^{c}\mathsf{armPos}$	position of the arm tip(x, y) from the inside left corner of the drawing surface in mm	**Real** × **Real**
${}^{c}\mathsf{armHeight}$	height of the arm tip above the surface in mm	**Real**
${}^{c}\mathsf{gripPres}$	pressure applied by the gripper (Pa)	**Real**

5.1.2 Mode Class ${}^{Cl}\mathsf{Motion}$

The mode class ${}^{Cl}\mathsf{Motion}$ is comprised of six modes, one of which is parameterized: ${}^{Md}\mathsf{uninitialized}$, ${}^{Md}\mathsf{raising}$, ${}^{Md}\mathsf{lowering}$, ${}^{Md}\mathsf{movingTo}(x : \mathbf{float}, y : \mathbf{float})$, ${}^{Md}\mathsf{holding}$ and ${}^{Md}\mathsf{stopped}$. The mode transition relation table describes the next mode for any combination of current mode and event. Informally, transition table for ${}^{Cl}\mathsf{Motion}$ can

be read as follows. The table predicate rule is "$H_1 \wedge G(H_2)$", therefore the values in header H_1 and the main grid G are used to select a column that satisfies this predicate. Note that $G(H_2)$ in the predicate rule denotes the status of event class in the cell of H_2. For example, in the second column of the first row, $G(H_2)$ denotes @T (`moveInitialPos`()). The table relation rule is "H_3". Therefore the result of the function is the value of the expression in the selected column of header H_3. For example, the value of $^{Md}\mathsf{uninitialized} \wedge$ @T (`moveInitialPos`()) $\wedge\, WHEN(\neg(^{c}\mathsf{armHeight} = {}^{C}\mathsf{HIGHTPOINT}))$ is $^{Md}\mathsf{raising}$. The notation "$*$" indicates that the system is not affected by the condition, as illustrated in Table 3.3. $^{C}\mathsf{RT_MOVING}$, whose value is defined in the Constant Table, restricts the maximum delay time for IM responding to the events in this mode class, as discussed in Section 4.2.4.

Modes : $^{Md}\mathsf{uninitialized}, {}^{Md}\mathsf{raising}, {}^{Md}\mathsf{lowering}, {}^{Md}\mathsf{movingTo}(x : \mathbf{float}, y : \mathbf{float}), {}^{Md}\mathsf{holding}, {}^{Md}\mathsf{stopped}$

Initial Mode : $^{Md}\mathsf{uninitialized}$

Transition Relation :

$p_T : H_1 \wedge G(H_2)$ $r_T : H_3$ Decision	`moveInitialPos()`	^{p}onPosition(x, y)	`moveLinear`$(x, y, \underline{true})$	`moveLinear`$(x, y, \underline{false})$	^{c}armHeight = ^{C}HIGHTPOINT	^{c}armHeight = 0	^{p}inRange(^{c}armPos.x, ^{c}armPos.y)	$Since$(@F(^{p}inRange(^{c}armPos.x, ^{c}armPos.y))) ≥ ^{C}HOLDING_TIME	New Mode
^{Md}uninitialized	@T	*	*	*	f	*	*	*	^{Md}raising
	@T	*	*	*	t	*	*	*	^{Md}movingTo(^{C}X$_i$, ^{C}Y$_i$)
^{Md}raising	t	*	*	*	@T	*	*	*	^{Md}movingTo(^{C}X$_i$, ^{C}Y$_i$)
	f	*	t	f	@T	*	*	*	^{Md}movingTo(x, y)
^{Md}movingTo(x,y)	*	@T	*	*	*	*	T	*	^{Md}stopped
	*	*	*	*	*	*	@F	F	^{Md}holding
^{Md}stopped	*	*	@T	f	f	*	*	*	^{Md}raising
	*	*	@T	f	t	*	*	*	^{Md}movingTo(x, y)
	*	*	f	@T	*	t	*	*	
	*	*	f	@T	*	f	*	*	^{Md}lowering
^{Md}lowering	f	*	f	t	*	@T	*	*	^{Md}movingTo(x, y)
^{Md}holding	@T	*	*	*	f	*	*	F	^{Md}raising
	F	*	@T	*	f	*	*	F	
	F	*	F	@T	*	f	*	F	^{Md}lowering
	@T	*	*	*	t	*	*	F	^{Md}movingTo(^{C}X$_i$, ^{C}Y$_i$)
	F	*	@T	*	t	*	*	F	^{Md}movingTo(x, y)
	F	*	F	@T	*	t	*	F	
	F	*	*	*	*	*	F	@T	^{Md}stopped

Maximum Delay : ^{C}RT_MOVING

5.1.3 Mode Class ^{Cl}Gripper

The mode class ^{Cl}Gripper describes the motion of the robot hand, grasping or releasing the goods. It contains four modes: ^{Md}grasping, ^{Md}grasped, ^{Md}releasing and ^{Md}released. Informally, it can be read as follows. For example, when the event @T (`graspGripper()`) occurs, IM enters to the mode ^{Md}grasping from the mode ^{Md}released. Therefore, the new mode resulting from ^{Md}released $\wedge$ @T (`graspGripper()`) is ^{Md}grasping. The Maximum Delay time ^{C}RT_GRIPPER for the reactions in the ^{Cl}Gripper is defined in the Constant Table.

Modes : ^{Md}grasping, ^{Md}grasped, ^{Md}releasing, ^{Md}released

Initial Mode : ^{Md}released

Transition Relation :

Mode	Event	New mode
^{Md}released	@T (`graspGripper()`)	^{Md}grasping
^{Md}grasping	@F (`graspGripper()`)	^{Md}grasped
^{Md}grasped	@T (`releaseGripper()`)	^{Md}releasing
^{Md}releasing	@F (`releaseGripper()`)	^{Md}released

Maximum Delay : ^{C}RT_GRIPPER

5.1.4 Conditions

The condition ^{p}onPosition verifies whether the arm tip has arrived the destination (x, y). The condition ^{p}inRange checks if the robot arm is out of the border or not. Both conditions are boolean functions. The syntax of the condition is defined firstly, then the semantics of the condition is defined. For example, the syntax of the condition ^{p}onPosition is defined as ^{p}onPosition : **Real** $\times$ **Real** $\rightarrow$ **Boolean**.

$$
\begin{aligned}
&{}^{p}\mathsf{onPosition} : \mathbf{Real} \times \mathbf{Real} \rightarrow \mathbf{Boolean} \\
&{}^{p}\mathsf{onPostion}(x, y) \\
&\qquad \stackrel{\text{df}}{=} |{}^{c}\mathsf{armPos.x} - x| < {}^{C}\epsilon \wedge \\
&\qquad\quad |{}^{c}\mathsf{armPos.y} - y| < {}^{C}\epsilon
\end{aligned}
$$

$^{p}\mathsf{inRange} : \mathbf{Real} \times \mathbf{Real} \rightarrow \mathbf{Boolean}$

$^{p}\mathsf{inRange}(x, y)$

$$\stackrel{\text{df}}{=} x \in [^{C}\mathsf{MIN_X}, ^{C}\mathsf{MAX_X}] \wedge$$
$$y \in [^{C}\mathsf{MIN_Y}, ^{C}\mathsf{MAX_Y}]$$

5.1.5 Controlled Value Functions

Controlled Value Function tables define the functions of controlled quantities in terms of monitored quantities, and every mode in the mode class. For example, in the Controlled Value Function table for $^{c}\mathsf{armPos}$, the robot arm does not move in the modes $^{Md}\mathsf{uninitialized}$, $^{Md}\mathsf{raising}$, $^{Md}\mathsf{lowering}$, $^{Md}\mathsf{holding}$, $^{Md}\mathsf{stopped}$, i.e., the value of $^{c}\mathsf{armPos}$ in these modes is such that $\frac{d}{dt}(^{c}\mathsf{armPos}) = (0, 0)$, i.e., $(H_1 \mid G)$. In the mode $^{Md}\mathsf{movingTo}(x, y)$, $^{c}\mathsf{armPos}$ is changing such that $\frac{d}{dt}(|^{c}\mathsf{armPos} - (x, y)|) < 0$, i.e., the arm is moving toward (x, y). The clause $Drtn(^{Md}\mathsf{movingTo}(x, y)) \leq {}^{C}\mathsf{MOVE_TIME}$ specifies that the time for a particular move must be less than $^{C}\mathsf{MOVE_TIME}$.

$^{c}\mathsf{armPos}$

|

$\mathrm{p_T} : H_1$ $\mathrm{r_T} : H_2 \mid G$ Vector	$^{c}\mathsf{armPos}$
$^{Md}\mathsf{uninitialized}$	$\frac{d}{dt}(^{c}\mathsf{armPos}) = (0, 0)$
$^{Md}\mathsf{stopped}$	
$^{Md}\mathsf{holding}$	
$^{Md}\mathsf{raising}$	
$^{Md}\mathsf{lowering}$	
$^{Md}\mathsf{movingTo}(x, y)$	$\frac{d}{dt}(|^{c}\mathsf{armPos} - (x, y)|) < 0 \wedge$ $Drtn(^{Md}\mathsf{movingTo}(x, y)) \leq {}^{C}\mathsf{MOVE_TIME}$

$^{c}\mathsf{armHeight}$

|

$p_T : H_1$ $r_T : H_2 \mid G$ Vector	${}^{c}\mathsf{armHeight}$
${}^{Md}\mathsf{uninitialized}$	$\frac{d}{dt}({}^{c}\mathsf{armHeight}) = 0$
${}^{Md}\mathsf{stopped}$	
${}^{Md}\mathsf{movingTo}(x, y)$	
${}^{Md}\mathsf{raising}$	$\frac{d}{dt}({}^{c}\mathsf{armHeight}) > 0$
${}^{Md}\mathsf{lowering}$	$\frac{d}{dt}({}^{c}\mathsf{armHeight}) < 0$

${}^{c}\mathsf{gripPres}$

|

$p_T : H_1$ $r_T : H_2 \mid G$ Vector	${}^{c}\mathsf{gripPres}$
${}^{Md}\mathsf{releasing}$	$\frac{d}{dt}({}^{c}\mathsf{gripPres}) < 0 \wedge$ $Drtn({}^{Md}\mathsf{releasing}) \leq {}^{C}\mathsf{RELEASING_TIME}$
${}^{Md}\mathsf{released}$	$\frac{d}{dt}({}^{c}\mathsf{gripPres}) = 0 \wedge {}^{c}\mathsf{gripPres} = 0$
${}^{Md}\mathsf{grasping}$	$\frac{d}{dt}({}^{c}\mathsf{gripPres}) > 0 \wedge$ $Drtn({}^{Md}\mathsf{grasping}) \leq {}^{C}\mathsf{GRASPING_TIME}$
${}^{Md}\mathsf{grasped}$	$\frac{d}{dt}({}^{c}\mathsf{gripPres}) = 0 \wedge {}^{c}\mathsf{gripPres} \geq {}^{C}\mathsf{GRIP_PRES}$

5.1.6 Constants

Constants table lists all the constants with constant name, discription and scope in the specification. For example, the constant ${}^{C}\mathsf{X_i}$ is the home x position in (mm) ranging ranging from -10 to 10. The constant ${}^{C}\mathsf{RT_GRIPPER}$ denotes the maximum delay on the mode class ${}^{Cl}\mathsf{Gripper}$ in (s) with the scope of $(0, 5)$.

Constant	Description	Range
${}^{C}\mathsf{X_i}$	Home x position (mm)	$(-10, 10)$
${}^{C}\mathsf{Y_i}$	Home y position (mm)	$(0, 20)$
${}^{C}\mathsf{HIGHPOINT}$	Up position for arm (mm)	(20)
${}^{C}\epsilon$	Tolerance on positions (mm)	$(0, 2)$
${}^{C}\mathsf{RT_MOVING}$	Maximum delay on mode class ${}^{Cl}\mathsf{Motion}$ (s)	$(5, 10)$
${}^{C}\mathsf{RT_GRIPPER}$	Maximum delay on mode class ${}^{Cl}\mathsf{Gripper}$ (s)	$(0, 5)$
${}^{C}\mathsf{GRIP_PRES}$	Pressure applied by the gripper (Pa)	$(15, 18)$
${}^{C}\mathsf{GRASPING_TIME}$	Maximum grasping duration (s)	$(10, 15)$
${}^{C}\mathsf{RELEASING_TIME}$	Maximum releasing duration (s)	$(10, 15)$
${}^{C}\mathsf{MOVE_TIME}$	Maximum moving duration (s)	$(30, 35)$
${}^{C}\mathsf{HOLDING_TIME}$	Maximum holding duration (s)	$(30, 35)$
${}^{C}\mathsf{ARM_RATE}$	Maximum moving rate (mm/s)	$(5, 8)$
${}^{C}\mathsf{MIN_X}$	Minimum x position	$(-20, 0)$
${}^{C}\mathsf{MAX_X}$	Maximum x position	$(0, 20)$
${}^{C}\mathsf{MIN_Y}$	Minimum y position	$(-20, 0)$
${}^{C}\mathsf{MAX_Y}$	Maximum y position	$(0, 20)$

5.1.7 Environmental Constraints

In the robot arm system, due to the physical limitation of the robot arm, there is maximum speed at which it can move. Such constraints can be described by constraints on environmental quantities in addition to the range limits. For example, $\left|\frac{d}{dt}({}^{c}\mathsf{armPos})\right| \leq {}^{C}\mathsf{ARM_RATE}$ describes that the rate of tip motion is bounded by the physical capability of robot mechanism.

- $\left|\frac{d}{dt}({}^{c}\mathsf{armPos})\right| \leq {}^{C}\mathsf{ARM_RATE}$
  The rate of tip motion is bounded by the physical capability of robot mechanism.

- $0 \leq {}^{c}\mathsf{armHeight} \leq {}^{C}\mathsf{HIGHPOINT}$
  The height position of the arm is greater than 0 and can not exceed the highest point ${}^{C}\mathsf{HIGHPOINT}$.

5.2 Public Variable Interface of Robot Arm Control System

This section illustrates a different kind of interface of the robot arm control system — other system modules access the interface module via the public variables in the interface module. Public variables are variables that are public resources shared by other modules in the system. They form the interface of the module that could be used and modified by the module that they belong to or accessed by other modules in the system. To illustrate this, the access programs in the robot system can be replaced by three public variables: ${}^{m}\mathsf{x}$, ${}^{m}\mathsf{y}$, ${}^{m}\mathsf{z}$ — denoting the position of the robot arm's destination. System software modules can directly assign values to these public variables for the destination of the next movement instead of calling the access programs to make the movement. The tabular expression of specification on this particular interface is illustrated below. This differs from the specification of the interface with access programs in that we add a public variable table to describe each public variable, the access program table is no longer used, the mode transition function of mode ${}^{Cl}\mathsf{Motion}$ and controlled value functions for ${}^{c}\mathsf{armPos}$ and ${}^{c}\mathsf{armHeight}$ are changed as illustrated below. The remaining parts — mode transition function for the mode class ${}^{Cl}\mathsf{Gripper}$, other controlled value functions and constants are the same as the specification for the interface with access programs. Therefore, we only illustrate the different parts, as follows.

5.2.1 Public Variables

${}^{m}\mathsf{x}$, ${}^{m}\mathsf{y}$, and ${}^{m}\mathsf{z}$ are public variables that can be accessed directly and changed by programs in other modules (or by the users directly) in the system.

Name	**Descriptions**	**Parameter Types**
${}^{m}\mathsf{x}$	desired x coordinate	**float**
${}^{m}\mathsf{y}$	desired y coordinate	**float**
${}^{m}\mathsf{z}$	desired height of the arm tip	**float**

5.2.2 Mode Class ^{Cl}Motion

The mode class ^{Cl}Motion is comprised of four modes: ^{Md}stopped, ^{Md}raising, ^{Md}lowering, ^{Md}moving. The mode transition relation table describes the next mode for any combination of current mode and event. The discipline of reading the mode transition table for ^{Cl}Motion is similiar to Section 5.1.2. The table predicate rule is "$H_1 \wedge G(H_2)$", therefore the values in header H_1 and the main grid G are used to select a column that satisfies this predicate.

For example, in the second column of the first row, $G(H_2)$ denotes $\mathrm{t}'({}^{c}\mathsf{armHeight} < {}^{m}\mathsf{z})$. Referring to the event class notation in Table 3.3, it can be expressed as $({}^{c}\mathsf{armHeight} < {}^{m}\mathsf{z}) \vee @\mathrm{T}({}^{c}\mathsf{armHeight} < {}^{m}\mathsf{z})$. The table relation rule is "H_3". Therefore the result of the function is the value of the expression in the selected column of header H_3. For example, the value of ${}^{Md}\mathsf{stopped} \wedge \mathrm{t}'({}^{c}\mathsf{armHeight} < {}^{m}\mathsf{z}) \wedge WHEN(\neg(|{}^{c}\mathsf{armHeight} - {}^{m}\mathsf{z}| < \delta)) \wedge \mathrm{t}'({}^{p}\mathsf{inRange}({}^{m}\mathsf{x}, {}^{m}\mathsf{y}, {}^{m}\mathsf{z}))$ is ^{Md}raising.

The notation "$*$" indicates that the system is not affected by the condition, as illustrated in Table 3.3. ^{C}RT_MOVING, whose value is defined in the Constant Table, restricts the maximum delay time for IM responding to the events in this mode class, as discussed in Section 4.2.4.

Modes : ^{Md}stopped, ^{Md}raising, ^{Md}lowering, ^{Md}moving

Initial Mode : ^{Md}stopped

Transition Relation :

$p_T : H_1 \wedge G(H_2)$ $r_T : H_3$ Decision	^{c}armHeight $<$ ^{m}z	^{c}armHeight $>$ ^{m}z	$\|^c$armHeight $-$ ^{m}z$\| < \delta$	^{p}onPosition(^{m}x, ^{m}y)	^{p}inRange(^{m}x, ^{m}y, ^{m}z)	New Mode
^{Md}stopped	t′	*	F	*	t′	^{Md}raising
	*	t′	F	*	t′	^{Md}lowering
	*	*	T	f′	t′	^{Md}moving
^{Md}raising	*	*	@T	t	*	^{Md}stopped
	*	*	*	*	@F	
	*	*	@T	f	t′	^{Md}moving
^{Md}lowering	*	*	@T	t	*	^{Md}stopped
	*	*	*	*	@F	
	*	*	@T	f	*	^{Md}moving
^{Md}moving	*	*	t′	@T	t′	^{Md}stopped
	*	*	*	*	@F	
	@T	*	F	*	t′	^{Md}raising
	*	@T	F	*	t′	^{Md}lowering

Maximum Delay : ^{c}RESPONSE_TIME_MOVING

5.2.3 Controlled Value Functions

There are two controlled variables to be defined in the controlled value function tables: ^{c}armPos and ^{c}armHeight. For example in the controlled value function table for ^{c}armPos, the robot arm does not move in the mode ^{Md}stopped, ^{Md}raising and ^{Md}lowering. The value of ^{c}armPos in these modes is such that $\frac{d}{dt}(^c\text{armPos}) = 0$. In the mode ^{Md}moving, ^{c}armPos is changing such that

$\frac{d}{dt}(|^c\text{armPos}) - (^m\text{x}, ^m\text{y}, ^m\text{z})|) < 0$, i.e., the arm is moving toward $(^m\text{x}, ^m\text{y}, ^m\text{z})$.

^{c}armPos

|

<table>
<tr><td>$\mathrm{p_T} : H_1$
$\mathrm{r_T} : H_2 \mid G$
Vector</td><td>${}^{c}\mathsf{armPos}$</td></tr>
<tr><td>${}^{Md}\mathsf{stopped}$</td><td rowspan="3">$\frac{d}{dt}({}^{c}\mathsf{armPos}) = 0$</td></tr>
<tr><td>${}^{Md}\mathsf{raising}$</td></tr>
<tr><td>${}^{Md}\mathsf{lowering}$</td></tr>
<tr><td>${}^{Md}\mathsf{moving}$</td><td>$\frac{d}{dt}\left(\lvert {}^{c}\mathsf{armPos} - ({}^{m}\mathsf{x}, {}^{m}\mathsf{y}, {}^{m}\mathsf{z}) \rvert\right) < 0$</td></tr>
</table>

${}^{c}\mathsf{armHeight}$

|

<table>
<tr><td>$\mathrm{p_T} : H_1$
$\mathrm{r_T} : H_2 \mid G$
Vector</td><td>${}^{c}\mathsf{armHeight}$</td></tr>
<tr><td>${}^{Md}\mathsf{stopped}$</td><td rowspan="2">$\frac{d}{dt}({}^{c}\mathsf{armHeight}) = 0$</td></tr>
<tr><td>${}^{Md}\mathsf{moving}$</td></tr>
<tr><td>${}^{Md}\mathsf{raising}$</td><td>$\frac{d}{dt}({}^{c}\mathsf{armHeight}) > 0$</td></tr>
<tr><td>${}^{Md}\mathsf{lowering}$</td><td>$\frac{d}{dt}({}^{c}\mathsf{armHeight}) < 0$</td></tr>
</table>

5.2.4 Conditions

The condition ${}^{p}\mathsf{onPosition}$ is defined to verify whether the robot arm has reached the position (x, y, z). The condition ${}^{p}\mathsf{inRange}$ is to denote whether the robot arm is out of the range. The constants ${}^{C}\mathsf{MIN_X}$, ${}^{C}\mathsf{MAX_X}$, ${}^{C}\mathsf{MIN_Y}$, ${}^{C}\mathsf{MAX_Y}$ and ${}^{C}\mathsf{HIGHPOINT}$ are defined in the constant table in Section 5.1.6.

$${}^{p}\mathsf{onPosition} : \mathbf{Real} \times \mathbf{Real} \times \mathbf{Real} \to \mathbf{Boolean}$$

$$\begin{aligned}
&{}^{p}\mathsf{onPostion}(x, y, z) \\
&\quad \stackrel{\text{df}}{=} \lvert {}^{c}\mathsf{armPos.x} - x \rvert < {}^{C}\epsilon \wedge \\
&\quad\quad \lvert {}^{c}\mathsf{armPos.y} - y \rvert < {}^{C}\epsilon \wedge \\
&\quad\quad \lvert {}^{c}\mathsf{armPos.z} - z \rvert < {}^{C}\epsilon
\end{aligned}$$

$$
\begin{aligned}
&{}^{p}\mathsf{inRange} : \textbf{Real} \times \textbf{Real} \times \textbf{Real} \rightarrow \textbf{Boolean} \\
&{}^{p}\mathsf{inRange}(x, y, z) \\
&\qquad \overset{\text{df}}{=} x \in [{}^{C}\mathsf{MIN_X}, {}^{C}\mathsf{MAX_X}] \wedge \\
&\qquad\quad y \in [{}^{C}\mathsf{MIN_Y}, {}^{C}\mathsf{MAX_Y}] \wedge \\
&\qquad\quad z \in [0, {}^{C}\mathsf{HIGHPOINT}]
\end{aligned}
$$

5.3 Callback Function Specification

As introduced in Section 4.3, a convenient human-machine interface is implemented by allowing the user to control the motion of the robot arm by dragging the mouse. The user can see the motion of the arm while dragging the mouse. Therefore such an interface is easy for the user to use to control the robotic arm. The interface is implemented in Java AWT. As an example of our callback function, the functionally complete part of java.awt.Component specification is illustrated as follows.

Due to the characteristics of the callback functions, we add abstract state variables, state invariant, and assumptions to specify the module behavior. State variables define the state space in terms of a collection of typed variables, as introduced in Section 4.2. In this example, the abstract state variables are **mouseListenerList** and **mouseMotionListenerList**. As illustrated in the access program table of java.awt.Component, the access programs are `addMouseListener` and `addMouseMotionListener`. The parameter type of these two access programs are **MouseListener** and **MouseMotionListener**, respectively. Therefore, for example, when the access program `addMouseListener` is called, a **MouseListener** object will be added to the **mouseListenerList** to be called with mouse events. The status of state variable **mouseListenerList**, whose value is a sequence of **MouseListener**, will change. Since such a state variable does not exist in the real execution, we name it "abstract" to assist the specification. The notation $`l$ and l' are used to denote the value of l immediately before and after the access program execution, respectively.

Access Programs of java.awt.Component

Name	Descriptions	Parameter Types
`addMouseListener`	Adds the specified mouse listener to receive mouse events from this component	**MouseListener**
`addMouseMotionListener`	Adds the specified mouse motion listener to receive mouse motion events from this component	**MouseMotionListener**

State Variables

mouseListenerList : sequence of reference to **MouseListener**
mouseMotionListenerList : sequence of reference to **MouseMotionListener**

State Invariant

none

Assumptions

Initially, **mouseListenerList** $= \oslash$ and **mouseMotionLisenerList** $= \oslash$

Before `addMouseListener()` or `addMouseMotionListener()` is called, mouse events that occurred are ignored since there are no registered listeners.

Access Program Semantics

`addMouseListener`(**MouseListener** l)
$\stackrel{\text{df}}{=}$ **mouseListenerList**$'$ = `**mouseListenerList** + l
`addMouseMotionListener`(**MouseMotionListener** l)
$\stackrel{\text{df}}{=}$ **mouseMotionListenerList**$'$ = `**mouseMotionListenerList** + l

5.3.1 Environmental Quantities

In this illustration we consider the mouse motion in two ways: one is physical mouse motion; the other is the cursor's position desplayed on the screen. The monitored variable ${}^{m}\mathsf{mouseMotion}$ monitors the rate of the physical mouse motion. ${}^{m}\mathsf{mousePos}$ monitors the change of the cursor displaying on the screen.

Variable	**Description**	**Value Set**
${}^{m}\mathsf{t}$	current time	**Real**
${}^{m}\mathsf{mouseMotion}$	the rate of the physical mouse motion in (x,y) mm/s	**Real** $\times$ **Real**
${}^{m}\mathsf{cursorPos}$	position of the cursor (x, y) displaying on the screen in pixel	**int** $\times$ **int**
${}^{m}\mathsf{locationCom}$	the location of the component displayed on the computer screen (x,y) in pixel	**int** $\times$ **int**
${}^{m}\mathsf{height}$	the height of the component in pixel	**int**
${}^{m}\mathsf{width}$	the width of the component in pixel	**int**
${}^{m}\mathsf{pressButton}$	the left button of the mouse	**Boolean**

5.3.2 Mode Class ${}^{Cl}\mathsf{MouseListener}$

The mode class ${}^{Cl}\mathsf{MouseListener}$ comprised of two modes, one of them is parameterized: ${}^{Md}\mathsf{idle}$ and ${}^{Md}\mathsf{processingEvent}(\mathsf{e} : \mathtt{mouseEvent}, \mathsf{i} : \mathbf{int})$. The mode transition table can be read as follows. The system starts from the mode ${}^{Md}\mathsf{idle}$. When the condition @T(${}^{p}\mathsf{inRange}$) occurs, for example, the system moves to the mode ${}^{Md}\mathsf{processingEvent}({}^{\mathsf{C}}\mathsf{mouseEntered}, 0)$, which causes the access program mouseEntered() of the first **mouseListener** to be executed corresponding to the mouse event. As the appropriate method of each registered listener is called in sequence, the system moves to the mode ${}^{Md}\mathsf{processingEvent}({}^{\mathsf{C}}\mathsf{mouseEntered}, \mathsf{i}+1)$ to process the event for each listener in the **mouseListenerList**.

Modes : ${}^{Md}\mathsf{idle}, {}^{Md}\mathsf{processingEvent}(\mathsf{e} : \mathtt{MouseEvent}, \mathsf{i} : \mathbf{int})$

e: a MouseEvent corresponding to the event being processed.

For all $i : \mathbf{int}, 0 \leq i < |\mathbf{mouseListenerList}|$

Initial Mode : ${}^{Md}\mathsf{idle}$

Transition Relation :

$p_T : H_1 \wedge G(H_2)$ $r_T : H_3$ Decision	^{p}inRange	^{m}pressButton $= {}^{C}$ON	$i+1 <$ |**mouseListenerList**|	**mouseListenerList**$_i$.mouseEntered(e)	**mouseListenerList**$_i$.mouseExited(e)	**mouseListenerList**$_i$.mousePressed(e)	**mouseListenerList**$_i$.mouseReleased(e)	New Mode
^{Md}idle	@T	*	*	*	*	*	*	^{Md}processingEvent(^{C}mouseEntered, 0)
	@F	*	*	*	*	*	*	^{Md}processingEvent(^{C}mouseExited, 0)
	*	@T	*	*	*	*	*	^{Md}processingEvent(^{C}mousePressed, 0)
	*	@F	*	*	*	*	*	^{Md}processingEvent(^{C}mouseReleased, 0)
^{Md}processingEvent	*	*	t	@F	*	*	*	^{Md}processingEvent(^{C}mouseEntered, i + 1)
(^{C}mouseEntered, i)	*	*	f	@F	*	*	*	^{Md}idle
^{Md}processingEvent	*	*	t	*	@F	*	*	^{Md}processingEvent(^{C}mouseExited, i + 1)
(^{C}mouseExited, i)	*	*	f	*	@F	*	*	^{Md}idle
^{Md}processingEvent	*	*	t	*	*	@F	*	^{Md}processingEvent(^{C}mousePressed, i + 1)
(^{C}mousePressed, i)	*	*	f	*	*	@F	*	^{Md}idle
^{Md}processingEvent	*	*	t	*	*	*	@F	^{Md}processingEvent(^{C}mouseReleased, i + 1)
(^{C}mouseReleased, i)	*	*	f	*	*	*	@F	^{Md}idle

5.3.3 Mode Class ^{Cl}mouseMotionListener

The mode class ^{Cl}MouseMotionListener comprised of two modes, one of them is parameterized: ^{Md}idle, ^{Md}processingEvent(e : `mouseEvent`, i : **int**). The mode transition table can be read as follows. The system starts from the mode ^{Md}idle. When the condition @T(|^{m}mouseMotion| > 0) occurs and ^{p}inRange is <u>*true*</u>, for example, the system moves to the mode ^{Md}processingEvent(^{C}mouseMoved, 0), which causes the method mouseMoved() of the **mouseMotionListener** to be executed corresponding to the

mouse event. As the appropriate method of each registered listener is called in sequence, the system moves to the mode ${}^{Md}\mathsf{processingEvent}({}^{C}\mathsf{mouseMoved}, \mathsf{i}+1)$ to process the event by the next object in the **mouseMotionListenerList** until all registered listeners have processed the event.

Modes : ${}^{Md}\mathsf{idle}$, ${}^{Md}\mathsf{processingEvent}(\mathsf{e} : \mathtt{MouseEvent}, \mathsf{i} : \mathbf{int})$

e: a MouseEvent corresponding to the event being processed.

For all $i : 0 \leq i < |mouseMotionListenerList|$, $i \in \mathbf{int}$

Initial Mode : ${}^{Md}\mathsf{idle}$

Transition Relation :

$\mathrm{p_T} : H_1 \wedge G(H_2)$ $\mathrm{r_T} : H_3$ Decision	${}^{p}\mathsf{inRange}$	${}^{m}\mathsf{pressButton} = {}^{C}\mathsf{ON}$	$i+1 < \lvert\mathbf{mouseMotionListenerList}\rvert$	$\lvert{}^{m}\mathsf{mouseMotion}\rvert > 0$	$\mathbf{mouseMotionListenerList}_i$.mouseMoved(e)	$\mathbf{mouseMotionListenerList}_i$.mouseDragged(e)	New Mode
${}^{Md}\mathsf{idle}$	t	f′	*	t′	*	*	${}^{Md}\mathsf{processingEvent}({}^{C}\mathsf{mouseMoved}, 0)$
	t	t′	*	t′	*	*	${}^{Md}\mathsf{processingEvent}({}^{C}\mathsf{mouseDragged}, 0)$
${}^{Md}\mathsf{processingEvent}$	*	*	t	*	@F	*	${}^{Md}\mathsf{processingEvent}({}^{C}\mathsf{mouseMoved}, \mathsf{i}+1)$
$({}^{C}\mathsf{mouseMoved}$, i)	*	*	f	*	@F	*	${}^{Md}\mathsf{idle}$
${}^{Md}\mathsf{processingEvent}$	*	*	t	*	*	@F	${}^{Md}\mathsf{processingEvent}({}^{C}\mathsf{mouseDragged}, \mathsf{i}+1)$
$({}^{C}\mathsf{mouseDragged}$, i)	*	*	f	*	*	@F	${}^{Md}\mathsf{idle}$

5.3.4 Controlled Value Functions

In the following controlled value functions, "l" represents the ith object in the mouseLisenerList, where $i \in$ **int**. The value of the **mouseLisenerList** is changed by the IM, so it is a controlled variable. In order to fully specify the behavior of **mouseLisener**, we list all the possible **mouseLisener**'s access programs that are related to the mouse motion in the controlled value function table. For example, ^{c}l.mouseEntered(^{C}mouseEntered) denotes the access program `mouseEntered`() to be called in response to the mouseEntered event.

^{c}l

$p_T : H_2$ $r_T : H_1 = G$ Vector	^{Md}idle	^{Md}processingEvent(^{C}MouseEntered, i)	^{Md}processingEvent(^{C}MouseExited, i)	^{Md}processingEvent(^{C}MousePressed, i)	^{Md}processingEvent(^{C}MouseReleased, i)
^{c}l.mouseEntered(^{C}MouseEntered)	*false*	*true*	*false*	*false*	*false*
^{c}l.mouseExited(^{C}MouseExited)	*false*	*false*	*true*	*false*	*false*
^{c}l.mousePressed(^{C}MousePressed)	*false*	*false*	*false*	*true*	*false*
^{c}l.mouseReleased(^{C}MouseReleased)	*false*	*false*	*false*	*false*	*true*

where ^{c}l = **mouseListenerList**$_i$, for all i : $0 \leq i <$ |**mouseListenerList**|, i $\in$ **int**

^{c}l

$\mathrm{p_T}: H_2$ $\mathrm{r_T}: H_1 = G$ Vector	^{Md}idle	^{Md}processingEvent(^{C}MouseMoved, i)	^{Md}processingEvent(^{C}MouseDragged, i)
^{c}l.mouseMoved(^{C}MouseMoved)	*false*	*true*	*false*
^{c}l.mouseDragged(^{C}mouseDragged)	*false*	*false*	*true*

where ^{c}l = **mouseMotionListenerList**$_i$, for all i : $0 \leq i <$ |**mouseMotionListenerList**|, i $\in$ **int**

5.3.5 Conditions

The condition ^{p}inRange represents whether the cursor position displyed on the computer screen is out of the range of the component. The monitored variables ^{m}width and ^{m}height denote the width and height of the component.

$$^{p}\mathsf{inRange} : \mathbf{Real} \times \mathbf{Real} \rightarrow \mathbf{Boolean}$$

$$^{p}\mathsf{inRange}(x, y)$$

$$\stackrel{\mathrm{df}}{=} {}^{m}\mathsf{cursorPos.x} \in [{}^{m}\mathsf{locationCom.x}, {}^{m}\mathsf{locationCom.x} + {}^{m}\mathsf{width}] \wedge$$
$$^{m}\mathsf{cursorPos.y} \in [{}^{m}\mathsf{locationCom.y}, {}^{m}\mathsf{locationCom.y} + {}^{m}\mathsf{height}]$$

5.3.6 Dictionary

The access programs of MouseListener interface and MouseMotionListener interface are defined as follows.

Access programs in the MouseListener Interface

Name	Descriptions	Parameter Types
`mouseEntered`	Invoked when the mouse enters a component	MouseEvent
`mouseExited`	Invoked when the mouse exits a component	MouseEvent
`mousePressed`	Invoked when a mouse button has been pressed on a component	MouseEvent
`mouseReleased`	Invoked when a mouse button has been released on a component	MouseEvent
`mouseClicked`	Invoked when the mouse button has been clicked (pressed and released) on a component	MouseEvent

Access programs in the MouseMotionListener Interface

Name	Descriptions	Parameter Types
`mouseMoved`	Invoked when the mouse has been moved on a component (with no buttons down)	MouseEvent
`mouseDragged`	Invoked when a mouse button is pressed on a component and then dragged	MouseEvent

5.4 Automated Teller Machine

An Automated Teller Machine (ATM), is an electronic device that allows bank customers to make cash withdrawals and check their account balances at any time without the need for a human teller. ATMs are activated by inserting a client card that contains the user's account number on a magnetic stripe. The ATM calls up the bank's computers to verify the PIN number. It can accept deposits, dispense cash or make bill payments according to the customer's request and then transmit a completed transaction notice.

Here, as an example of IMS application, we specify some of the interface modules that could be used in an ATM. In the banking system, the operations in the banking machine (i.e., the users key in the password) belong to the interface modules. For the individual banking machine, there is no concurrent event occurring in the interface

module. The interface modules that are relevant to an ATM are a Card Reader and a keyboardAdaptor. It is clear that there are other modules that would also be relevant for a real ATM, e.g., screen display, cash dispenser.

5.4.1 Card Reader

A Card Reader accepts cards from banking customers, validates the cards by reading the card number and confiscates the cards if the system finds any potential violation (e.g., false pretenses). When the system is powered, the Card Reader is initialized and it is ready to accept the card from the customer. If a card is inserted, the Card Reader reads the card number and sends it for authorization. The module accepts the commands from the system software to eject or confiscate the card. We assume the ATM is kept on running, except for technical checking, failure or power shut off. So the Card Reader module starts from the mode ${}^{Md}\mathsf{Ready}$. When a card is confiscated, the transaction is over right away. The ATM turns to be ready to serve another customer. In this case, the ATM returns to the mode ${}^{Md}\mathsf{Ready}$.

Access Programs

Name	Description	Parameter Type
`cardEject`	Causes the card to be ejected	
`cardConfiscate`	Causes the card to be confiscated	
`inforRetreive`	Acquires the card information to determine whether the card is valid	

Environmental Variables

Name	Description	Value Set
^{m}t	time	Real
^{m}cardInSlot	True if a card is inserted on the input slot	Boolean
^{m}cardInReader	True if the card is in the card reader slot	Boolean
^{c}graspCard	State of card grasping mechanism. When true the mechanism operates so as to move the card from the input slot to the reader position so that it is ready to be read	Boolean
^{c}readCard	State of card reading mechanism. When true the mechanism operates so as to read the card information	Boolean
^{c}confiscateCard	State of card confiscating mechanism. When true the mechanism operates so as to confiscate the card from the reader position to the confiscating box	Boolean
^{c}ejectCard	State of card ejecting mechanism. When true the mechanism operates so as to move the card from the the reader position to the input slot so that it is ready to be taken by the user when the transaction is finished	Boolean

Mode Class ^{Cl}cardReading

Modes : ^{Md}Ready, ^{Md}Grasping, ^{Md}cardReady, ^{Md}Reading, ^{Md}Confiscating, ^{Md}Ejecting

Initial Mode : ^{Md}Ready

Transition Relation :

$p_T : H_1 \wedge G(H_2)$ $r_T : H_3$ Decision	^{m}cardInSlot	^{m}cardInReader	inforRetreive	cardEject	cardConfiscate	$Since(@T(^{Md}\text{Confiscating})) \geq {}^{C}\text{CONFISCATING_TIME}$	New Mode
^{Md}Ready	@T	∗	∗	∗	∗	∗	^{Md}Grasping
^{Md}Grasping	t	@T	∗	∗	∗	∗	^{Md}cardReady
^{Md}cardReady	∗	t	∗	∗	@T	∗	^{Md}Confiscating
	∗	t	∗	@T	∗	∗	^{Md}Ejecting
	∗	t	@T	∗	∗	∗	^{Md}Reading
^{Md}Reading	∗	t	@F	∗	∗	∗	^{Md}cardReady
^{Md}Ejecting	@F	∗	∗	∗	∗	∗	^{Md}Ready
^{Md}Confiscating	f	f	∗	∗	t	@T	^{Md}Ready

Controlled Value Functions

^{c}graspCard

$p_T : H_1$ $r_T : H_2 \mid G$ Vector	^{c}graspCard
^{Md}Ready	$\underline{\underline{false}}$
^{Md}Grasping $\wedge$ $Since(@T\,(^{m}\text{cardInSlot})) \leq {}^{C}\text{GRASPING_TIME}$	$\underline{\underline{true}}$
^{Md}Grasping $\wedge$ $Since(@T\,(^{m}\text{cardInSlot})) > {}^{C}\text{GRASPING_TIME}$	$\underline{\underline{false}}$
^{Md}cardReady	$\underline{\underline{false}}$
^{Md}Reading	$\underline{\underline{false}}$
^{Md}Confiscating	$\underline{\underline{false}}$
^{Md}Ejecting	$\underline{\underline{false}}$

^{c}readCard

$p_T : H_1$ $r_T : H_2 \mid G$ Vector	^{c}readCard
^{Md}Ready	$\underline{\underline{false}}$
^{Md}Grasping	$\underline{\underline{false}}$
^{Md}Reading $\wedge$ $Since(@T\,(^{m}\text{cardInReader})) > {}^{C}\text{SCANTIME}$	$\underline{\underline{true}}$
^{Md}Reading $\wedge$ $Since(@T\,(^{m}\text{cardInReader})) \leq {}^{C}\text{SCANTIME}$	$\underline{\underline{false}}$
^{Md}cardReady	$\underline{\underline{true}}$
^{Md}Confiscating	$\underline{\underline{false}}$
^{Md}Ejecting	$\underline{\underline{false}}$

^{c}confiscateCard

$\stackrel{\text{df}}{=}$ ^{Md}Confiscating

i.e., ^{c}confiscateCard is $\underline{true}$ if and only if the card reader is in ^{Md}Confiscating.

^{c}ejectCard

$p_T : H_1$ $r_T : H_2 \mid G$ Vector	^{c}ejectCard
^{Md}Ready	$\underline{false}$
^{Md}Grasping	$\underline{false}$
^{Md}cardReady	$\underline{false}$
^{Md}Reading	$\underline{false}$
^{Md}Confiscating	$\underline{false}$
$^{Md}\mathsf{Ejecting} \wedge Since(@\mathrm{T}\left(^{Md}\mathsf{Ejecting}\right)) < {}^{C}\mathsf{EJECTING_TIME}$	$\underline{true}$
$^{Md}\mathsf{Ejecting} \wedge Since(@\mathrm{T}\left(^{Md}\mathsf{Ejecting}\right)) \geq {}^{C}\mathsf{EJECTING_TIME}$	$\underline{false}$

Constants

Name	Description	Range
^{C}GRASPING_TIME	Maximal allowed grasping time (s)	$(0, 5)$
^{C}SCANTIME	Maximal allowed scan time (s)	$(0, 20)$
^{C}CONFISCATING_TIME	Maximal allowed confiscate time (s)	$(0, 5)$
^{C}EJECTING_TIME	Maximal allowed ejectign time (s)	$(0, 5)$

5.4.2 KeyboardAdaptor

The KeyboardAdapter collects the customer's input, e.g., password number or transaction requests. When a button is pressed, the corresponding event is generated to be processed by the ATM system. The application (ATM) software acquires the pressed key by calling the access program `read`, which returns a string in the keyList. keyList is a list of string which represents the key pressed by the customer. The string of keys in the list is in the same order as the user input. The software empties the keyList by calling `clear`. We assume that when a key is pressed, the other keys on the keyboard are locked before that key is released. For example, if "1" is pressed, the ATM software will not respond any other keys until "1" is released.

We are using tabular expression to describe the change of **keyList** when the access programs are called. The notation $`l$ and l' are used to denote the value of l immediately before and after the access program execution, respectively. *head*(**keyList**) and

tail(**keyList**) represent the first element in the sequence **keyList** and the sequence with the head removed, respectively.

In the behavior section for $^{m}\mathsf{keyPressed}(\mathsf{x} : \textbf{String})$, we are using another form of notation here in that we are treating a monitored variable as if it were an access program. This is a reasonable extension, since the change of $^{m}\mathsf{keyPressed}(\mathsf{x} : \textbf{String})$ will cause the change of **keyList**.

Access Programs

Name	Descriptions	Parameter Type
`keyRead`	Reads a key from the keyList	String
`clear`	Empties the keyList	

State Variables

keyList : sequence of **String**

State Invariant

none

Environmental Quantities

Variables	Description	Value Set
$^{m}\mathsf{keyPressed}(x : \textbf{String})$	True if and only if the key labeled with x is pressed	Boolean

Behavior

`clear()`

$\mathrm{p_T} : H_1$ $\mathrm{r_T} : H_2\ G$ Vector	<u>*true*</u>
keyList$'$ =	$\oslash$

`keyRead()`

$p_T : H_1$ $r_T : H_2\ G$ Vector	$\underline{true}$
keyList$'$ =	*tail*(`**keyList**)
value =	*head*(`**keyList**)

^{m}keyPressed(x)

$p_T : H_1$ $r_T : H_2\ G$ Vector	$\underline{true}$
keyList$'$ =	`**keyList**.x

5.5 Discussion

The technique is used to specify the interface modules of two systems — a robot arm control system and an Automated Teller Machine (ATM). Although the IMS of two systems are subtly different, the skeleton of the specification remains the same — specifying the interface module behavior in terms of mode classes, events, conditions, and terms.

In the IMS of the robot arm control system, the specification is composed of Access Program table, Environmental Quantities table, Mode Transition Relation table, Conditions function, Controlled Value Functions table, Constants table and Environmental Constraint functions. When describing callback functions in Section 5.3 and KeboardAdaptor in Section 5.4.2, we introduce state variables to assist the specification, together with assumptions and access program semantics when necessary. The specification is written using tabular expressions, which makes it more easily understood. For example, we use mode transition relation table to describe the behavior of the modes in the mode class. We also use direct definition to define modes directly in the case of where the current mode is a simple function of recent events, as illustrated in Section 5.3.4. Such a various expression of specification illustrates the wide flexibility of the technique.

In Section 5.4.2, we use a monitored variable as if it were an access program. Since

the change of parameter variables relates to the change of the monitored variable, we treat the monitored variable as an access program. This is an extension from SCR method.

Chapter 6

Conclusions

This work has demonstrated that practical interface modules can be clearly specified in a notation that is concise, precise and readable. We have presented examples for two applications — a robot arm control system and an ATM banking machine to show how the SCR requirements model can be extended to specify Interface Modules.

6.1 Contributions

The main contributions of this work are as follows.

- It extends the SCR method for use in module specification.
  - It introduces the use of access programs as conditions.
  - It identifies and discusses public variables as another approach to accessing the interface modules from system software modules.
  - It introduces the use of parameterized modes to specify a set of modes by parameter values.
- It describes the callback functions in the user interface.
- It applies these techniques in specifying interface modules.

6.2 Applicability of This Work

The specification techniques presented in Chapter 4 are suitable for systems with a combination of software and hardware components, such as embedded systems. The interface modules of these systems interact with both the environment and the system software modules that access interface modules by calling the access programs or changing the public variables. Access programs and public variables form the interface between the interface modules and system software modules. Such systems are widely used in process control and industrial automation applications, where they are often safety critical.

Interface modules, such as described in this work, are particular modules that are examples of hybrid systems. By specifying the interface by access programs, public variables, and callbacks, this application provides a complete description of the module interface. As discussed in Section 4.5, the technique can also apply in software modules specifications.

Interface module specifications, as described in this work, can provide the complete, unambiguous module behavior to the developers. Also, combined with other module specifications, interface module specifications can be used to analyze and verify that the design satisfies the system requirements. Thus, faults relative to interface modules can be found as well some faults in the system in the early stages. The reliability is increased and the cost of maintenance for the project can be reduced by well-specified documentation.

6.3 Limitations of the Method

The techniques in this work are limited to specifying interface modules in a single threaded environment. As we discussed in Section 4.4 and Section 4.5, the concurrency issue is neglected by assuming that an IM acts as a “monitor”. Only one thread can be executing any of the access programs at a given instance. However, concurrency issues can only be avoided as long as no access program blocks waiting for an external event. If an IM were intended to be used in a multi-threaded environment and not restricted in this way, some other way is needed to denote the possibility that more than one thread could be exceting an access program at one time. The

enhancement of the concurrent application of the technique would be our future work.

The ability to specify the user interface is limited. As a part of the IM, the user interface has not been fully specified. In the examples in chapter 5, the user interface is specified in Section 5.1.1 and Section 5.3. In Section 5.3, a small, but functionally complete part of java.awt.Component is specified as an example of our specification on callback functions.

6.4 Future Work

The most significant weakness of the technique is that it does not provide the complete solution for the concurrency issues. As discussed in Chapter 4, we assume that the IM acts as a "monitor", so only one process can be executing an access program at a time. Thus the concurrency issues can be neglected in this way. If an IM were not restricted in this way, and it were applied in the multi-threaded environment, then some extensions to the techniques may be needed to deal with the possibility that more than one thread could be executing an access program at one time.

Some other further investigations could focus on:

- Developing a new tool to analyze modules. As a part of system documentation, an interface module specification needs to be analyzed to check if it meets the system requirement. The evaluation of generalized tabular expressions in software documentation is discussed in [1] and the table evaluation algorithms have been developed. Although the technique in this work is SCR-style based, the technique is different from the original SCR method in some notations (i.e., modes and mode classes). Thus the developed tool-set for specifying and analyzing requirements documents is based on the NRL version of the SCR approach [31] and such tools can not be directly applied in this work. Therefore, a new tool is needed to realize the analysis of modules.

- Specifying the interface modules for software systems. The techniques used in this work provide interface module specifications for real-time systems by specifying the module interface. They express module interface by using access programs as conditions and public variables as another access method. They

are most suitable to systems with interfaces that interact with both the environment and system software modules, i.e., embedded systems. To get more general applications, further application of the interface module specification will focus on the specification of broader interface modules (e.g., human-machine interface) that will further illustrate the usefulness and may allow us to draw more conclusions on specifying real-time systems.

- Application in the code checking. The demand of code checking has risen in recent years. However, the explicit and efficient solution is still on the way, leaving it a tough task. M. Chechik and J. Gannon developed a tentative approach to fill the blank of automatic verification of requirements implementation [15]. To show that an implementation is consistent with its requirement, the appearance of the events that cause state changes is checked in both requirement and the implementation correspondingly. A tool is built to verify the implementation (source code) with tokens (or annotations) inserted to track the state changes so that the state changes can be recognized if they occur. Referring to this approach, the technique of specification could be applied in checking the mode changes. Therefore the practicality and effectiveness of the specification would be improved.

- Generating a test oracle for module behavior checking. The module specification could be used to generate an oracle to verify if the module behavior is consistent with the specification. In [58], D. Peters developed a generator of monitors to observe the consistency of the behavior of the target system versus the system requirement. As a part of system design, a module could be viewed as a target system; and thus its behavior could be controlled and inspected by an oracle.

6.5 Conclusions

This work has provided a technique of Interface Module Specification, an area that attracted little attention. The technique presented in this work is an extension of the System Requirements Documentation technique presented in [58], which is based on the SCR method. An IM is specified as a “sub-system” that interacts with both the external environment and other software modules in the larger system. The IM

specification provides for both continuous and discrete quantities in hybrid systems. The interface quantities are modeled as functions of time and the behavior is described in terms of conditions, events and mode classes. This technique facilitates concise and formal description of the module behavior, including tolerance and delays.

Interface Modules are modules that encapsulate input or output device hardware and the related software, so that the application software can be written without specific knowledge of the particular devices used. Replacing or modifying an interface device will only lead to changes in the IM, rather than changing the other modules in the whole system. In real-time and embedded systems, an IM will often relate real-valued external quantities (e.g., time, positions in space) with discrete valued software quantities. An IM specification must therefore use a combination of notations and formalism.

Based on features of the IM, the technique in this work specifies the module interface of IM by using access programs as conditions and public variables as another method for system software modules to access the IM. Parameterized modes are introduced to specify the IM. In addition, the solution includes specifying callback functions in the user interface. These three factors form the interface of IM as discussed in Chapter 4 and illustrated in Chapter 5 in the example applications. The use of events and mode classes provides a foundation for concise descriptions of the required behavior. Since events are instants, we can express real-time aspects of the behavior using simple constraints on the time elapsed between events. Application of these techniques to the specification of other interface modules will further illustrate their usefulness and may allow us to draw more conclusions on specifying real-time systems.

Bibliography

[1] Ruth F. Abraham, *Evaluating Generalized Tabular Expressions in Software Documentation*, M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, February 1997, Also printed as 346.

[2] T. A. Alspaugh, S. R. Faulk, K. Heninger Britton, R. Alan Parker, D. L. Parnas, and J. E. Shore, *Software Requirements for the A-7E Aircraft*, Tech. Report NRL/FR/5546-92-9194, Naval Research Lab., Washington DC, 1992.

[3] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. H. Ho, *Hybrid Automata: an Algorithmic Approach to the Specification and Verification of Hybrid Systems*, Hybrid Systems (R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, eds.), Lecture Notes in Computer Science, no. 736, Springer-Verlag, October 1993, pp. 209–229.

[4] R. Alur and D.L. Dill, *Automata for Modeling Real-time Systems*, ICALP90: Automata, Languages, and Programming, Lecture Notes in Computer Science 443 (1990), 322–335.

[5] R. Alur and T. A. Henzinger, *Real-time Logics: Complexity and Expressiveness*, Information and Computation **104** (1993), no. 1, 35–77.

[6] R. Alur and T.A. Henzinger, *A Really Temporal Logic*, In Proceedings of the 30th Annual Symposium on Foundations of Computer Science (1989), 164–169.

[7] James Armstrong and Leonor Barroca, *Specification and Verification of Reactive System Behavior: The Railroad Crossing Example*, Real-Time Systems **10** (1996), 143–178.

[8] Joanne M. Atlee and M. A. Buckley, *Logic-Model Semantics for SCR Software Requirements*, Proc. Int'l Symp. Software Testing and Analysis (ISSTA '96), ACM SIGSOFT Software Engineering Notes, vol. 21, no. 3, May 1996, pp. 280–292.

[9] Joanne M. Atlee and John Gannon, *Analyzing Timing Requirements*, Proc. Int'l Symp. Software Testing and Analysis (ISSTA '93), ACM SIGSOFT Software Engeering Notes, vol. 18, no. 3, June 1993, pp. 117–127.

[10] M. Ben-Ari, Z. Manna, and A. Pnueli, *The Temporal Logic of Branching Time*, Proc. of the 8th Annual Symposium on Principles of Programming Languages (1981), 164–176.

[11] R. Bharadwaj and C. L. Heitmeyer, *Applying the SCR Requirements Specification Method to Practical Systems: A Case Study*, Proc. Software Engineering Workshop, NASA Goddard Space Flight Center, 1996.

[12] P. Du Bois, *The Albert II Language: On the Design and the Use of a Formal Specification Language for Requirements Analysis*, Ph.D. thesis, Computer Science Department, University of Namur, Belgium, September 1995.

[13] P. Du Bois, E. Dubois, and J-M. Zeippen, *On the Use of a Formal RE Language: The Generalized Railroad Crossing Problem*, Proc. Int'l Symp. Requirements Eng. (RE '97), IEEE Computer Society Press, January 1997, pp. 128–139.

[14] Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas, *A Procedure for Designing Abstract Interfaces for Device Interface Modules*, Proc. Int'l Conf. Software Eng. (ICSE), 1981, pp. 195–204.

[15] Marsha Chechik and John Gannon, *Automatic Analysis of Consistency between Requirements and Designs*, IEEE Software **27** (2001), no. 7, 651–672.

[16] Yoonsik Cheon and Gary T. Leavens, *The Larch/Smalltalk Interface Specification Language*, ACM Transactions on Software Engineering and Methodology (TOSEM) **3** (1994), 221–153.

[17] E.M. Clarke, E.A. Emerson, and A.P. Sistla, *Automatic Verification of Finite-State Concurrent Systems using Temporal-Logic Specifications*, ACM Transactions on Programming Languages and Systems **8(2)** (1986), 244–263.

[18] D.L. Dill, *Timing Assumptions and Verification of Finite-State Concurrent Systems*, CAV 89: Automatic Verification Methods for Finite-state Systems, Lecture Notes in Computer Science 407 (1989), 197–212.

[19] D.Ron, *Temporal Verification of Communication Protocols*, Master's thesis, The Weizmann Institute of Science, Rehovot, Israel, 1984.

[20] E. A. Emerson, *Temporal and Modal Logic*, Handbook of Theoretical Computer Science (J. van Leeuwen, ed.), vol. B: Formal Models and Semantics, Elsevier Science, 1990, pp. 995–1072.

[21] Stuart R. Faulk, *Specifying Concurrent Events in SCR Requirements*, Proc. Int'l Software Cost Reduction Workshop (Ottawa, Ontario), February 1996.

[22] John Fitzgerald and Peter Gorm Larsen (eds.), *Modelling Systems: Practical Tools and Techniques in Software Development*, Cambridge University Press, 1998.

[23] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, *On the Temporal Analysis of Fairness*, Proc. of 7th Annual Symposium on Principles of Programming Languages (1980), 163–173.

[24] Hassan Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison-Wesley, 2000.

[25] D. Harel, *Statecharts: A Visual Formalism for Complex Systems*, Science of Computer Programming **8** (1987), 231–274.

[26] David Harel and Michal Politi, *Modeling Reactive Systems with Statecharts : the STATEMATE Approach*, New York ; Montreal : McGraw-Hill, c1998, 1998.

[27] E. Harel, *Temporal Analysis of Real-Time Systems*, Master's thesis, The Weizmann Institute of Science, Rehovot, Israel, 1988.

[28] E. Harel, O. Linchtenstein, and A. Pnueli, *Explicit-Clock Temporal Logic*, In Proceedings of the Fifth Annual Symposium on Logic in Computer Sceince (1990), 402–413.

[29] M. P. E. Heimdahl and N. G. Leveson, *Completeness and consistency in hierarchical state-based requirements*, IEEE Software **22** (1996), no. 6, 363–377, reprinted from ICSE96.

[30] C. L. Heitmeyer, *Requirements Specifications for Hybrid Systems*, Hybrid systems III: verification and control (New York, NY, USA) (Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, eds.), Lecture Notes in Computer Science, no. 1066, Springer-Verlag, 1996, pp. 304–314.

[31] Constance L. Heitmeyer, A. Bull, C. Gasarch, and Bruce G. Labaw, *SCR*: A Toolset for Specifying and Analyzing Requirements*, Proc. Conf. Computer Assurance (COMPASS) (Gaithersburg, MD), National Institute of Standards and Technology, June 1995, pp. 109–122.

[32] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw, *Automated Consistency Checking of Requirements Specifications*, ACM Trans. Software Eng. and Methodology **5** (1996), no. 3, 231–261.

[33] K. Heninger, R. Parker, and D. Parnas, *A Procedure for Designing Abstract Interfaces for Device Interface Modules*, Software fundamentals: collected papers by David L. Parnas (2001), 295–314.

[34] Katherine L. Heninger, *Specifying Software Requirements for Complex Systems: New Techniques and their Application*, IEEE Software **SE-6** (1980), no. 1, 2–13.

[35] Katherine L. Heninger, David Lorge Parnas, John E. Shore, and J. Kallander, *Software Requirements for the A-7E Aircraft*, Tech. Report MR 3876, Naval Research Laboratory, 1978.

[36] T.A. Henzinger, Z. Manna, and A. Pnueli, *Temporal Methodologies for Real-Time Systems*, In Proceedings of the 18th Annual Symposium on Principles of Programming Languages (1991), 353–366.

[37] Daniel Hoffman and Paul Strooper, *Software Design, Automated Testing, and Maintenance: A Practical Approach*, International Thomson Computer Press, 1995.

[38] International Telecommunication Union, Geneva, *Specification and Description Language, SDL*, 1992, Recommendation Z.100.

[39] A. Jackson and D. Hoffman, *Inspecting Module Interface Specifications*, Software Testing, Verification and Reliability **4** (1994), no. 2, 101–117.

[40] Jonathan Jacky, *The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, 1997.

[41] F. Jahanian and A. K. Mok, *Modechart: A Specification Language for Real-Time Systems*, IEEE Software **20** (1994), no. 12, 933–947.

[42] Farnam Jahanian and Aloysius K. Mok, *Safety Analysis of Timing Properties in Real-Time Systems*, IEEE Software **SE-12** (1986), no. 9, 890–904.

[43] Ryszard Janicki, David Lorge Parnas, and Jeffery Zucker, *Tabular Representations in Relational Documents*, Relational Methods in Computer Science—Advances in Computing Science (C. Brink, W. Kahl, and G. Schmidt, eds.), Springer Wien, New York, 1997, pp. 184–196.

[44] Ryszard Janicki and Emil Sekerinski, *Foundations of the Trace Assertion Method of Module Interface Specification*, IEEE Software **27** (2001), no. 7, 577–598.

[45] James Kirby, Jr., *Example NRL/SCR Software Requirements for an Automobile Cruise Control and Monitoring System*, Tech. Report TR-87-07, Wang Institute of Graduate Studies, July 1987.

[46] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, *Requirements Specification for Process-Control Systems*, IEEE Software **20** (1994), no. 9.

[47] N. Lynch and F. Vaandrager, *Forward and Backward Simulations-Part II: Timing-Based Systems*, Information and Computation **128** (1996), no. 1, 1–25.

[48] N.A. Lynch and H. Attiya, *Using Mappings to Prove Timing Properties*, In Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing (1990), 265–280.

[49] Nancy Lynch, Roberto Segala, and Frits Vaandrager, *Hybrid I/O Automata Revisited*, Hybrid Systems: Computation and Control, Fourth International Workshop (Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, eds.), Lecture Notes in Computer Science, no. 2034, Springer-Verlag, March 2001.

[50] D. Mandrioli, A. Morzenti, M. Pezzè, P. San Pietro, and S. Silva, *A Petri Net and Logic Approach to the Specification and Verification of Real-Time Systems*, Formal Methods for Real-Time Computing (Constance L. Heitmeyer and Dino Mandrioli, eds.), Trends in Software, no. 5, John Wiley and Sons, 1996, pp. 135–166.

[51] Theodore S. Norvell, *On Trace Specifications*, CRL Report 305, Communications Research Laboratory, Hamilton, Ontario, Canada, March 1995.

[52] J. S. Ostroff, *Temporal Logic for Real-Time Systems*, John Wiley & Sons Inc., 1989.

[53] D. Parnas, *Tabular representation of relations*, CRL Report 260, Communications Research Laboratory, Hamilton, Ontario, Canada, November 1992.

[54] David L. Parnas, *Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems*, Naval Research Laboratory (1977), no. 8047, NRL Report.

[55] David Lorge Parnas, *On the Criteria to be Used in Decomposing Systems into Modules*, Communications of the ACM (1972), 1053–1058.

[56] David Lorge Parnas and Jan Madey, *Functional Documentation for Computer Systems*, Science of Computer Programming **25** (1995), no. 1, 41–61.

[57] David Lorge Parnas and Yabo Wang, *The Trace Assertion Method of Module Interface Specification*, Tech. Report TR89-261, Queen's University, Telecommunications Research Institute of Ontario (TRIO), October 1989.

[58] Dennis K. Peters, *Deriving Real-Time Monitors from System Requirements Documentation*, Ph.D. thesis, McMaster University, Hamilton ON, January 2000.

[59] A. Pnueli, *The Temporal Logic of Programs*, Proc. of 18th Annual Symposium on Fundations of Computer Science (1977), 46–57.

[60] A. Pnueli and W.-P. de Roever, *Rendez-vous with Ada: A Proof-Theoretical View*, In Proceedings of the SIGPLAN AdaTEC Conference on Ada (1982), 129–137.

[61] A. Pnueli and E. Harel, *Applications of Temporal Logic to the Specification of Real-Time Systems*, Formal Techniques in Real-time and Fault-tolerant Systems, Lecture Notes in Computer Science 331 (1988), 84–98.

[62] Rational Software Inc., *et al*, *UML summary*, version 1.1 ed., September 1997.

[63] A. P. Ravn, H. Rischel, and K. M. Hansen, *Specifying and Verifying Requirements of Real-Time Systems*, IEEE Software **19** (1993), no. 1, 41–55.

[64] S. Sankar and R. Hayes, *Specifying and Testing Software Components using ADL*, Tech. Report SMLI TR-94-23, Sun Microsystems Laboratories, Inc., Mountain View, California, April 1994.

[65] Sriram Sankar, *Introducing Formal Methods to Software Engineers through OMG's CORBA Environment and Interface Definition Language*, Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology (1996).

[66] Arcot Sowmya and S. Ramesh, *Extending Statecharts with Temporal Logic*, IEEE Software **24** (1998), no. 3, 216–231.

[67] E. W. Thompson and R. F. Bridge, *A Module Interface Specification Language*, Proceedings of the 12th conference on Design Automation (1975), 42 – 49.

[68] A. John van Schouwen, *The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems*, Tech. Report TR 90-276, Queen's University, Kingston, Ontario, 1990, also printed as 242.

[69] A. John van Schouwen, David Lorge Parnas, and Jan Madey, *Documentation of Requirements for Computer Systems*, Proc. Int'l Symp. Requirements Eng. (RE '93), IEEE, January 1993, pp. 198–207.

[70] Sreenivasa Viswanadha and Deepak Kapur, *IBDL: A Language for Interface Behavior Specification and Testing*, Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems(COOTS) (1998).

[71] G. H. Weiss, R. Hohendorf, A. Wassyng, B.Quigley, and M. R. Borsch, *Verification of the Shutdown System Software at the Darlington Nuclear Generating Station*, Int'l Conf. Control & Instrumentation in Nuclear Installations (Glasgow, United Kingdom), no. 4.3, Institution of Nuclear Engineers, May 1990.

[72] Jeannette M. Wing, *Writing Larch Interface Language Specification*, ACM Transactions on Programming Languages and Systems (TOPLAS) **9** (1987), 1–24.