# Framework for Specification of Concurrent and Reactive Systems in Unified Modeling Language

**THESIS**

Submitted in partial fulfillment
of the requirements for the degree of
**DOCTOR OF PHILOSOPHY**

by

**Suryadevara Jagadish**

Under the Supervision of
**Prof. R. K. SHYAMASUNDAR**

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE**
**PILANI (RAJASTHAN) INDIA**
**2009**

Dedicated to

**my grandparents**

Late. Mallampati Arjunaiah

& Seetharathnamma

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE**

**PILANI (RAJASTHAN) INDIA**

## <u>CERTIFICATE</u>

This is to certify that the thesis entitled "**Framework for Specification of Concurrent and Reactive Systems in Unified Modeling Language**" and submitted by **Mr. Suryadevara Jagadish** ID No. **2002PHXF415** for award of the Ph.D degree of the Institute embodies original work done by him under my supervision.

Signature of the Supervisor

Date:

**Name**: Prof. R.K. Shyamasundar

**Designation**: Sr. Professor,

Tata Institute of Fundamental Research (TIFR), Mumbai, India

# ACKNOWLEDGEMENTS

I am most indebted to my supervisor, Professor R.K. Shayamasundar, Tata Instittue of Fundamental Research (TIFR) and Professor L.K.Maheswari, Vice-Chancellor, BITS for their guidance, wisdom, valuable suggestions and encouragement throughout my work. Talking to them has always been enlightening and highly productive. It has been a great learning experience working under their guidance.

I thank Prof. Lawrence Chung, University of Texas, Dallas, USA for providing critical inputs during the thesis work.

I thank Prof Sundar Balasubramaniam, and Prof J.P. Misra, senior members of CSIS Group, BITS for their support and encouragement. I also thank all my colleagues who directly and indirectly helped me in completing my thesis. Special thanks are to Dr. T.S.B Sudarshan, member, CSIS group.

I thank Prof Ravi Prakash, Dean and other members of Research & Consultancy Division, BITS who directly or indirectly helped me during this period. I also thank Prof. Navneet Goyal, and Dr. Mukesh K. Rohil, members of my Doctoral Advisory Committee (DAC) for reviewing the draft thesis and providing valuable inputs.

Last but not least, my special thanks to my family; Anuradha, my wife, for all the support and Baby Nandana and Baby Mahima, my little daughters for giving me joyous moments during the stress and strain of ph.d study.

I thank *The Almighty GOD* for all good things that happened in my life.

# ABSTRACT

With emergence of real-time, embedded systems and high performance computing environments, development processes for complex systems with concurrent and reactive behaviors has become important. There exist wide gap between the initial requirements phases and final system design for these systems. Ideally a formal, intuitive specification phase between requirements and architecture design is needed. The Unified Modeling Language (UML), a *de facto* industry standard, provides intuitive graphical design notations for modeling static as well as dynamic aspects of systems. But, the behavioral specification in UML is representative in nature rather than complete. Further, UML lacks constructs, semantics towards precise specification of concurrent, reactive behaviors.

There are major shortcomings in UML as a precise language to specify complex behaviors of concurrent and reactive systems. The higher level formalisms like Sequence diagrams, Statecharts, and Activity diagrams are inconsistent and are not well integrated with the underlying object model of UML. Current UML based approaches depend on low-level primitives such as semaphores, monitors etc., to model concurrency features. Many Real-Time UML methodologies for example COMET, CODARTS are largely based on informal design heuristics with focus on static aspects of the systems. For example liveness issues of concurrent systems are not considered in above approaches. There exist formal approaches with precise semantics for UML such as RT-UML, UML-RT, and UML/SDL. There also exist translation based approaches into specific formalisms like CSP, LOTOS, Esterel etc for translating and analyzing certain aspects of UML models. Though promising, these approaches are neither intuitive nor integrated within traditional development processes.

This thesis proposes a semi-formal specification framework in UML, namely cmUML, in the development of systems with concurrent and reactive behaviors. The framework adopts the principles and semantics of formal specification approaches for concurrent and reactive systems. It further clarifies inconsistencies and ambiguities in the usage and semantics of UML diagrams through a unifying framework, based on conceptual semantics and the foundations of the standard UML/ SPT Profile, and provide a basis for explicit introduction of concurrency and reactivity. The framework is independent of implementation level primitives and associated semantics.

The main contributions of the thesis are:

- Definition of abstract components, with heterogeneous behaviors and semantics, to hierarchically constitute precise specification of complex systems
- Integrating principles and semantics of formal specification approaches for concurrent, reactive systems e.g., liveness
- Definition of specification constructs and mechanisms for explicit specification of complex behaviors of systems i.e., concurrency, reactivity, exception handling, and synchronization
- Use of multiple behavior diagrams of UML i.e., sequence charts, state machines, activities to provide multi-view, intuitive graphical specifications under a unified semantic framework
- Application of *separation-of-concerns* in system specifications through interface and internal specifications corresponding to *precise* requirements and an *abstract* implementation respectively
- Definition of a UML Profile, based on standard extension mechanisms of UML, for application of the framework
- Specification methodology for application of the specification framework and related UML profile
- Definition of a formal semantics for the proposed specification framework
- Integration of existing verification techniques and tools for early analysis of system specifications

The proposed specification framework is validated through specification of a case-study i.e. a *vending machine* as well as specification of well-known concurrency patterns e.g., *producer-consumer*, *reader-writer*, and *leader-follower*. The specifications are compared with pure formal approaches as well as existing UML-based approaches.

The thesis is supported through the following publications:

1. Jagadish Suryadevara, Lawrence Chung, Shyamasundar R.K., “cmUML – A UML based Framework for Formal Specification of Concurrent, Reactive Systems”, in *Journal of Object Technology*, vol. 7, no. 8, November - December 2008, pp. 187 – 207 (submitted 2007)

2. Jagadish Suryadevara, Shyamasunder R.K., “An UML-based approach to Specify Secured, Fine-grained, Concurrent Access to Shared Resources” Journal of Object Technology (JOT), vol.6 no.1, Jan-Feb, 2007, pp 107-119 (submitted 2006).

3. Jagadish Suryadevara, Shyamasundar R.K., “cmUML- A Precise UML for Abstract Specification of Concurrent Components”, Proceedings of 18th *International Conference on Parallel and Distributed Computing and Systems (PDCS), Dallas*, USA, Acta press, November 2006, pp 141-146.

4. Jagadish Suryadevara, Shyamasundar R.K., “A Multi-threaded Active Object Behavioral Pattern for Concurrent/ Distributed Programming in Java”, Proc. First International Conference on Web Engineering and Applications (ICWA-06) Bhubaneswar, India, pp 230-239, Macmillan press, India

# TABLE OF CONTENTS

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| ADL | : Architecture Description Language |
| DF | : Data Flow |
| FSM | : Finite State Machine |
| ITU | : International Telecommunications Union |
| LSC | : Live Sequence Chart |
| MSC | : Message Sequence Chart |
| OCL | : Object Constraint Language |
| OMG | : Object Management Group |
| SPT | : Schedulability, Performance, and Time |
| SR | : Synchronous Reactive |
| STS | : Symbolic Transition System |
| UML | : Unified Modeling Language |

# CHAPTER 1

# INTRODUCTION

Software development life cycle methods and processes play an important role in the development of software systems. With the emergence of inexpensive multi-processor hardware platforms, real-time embedded software systems with complex behaviors have grown manifold. Traditionally the development processes are designed around sequential execution models with guaranteed correctness. But, modern software systems have characteristic behaviors of concurrency, reactivity and timeliness aspects. These systems need rigorous development processes in formal or semi-formal approaches. Thus there is growing need to integrate these approaches in traditional software engineering methods and processes. Further, the integration needs to be done in an intuitive way to enable common system modelers and developers to work with formal concepts, techniques, and tools.

This chapter describes the motivation and scope of study behind the thesis work in terms of software engineering methods, formal methods, and emergence of Unified Modeling Language. The chapter further describes the main objectives of the thesis. Also, the organization of the thesis is described providing a brief overview of each chapter.

## 1.1 THESIS MOTIVATION

Modern software systems have grown in size and complexity. While the computing hardware is becoming cheaper and faster, the development complexity and cost of software has been increasing. Traditionally the software development has been organized into software engineering phases of requirements, software design, testing, maintenance etc [Pressman 2004]. To handle the complexity of modern software systems, approaches like model driven development, programming-in-large, software architectures, component based software development have emerged. These approaches are traditionally integrated into software development phases [Gomaa 1993, 2000]. These approaches are largely successful in obtaining system design in terms of functional requirements.

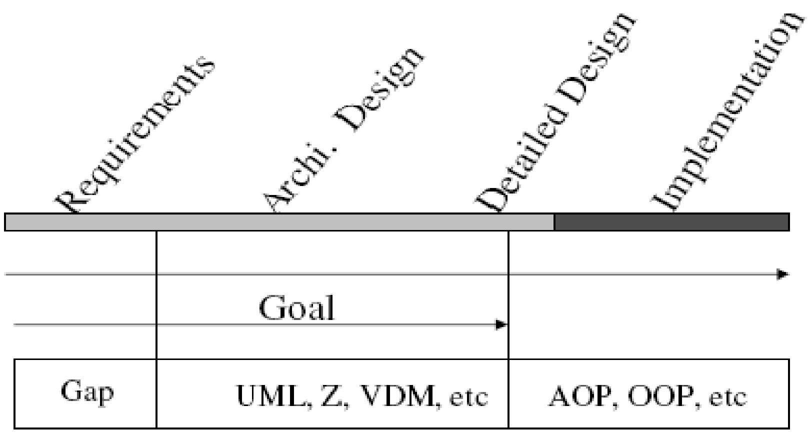


Figure 1.1 Software Development Phases

But modern software systems possess complex behaviors of concurrency, parallelism, reactivity, timeliness etc [Gomaa 1993, 2000]. With the increase in complexity of software systems an early design phase known as architectural design has become important (figure 1.1). This phase focuses on designing the system to achieve non-functional properties [Bass 2003]. But existing architectural design practices are ad hoc and informal in nature. Hence there still exists a wide gap between requirements phase and architectural design phase.

For systems with complex behaviors the gap between requirements analysis and architectural design is a major challenge. The critical non-functional aspects of the systems like concurrency, timeliness etc are neither well specified nor understood under current development approaches. Traditionally concurrency issues are handled during architecture or system design phases using implementation primitives like threads, semaphores, monitors etc. Hence there is a need to introduce a formal or semi-formal specification phase between requirements and architectural design towards better understanding, analysis, design and implementation of complex software systems.

### 1.1.1 SOFTWARE ENGINEERING AND FORMAL METHODS

Formal methods are useful to develop complex software systems using engineering methods and tools that are verifiable [Liu 2005]. Formal approaches complement informal engineering methods by techniques like formal specification and verification. They have been extensively researched and studied [Liu 2005]. A range of semantic theories, specification languages, design techniques, and verification methods and tools have been developed that are used in critical applications. However, it is still a challenge to scale up formal methods and integrate them into engineering development processes for the correct construction of software systems. Both formal methods and the methods adopted by software engineers are far from meeting the developmental complexity of complex systems. Further, complete assurance of correctness requires too much to specify and verify and thus a full automation of the verification is infeasible.

However, recently there have been encouraging developments in both approaches. The software engineering community has started using precise models for early requirement analysis and design [Mellor 2002]. Theories and methods for object-oriented, component-based and aspect-oriented modeling and development are gaining the attention of the formal methods community. But, as system development community largely consists of people who do not possess expertise in formal methods, there needs to be intuitive yet formal approaches hiding the complexity of formal methods.

### 1.1.2 UNIFIED MODELING LANGUAGE (UML)

The Unified Modeling Language (UML) has become the de facto industry standard visual specification language for specification of software systems [OMG 2001]. UML consists of design notations such as class diagrams, sequence diagrams, activity diagrams, and statecharts towards static as well as dynamic aspects of systems. UML can be used in several phases of development processes of systems i.e., requirements, architecture design, and detailed design. An oft-repeated criticism of UML is that it has no semantics largely due to the fact that UML lacks formal semantics. The informal semantics described using a natural language like English leads to various inconsistencies, and ambiguities.

## 1.2 SCOPE OF THE THESIS WORK

There are major shortcomings in UML as a precise language to specify complex behaviors of concurrent and reactive systems. The higher level constructs or formalisms like Sequence diagrams, Statecharts, and Activity diagrams are inconsistent and are not well integrated with the underlying object model of UML [Ober 1999, Jagadish 2006a]. Current UML based approaches depend on low-level primitives such as semaphores, monitors, etc to model concurrency features. From the perspective of formal languages, they use complex OCL (Object Constraint Language) statements [Goni 2004]. Many Real-Time UML methodologies for example COMET, CODARTS are largely based on informal design heuristics with focus on static aspects of the systems [Gomaa 2000]. For example safety and liveness issues of concurrent systems are not handled in these approaches.

There have been many attempts towards providing formal approaches with precise semantics for UML such as RT-UML, UML-RT, and UML/SDL. There also exist translation based approaches (e.g. [Clark 2000]) into specific formalisms like CSP, LOTOS, Esterel etc translating and interpreting certain aspects of UML models under specified semantic domain and analyzed with the related tools. Though promising, these approaches are not satisfactory for complete behavioral aspects of concurrent and reactive systems. Further these approaches are not integrated with traditional development phases in general and requirements phase in perticular.

Object Management Group (OMG), the UML consortium, has defined a generic resource modeling framework with concurrency and causality for real-time systems known as 'Standard UML profile for Schedulability, Performance, Time' (also known as UML/ SPT Profile) [OMG 2001]. The SPT profile defines basic concepts of events, causality, and concurrency independent of formal semantics and UML meta model [Ober 2004].

Component based approaches have emerged as an effective way of specifying, verifying, and implementing complex systems. Of these, works on software architectures for example Architecture Description Languages (ADLs) are significant. The ADL community has contributed much in terms of interfaces, components, and system compositionality. Certain ADLs like Rapide, Darwin, and Wright are based on precise formal semantics [Luckham 1995, Nenad 2000, Magee 1995, Allen 1997].

A multi-view and multi-notation modeling language, as a formalized subset of the UML, can be used to specify precise system or component models and analyzed for inconsistencies using formal techniques, e.g. model-checking [Shrotri 2003]. The models can further be enhanced by adding descriptions of interaction protocols with the environment, timing aspects etc. The analysis can be carried out incrementally, a small number of use cases at a time that only involve a small number of domain classes.

## 1.3 OBJECTIVES OF THE THESIS

Following objectives are identified for the thesis work:

- Precise behavioral specification of concurrent, reactive systems in hierarchical approach with higher level conceptual semantics
- Integrating principles and semantics of formal specification approaches
- Explicit specification of externally visible behaviors i.e., concurrency, reactivity, exception handling, and synchronization of system components
- Use of all behavior diagrams of UML i.e., sequence charts, state machines, activities in well defined contexts under a unifying semantic framework
- Resolving inconsistencies, and ambiguities in UML behavioral semantics
- Integration of the specification process with functional requirements (use cases)
- The definition of a specification process, independent of design or implementation constructs e.g., threads, semaphores, monitors
- To define constructs and mechanisms for explicit specification of exception handling and synchronization features.

- To investigate the existing formal verification techniques that can be integrated with the proposed specification framework

## 1.4 THESIS ORGANIZATION

The thesis has been organized into seven chapters as described below.

Chapter 1 has introduced the motivation and objectives of the thesis work describing the limitations of traditional software development processes, Unified Modeling Language, and role of formal methods in software engineering. The scope of the research work is discussed. The objectives of the thesis work are described.

Chapter 2 presents the detailed background and literature survey of the thesis work. The chapter describes formal approaches, and principles of formal specification of concurrent, reactive systems. The transition axiom method, a simple and efficient formal approach proposed by Lamport, is described. It combines both axiomatic and operational approaches towards component based specification of concurrent systems. Also, the chapter presents the UML framework as described in the literature. The advantages and limitations of existing UML based approaches for modeling concurrent and reactive systems are described. An overview of the existing formal approaches in UML is presented.

Chapter 3 presents the first part of the thesis contribution. It describes the conceptual model of the proposed framework (namely cmUML). The corresponding specification language, as precise subset of UML using standard extension mechanisms known as stereotypes, tags and constraints, is defined. The standard UML/SPT profile which forms the conceptual foundation of the proposed framework is described. The chapter also presents the informal semantics of the proposed abstractions of the cmUML framework.

Chapter 4 presents a specification methodology with the proposed cmUML specification framework. The methodology is defined in terms of a sequence of well-defined specification tasks. This is significant as OMG does not prescribe any standard process for application of UML (usually left to the profile developers). The specification process is integrated with the traditional requirements phase. This requirements driven process identifies suitable system decomposition and specification of containing behaviors with concurrency, reactivity, exception handling, and synchronization features. Also, the specification process incorporates design guidelines, and heuristics of general object oriented analysis and design approaches [Gomaa 2000]. The specification methodology is illustrated with a simple case study of vending machine specification. The chapter further presents a comparison and validation of cmUML approach using classical examples of concurrency i.e., readers-writers problem and producer-consumer problem against existing UML approaches as well as formal approaches. Also, the concurrency pattern leader-follower is specified.

Chapter 5 describes the semantics foundations of the proposed specification framework. As the framework proposes a two level specification process in terms of interface specifications (based on LSCs and protocol state machines), and internal specifications (based on activities and statemachine), the semantic description is described along two separate but related dimensions. For interface specifications, the semantic framework is described in terms of LSCs and proposed extensions. The semantics of the internal specifications is described using the formalism of 'Symbolic Transition Systems (STS)' and is based on UML foundations e.g., action semantics.

Chapter 6 presents the existing verification approaches that can be integrated with the cmUML specifications. For interface verifications, the LSC semantic framework is extended to integrate the exception handling aspects of the specifications. For this, a simple algorithm is proposed. Also LSC based verification techniques are described for consistency checks of the various parts of the specifications. For example, consistency checking scenarios are described like LSCs vs interface statemachines, LSCs vs Internal statemachines, and Interface vs internal statemachines. For verification of internal specifications, an application of CSP based model checking technique is described (e.g. deadlock analysis).

Chapter 7 presents the thesis conclusions in terms of contributions, and limitations of the thesis work. Further the future work related to the thesis work is discussed.

The thesis includes following appendices:

The appendix A describes the characterization of concurrent systems in terms of their safety and liveness properties.

The appendix B describes the precise formalism of Live Sequence Charts and their semantic foundation.

The appendix C presents the complete specification of the vending machine case study described in chapter 4.

# CHAPTER 2
# BACKGROUND & LITERATURE SURVEY

Specification serves as a contract, a valuable piece of documentation, and a means of communication among clients, designers and implementers. A formal specification provides the means of precisely defining the notions like consistency, completeness, and more importantly correctness. Formal approaches help specify, develop, and verify systems in a systematic approach rather than ad hoc means.

This chapter describes the foundations of formal methods in specifying systems with a special focus on Transition Axiom method which combines axiomatic and operational approaches for intuitive yet analyzable specifications [Wing 1995, Lamport 2000]. A brief overview of UML is presented, in particular the semantic architecture of UML 2.0, and corresponding ambiguities, inconsistencies with respect to specification of concurrency and reactivity. Further the related works are described. Finally, the research gaps are identified and the research problem formulated.

## 2.1 FORMAL SPECIFICATIONS OF SOFTWARE SYSTEMS

### 2.1.1 INTRODUCTION

A formal specification language provides a notation (its syntactic domain), a universe of objects (its semantic domain), and a precise rule defining which objects satisfy each specification [Wing 1990]. A language's syntactic domain is usually defined in terms of a set of symbols and a set of grammatical rules for combining these symbols which need not be restricted to text; graphical elements such as boxes, circles, lines, arrows, and icons can be given a formal semantics. The languages differ in their choice of semantic domains. For example, specification languages for concurrent systems need to specify state sequences, event sequences, state and transition sequences, streams, synchronization trees, partial orders, and state machines. The different aspects or 'views' of a single specificand (a semantic object) are best specified perhaps using different specification languages (or sublanguages). The specifications i.e. both structural and behavioral imply constraints to be 'satisfied' by specificands. Behavioral specifications describe only constraints on the observable behavior of specificands and usually address system's functionality but can include non-functional aspects.

Formal specifications can play an important role in the traditional development processes. The general scenario involving the main stakeholders during system development process is described in figure 2.1. Specifiers should specify enough so that implementers do not make unacceptable implementations. On the other hand, saying too much may leave little design freedom for the implementer [Lamport 2000]. Informally, a specification has 'implementation bias' if it specifies externally unobservable properties of its specificands. From figure 2.1 it is clear that many actors of system development process including human beings, and tools interact with the specifications. Many languages are suitable only for a subset of the actors. Further a specification language may suit only specific kinds of systems as well as specific phase of the development process.

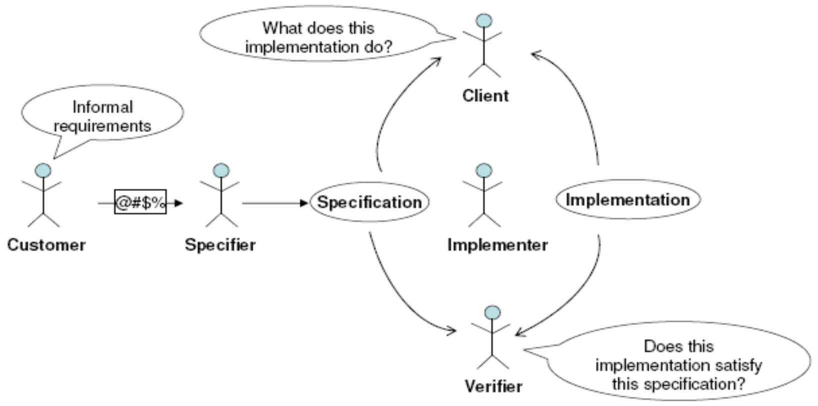


Figure 2.1. Specification as Core Artifact of Development Process [Source: Wing 1990]

Verification is the process of showing that a system satisfies its specification. Formal verification is impossible without a formal specification. Although an entire system may never be verified completely, its smaller, critical components can be verified. Further specifications can be used to generate test cases for black-box testing. For example, specifications that explicitly state assumptions on a module's use identify test cases for boundary conditions. Further specification languages could be executable allowing rapid prototyping of systems but the disadvantage could be that such languages may suffer from implementation bias.

### 2.1.2 SPECIFICATION METHODS & LANGUAGES

Formal methods can not describe an entire large, software system but only certain aspects or certain views of it [Lamport 2000]. It is very important to understand 'why', 'what', and 'how' to formally specify systems. Formal specifications may help precise documentation of system or its component interfaces, or towards precise and abstract design, or to perform some formal analysis, etc [Wing 1995]. Formal methods can further be used to specify 'global correctness conditions' (e.g. deadlock freedom), 'system invariants', 'observable behavior', or 'properties or entities' of a system at a suitable level of abstraction (e.g. interface, implementation) by characterizing the observable entities forming system's 'state variables' at that level. State transitions correspond to operations that access or modify the observable behavior. For each operation, its observable effect on the observable state entities may be specified. Further the observable behavior should include any change in state that is observable at that abstract level e.g. changes to state variables, signaled exceptions, errors, etc.

The fundamental techniques regarding 'how' to specify systems are known as 'abstraction' and 'decomposition'. Certain methods e.g., Z, VDM facilitate a model based approach to specification [Spivey 1988, Jones 1986]. Models give good intuition about the system but needs to be related with necessary algebraic or axiomatic assertions about the system. The system may be specified following incremental abstraction techniques:

i) finding necessary pre-conditions (i.e. assumptions about the environment)
ii) first handling the normal case, then the failure case
iii) first assuming the operations are atomic, then introducing necessary interleaving

Further, it is important to specify erroneous or exceptional behaviors i.e. errors, exceptions, or failures. For any system it is important to identify operations that are ‘atomic’ w.r.t the level of abstraction. For a concurrent system, it is critical to state explicitly operations that are atomic. Specifying ‘non-determinism’ is another effective technique of achieving abstraction.

Formal methods for specification of concurrent systems differ in terms of their semantic domains i.e. states, or events or both. A system’s behavior can be modeled as sequence of states and associated events or set of trees of states and associated events. When specification of concurrent systems is interpreted as sets of sequences of states, the system properties can be described in terms of ‘safety’ (e.g. functional correctness) and ‘liveness’ (e.g. termination) [Alpern 1985]. Temporal logic is a property oriented method for specifying properties of concurrent systems [Pnueli 1986]. It uses specific operators to refer to past, current, and future states (or events). CSP uses a model oriented method for specifying concurrent processes and a property oriented method for stating and proving properties about the model [Hoare 1985]. CSP is based on model of ‘traces’ or event sequences, and assumes processes communicate by sending messages across channels. Processes synchronize on events. Lamport’s transition axiom method combines an axiomatic method for describing the behavior of individual operations with temporal logic assertions for specifying safety and liveness properties [Lamport 1983].

Last but important aspect is about the assumptions (implicit or explicit) made on the environment of a system or its components. Many formal methods do not make these explicit. Environment represents a set of assumptions for the correct behavior of the system or its components. Whereas a specifier places constraints on the system’s behavior, the specifier can not place constraints on the environment but can only make assumptions about its behavior. As no one method is suitable for specifying all aspects or all kinds of systems, the only practical strategy is to mix the methods e.g. Z to specify static aspects (i.e. state space), and CSP to specify dynamic behavior (sequences of state transitions).

However mixing methods is dangerous as they are based on different semantics and could lead to inconsistencies of the model. This is still the subject of research efforts.

### 2.1.3 TRANSITION AXIOM METHOD

Proposed by Lamport, the transition axiom method provides a conceptual and logical foundation for writing formal specifications of concurrent systems. The transition axiom method specifies the behavior of a system i.e., the sequence of observable actions it performs when interacting with the environment. More precisely, it specifies two classes of behavioral properties: safety and liveness properties. Safety properties assert what the system is allowed to do, or equivalently, what it may not do. Liveness properties assert what the system must do. The method emphasizes the precise and detailed specification of interfaces, the mechanisms by which the system communicates with the environment, as the influence of the interface on the rest of the specification is especially important in concurrent systems. For example, it is shown that the specification of even so basic a property as *first-come-first-served priority* cannot be independent of the interface's implementation details.

Transition axiom method can be used for specification of a module in a concurrent program, where a module is a collection of related subroutines. A module must be specified in terms of its behavior, rather than the values it returns. Though temporal logic has proved to be a successful tool in reasoning about the behavior of concurrent programs, it is not convenient for expressing many aspects of concurrent programs. New kinds of constructs, with precise formal interpretations, make the specifications simpler and easier to understand. The transition axiom method introduces new kinds of assertions. The transition axiom method is formalized based on a generalization of temporal logic to include predicates for describing the actions that are executed. However, the specifications can be understood with no knowledge of the formal temporal logic upon which they are based.

Unlike many other protocol specification methods, the transition axiom method specifies not an abstract protocol but an actual program module containing subroutine calls for sending and receiving messages. In practice, one is concerned not with abstract protocols but with the program modules that implement them. An extra layer of formalism relating programs and abstract protocols is needed to verify that a program module correctly implements such an abstract protocol specification. In transition axiom method, this extra layer is avoided by specifying the program module itself.

The execution of a concurrent program can be represented as a sequence of state transitions of the form (S → S' after executing a single atomic statement 'a') which denotes that the action 'a' takes the program from state s to state *s'*. Typically, this transition would represent the execution of a single atomic statement in some process, in which case s is the state before the execution, *s'* is the state immediately after the execution, and 'a' denotes the program statement being executed. However, the exact nature of the states and actions does not concern the method. Concurrency is represented by the interleaving of concurrent atomic operations. A state represents a complete "snapshot" of the program at some instant of time. At any point during the execution, the possible future behavior of the program must depend only upon its current state, and not upon how it reached that state. Thus, the state must include not only the value of program variables, but also the values of processes 'program counters', the values of parameter passing stacks, the contents of message queues, the states of transmission lines, etc.

A *state function* is a mapping from the set S of states into some set of values. A *predicate* is a boolean-valued state function. There are two kinds of primitive state functions that can be used: program variables, control predicates. Complex state functions can be constructed from these primitive ones. A program is specified by specifying all its possible execution sequences. A specification consists of a collection of conditions on execution sequences. A program satisfies the specification if all of its possible execution sequences satisfy each of these conditions.

## 2.2 UNIFIED MODELING LANGUAGE (UML)

The Unified Modeling Language (UML) is a general-purpose graphical language to specify, visualize, construct, and document the artifacts of a software system [OMG 2001, Selic 2004]. It captures decisions and understanding about systems that must be constructed. It is intended for use with all development methods, lifecycle stages, application domains, and media and is intended to unify past experience about modeling techniques and to incorporate current software best practices into a standard approach. UML includes semantic concepts, notation, and guidelines. It has static, dynamic, environmental, and organizational parts. The UML specification does not define a standard process but is intended to be useful with an iterative development process. The UML captures information about the static structure and dynamic behavior of a system. Modeling a system from several separate but related viewpoints permits it to be understood for different purposes. UML is not primarily a programming language. Tools can provide code generators from UML into a variety of programming languages.

### 2.2.1 OVERVIEW

After several years of experience with UML, the OMG has revised the language features, semantics fixing problems uncovered by experience gained in using UML. Current version UML2 has some new features [Selic 2004]:

- Sequence diagram constructs and notation based, largely on the ITU Message Sequence Chart standard, adapted to make it more object-oriented
- Decoupling of UML ‘activities’ from state machines
- Unification of ‘activities’ with ‘actions’ to provide a complete procedural model.
- Contextual modeling constructs for the internal composition of classes and collaborations.
- Repositioning of components as design constructs and artifacts as physical entities that are deployed.

### 2.2.2 SEMANTICS AND INCONSISTENCIES

An oft-repeated criticism of UML is that it has “no semantics”; that it is primarily a visual notation whose graphical constructs can be interpreted differently. This is because UML is intended to model systems across a broad spectrum of different application domains. An additional consideration related to formal models, is that it is often the case that the same entity may need to be modeled from different viewpoints. This suggests that basing UML on any specific concrete formalism would likely severely hamper one of its primary objectives: to unify a set of broadly applicable modeling mechanisms in a common conceptual framework. This aspect of UML must not be underrated even while defining a formal semantic for UML.

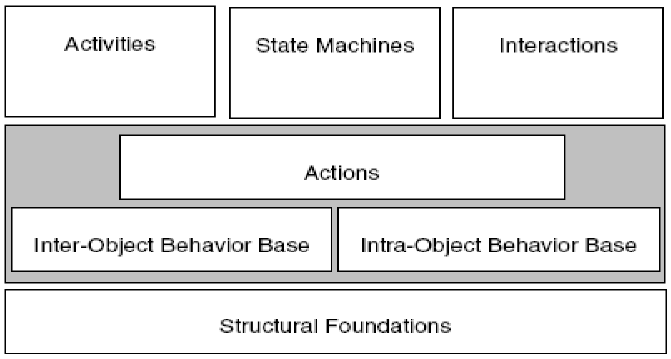


Figure 2.2. The Semantic Architecture of UML2.0 [Source: Selic 2004]

The semantic architecture of UML2, as given in figure 2.2, identifies the key semantics areas covered. It also shows the dependencies that exist among them. At the highest level of abstraction, it is possible to distinguish three distinct layers of semantics. The foundational layer is structural. This reflects the premise that is no disembodied behavior in UML – all of it emanates from the actions of structural entities. The next layer is behavioral and provides the foundation for the semantic description of all higher-level behavioral formalisms. This layer is called the Behavioral Base and consists of three separate sub-areas arranged into two sub-layers. The bottom sub-layer consists of the inter-object behavior base, which deals with how structural entities communicate with each other, and the intra-object behavior base, which addresses the behavior occurring within structural entities. ‘Action’ layer defines the semantics of individual actions. Actions are the fundamental units of behavior in UML and are used to define fine-grained behaviors. Their resolution and expressive power are comparable to the executable instructions in traditional programming languages. Actions in this sub-layer are available to any of the higher-level formalisms to be used for describing detailed behaviors. The topmost layer in the semantics hierarchy defines the semantics of the higher-level behavioral formalisms of UML: activities, state machines, and interactions, dependent on the semantics provided by lower layers.

Though UML2 has prescribed precise semantic foundation at action level, its semantics at higher levels are ambiguous, and inconsistent particularly for specification of concurrency [Ober 2001]. The concurrency is an important issue to tackle when modeling real-time applications which are intrinsically concurrent. For concurrency, UML offers the concept of active object, which is an instance of an active class. According to the UML definition, classes may be either ‘active’ or ‘passive’. The internal concurrency of active objects outcomes from:

- state machine specification: concurrent states of the state machine can be perceived as concurrent threads of control;
- operation executions: concurrent invocations of a same operation may be executed at a same time, leading to concurrent executions of the operation specification;
- action specification: according to the new action semantics definition, the actions contained in an action set may be executed concurrently, unless explicit or causal dependencies constrain their sequencing.

At run-time, the above mechanisms should work together correctly in order to ensure the correct behavior of the model. But, ambiguities, and inconsistencies in UML regarding specification of concurrency are well documented [Ober 2001]. The following inconsistencies are observed.

- UML allows both passive and active objects to own state machines without any constraint. But, a passive object, by not having its own thread of control, does not have a scope for executing the state machine.

- In case of active objects associated with statemachine, it is not clear whether operations and statemachine execute independently.

- The state machine processes events in a sequential way so that all events are queued and processed following a given policy that the user has to define (FIFO, priority based, earliest deadline based, …), even if they are *method calls*. The semantics of classes and operations and that of state machines interfere dramatically in UML, without any clarifications.

### 2.2.3 SEMANTICS DEFINITION APPROACHES

All attempts to define UML semantics can be classified into different orthogonal dimensions described below. One dimension is the level of UML coverage. Many people have been trying to build the semantics of individual diagrams of the UML e.g. on state-machines [Kwon 2000], on collaboration diagrams [Engels 2000], on class diagrams [Evans 1999], on use cases [Overgaard 1998], on activity diagrams [Borger 2000] or just to give formal foundations for action language [Alvarez 2001]. Because all diagrams are only views on one and the same model, the attempts to give semantics for separated UML diagrams fail in producing the right semantics for the entire UML. The combination of statics and dynamics is also given in [Reggio 2000] which considers the problem of defining active classes with associated state-machines. It gives a very fine interleaving semantics for state-machines in terms of transition systems, but the authors do not give precise semantics for state-machines, for event queue handling, and they treat only at UML state-machines without action semantics.

Another coverage level relates to the problems with possible concurrency as well as aspects of objects communication, which have been uncovered in [Reggio 2000] and not addressed in the original UML 1.x documents itself. Such open problems are typical for so called loose semantics introduced in [Hubmann 2002], where the aspects of concurrency and object communication are not fixed to some design decision, but cover different implementations. Such loose semantics is not suitable for formal verification. There are a number of UML modelling and/or verification tools implementing precise semantics by translating UML models to programming language or model checker internal formats ([Lilius 1999, ILogix 2002]). These tools have different limitations on the supported UML features and do not provide formal description of the implemented semantics or it is just technical translations.

Translation approaches define translations from UML diagrams to traditional specification languages (Z [Evans 1998], Object-Z [Kim 1999], CASL [30] etc.). For example, G. Reggio et al. [Reggio 2000] proposed a general schema of the UML semantics by using an extension of the algebraic language CASL for describing individual diagrams (class diagrams and state-machines) and then their semantics are composed to get the semantics of the overall model. Also other UML diagram types have been translated to formal notations, e.g., using Abstract State Machines ([Borger 2000, Ober 2001]). E. Borger et al. [Borger 2000] defined the dynamic semantics of UML in terms of ASM extended by new construct to cover UML state-machine features. The model covers the event-handling and the run-to-completion step, and formalises object interaction by combining control and data flow features. However, the authors did not give a complete solution to solve transition conflicts and it is not clear how firable transitions are selected. The semantics implemented by UML-tool vendors via code generation or model simulation can be also classified to this group of approaches.

## 2.3 RELATED WORKS IN UML

Several research efforts are made to develop precise approaches, both formal and informal, in UML for modeling concurrent, reactive systems. These approaches aimed at resolving inconsistencies in UML semantics as well as defining formal approaches in UML based specification processes.

### 2.3.1 A PATTERN FOR CONCURRENCY IN UML

A formal approach for the description of systems, in which two or more operations may be acting concurrently upon the same object, was proposed [Crichton 2002]. The pattern addresses two common problems – inadequate models, and complicated state diagrams. Changes in attribute state and changes in operation state are described separately using two different types of diagrams i.e. state and activity diagrams. The pattern attempts to present models that characterize every possible sequence of interaction (in terms of the set of actions and events defined in the model). The essence of the pattern can be represented as a fragment of the UML metamodel: a class diagram linking the diagrams and the entities that they represent [figure 2.3].

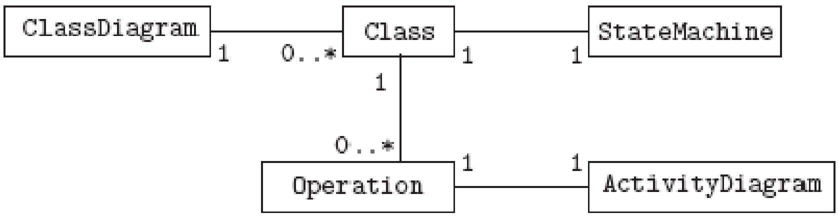


Figure 2.3. A meta-Model for a Concurrency Pattern in UML [Source: Chricton 2002]

In the application of the pattern, (attribute) state diagrams do not make use of call or call* (i.e. asynchronous) actions, due to complication with run-to-completion semantics of UML statemachines, except when the operation is considered atomic i.e. nothing else can happen to the sate of the current object while it is executing. Each operation (non-atomic) is described using UML activity diagram (where the emphasis is upon activity, or flow of control, rather than state). In application of the pattern, the activity diagrams may perform any of the actions send, call, and call* (in addition to other local actions).

The definition and use of the pattern, in figure 2.3, raised several issues regarding the syntax and meta-model of UML (some of which were addressed in later revision of UML [Selic 2004]). 1) local actions of activity diagrams are considered 'atomic'. There effects can be implicitly, rather than explicit action-event pairs, regarded as communication with the underlying object state, and include their effect as 'change' events in the attribute state diagram. 2) it may be required to refer to a particular invocation of an operation: to make explicit the target of a return action, or to describe the effect of an exception. 3) the separation of state, and activity diagrams makes it difficult to represent the effect of exceptions upon operations. Ideally there should be a mechanism to achieve the effect of a try-catch block in Java.

Models with single state diagram impose a sequential execution model due to associated run-to-completion semantics. Using the patter, with the separation of activity diagrams (operations) from state diagram, the effect of interleaving of multiple invocations of operations on the state diagram can be easily represented. Concurrent invocations of operations are best seen as peers, alongside the attribute state of the object. Any attempt to represent them using concurrent substates, within the object state diagram, is likely to produce a confusing inadequate model. Further the pattern identified that behavioral features such as operations do not have classifiers making it impossible to construct an explicit representation of concurrent invocation (rectified in UML2 [Selic 2004]). The pattern points out the complexity associated with the event deferral mechanism in UML statecharts semantics. Although multiple events may be deferred, only one of these will ever be processed; the others will be lost; clearly, in a description of concurrent behavior, this may not be appropriate. The pattern proposes a simple solution; to avoid the use of deferred events, and to include a component within the model whose role is the management and delivery of signal events. Another possible solution would be a persistent version of event queue associated with state diagram that retains the events until they are used to trigger transitions, or are explicitly discarded by the state machine.

The proposed cmUML framework adopts the similar notion of separation of operations from statecharts and separately modeling them using activity diagrams. Further cmUML makes benefit of new behavioral elements in the revised meta-model of UML for e.g. classifier form of behaviors. This helps explicit specification of multiple invocations of operations. cmUML adopts the architectural framework thus providing the 'port' components to handle explicit event management as suggested by the pattern. cmUML further defines an exception handling mechanism in the fashion of try-catch semantics of Java. Compared to the above proposed pattern, cmUML is a comprehensive approach to model complex systems.

### 2.3.2 ATOM-S: A CONCURRENT MODEL IN UML

ATOM-S, a design model based on active/ passive object paradigm, for specifying concurrency in UML was proposed [Ober 1999]. The approach systematically tackles the issues of concurrency in UML object model by integrating ATOM, a well-designed concurrent object model, into UML [Papathomas 1996]. UML object model is compared against the well known classification of concurrent object models given by Papathomas [Papathomas 1992]. The classification uses three dimensions. On the first dimension, object models are divided with respect to what combination of objects they support, into three categories: orthogonal (objects are independent of threads), homogenous (all objects are active), heterogeneous (objects may be active or passive). The second dimension captures the internal concurrency of objects: internally sequential, quasi-concurrent or concurrent. Finally, the third dimension captures the available inter-object communication and synchronization mechanisms: synchronous/asynchronous feature calls, conditional/unconditional acceptance of incoming calls, etc. The position of UML in the design space of concurrent object models, as per the above comparison framework, is described in the table 2.1.

The main drawbacks in the approach taken by UML towards concurrency are identified: active objects are 'sequential', UML semantics says nothing about the situation when multiple concurrent calls are made to the same active object (except when the object has a statemachine), no constraints specified regarding asynchronous calls which could be problematic when the target is a 'passive' object, inconsistency and ambiguity in case of passive objects if associated with a state machine, semantics of method invocation is ambiguous in the case of active objects associated with statemachine for e.g. it is not clear whether the methods of an object are executed by its state machine or the statemachine is manipulated by the methods. It is observed that a cleaner semantics and more powerful primitives are desirable in UML object model towards concurrency.

| Classification Criterion | UML Position | Remarks |
|---|---|---|
| Object Model | Heterogeneous | The definition of active and passive in UML are not same as those used in the classification e.g., passive objects are protected against concurrent calls. |
| Internal Concurrency | Active objects – sequential<br>Passive objects – internally concurrent | The methods of passive objects are split in three classes: sequential, guarded, and concurrent |
| Client/ Server Interaction | One way message passing through signals and asynchronous calls.<br>And RPC | No support for reply scheduling |
| Constructs for accepting requests | Activation conditions supported through state machines | States/ transitions/ guards provide a powerful mechanism for expressing activation conditions, but no semantics for the inheritance of a state machine |

Table 2.1. UML Position in the Design Space of Concurrent Object Models

[Source: Ober 1999]

ATOM-S is the result of the drawbacks of UML object model towards better combination of a concurrent object model (ATOM) and UML statemachine (S). ATOM, with quasi-concurrent active objects and no passive objects, has proven to solve many of the classical problems of object oriented concurrency, and inheritance [Papathomas 1996]. It is a good compromise between expressive power and protection of the integrity against concurrent calls. Special events are designed to indicate when a method call is received, when the execution begins or when it ends. ATOM-S integrates statemachines into ATOM and retains passive objects of UML without statemachines to resolve associated ambiguities w.r.t asynchronous calls. Besides attributes and methods, an active object may have a statemachine that specified 'reactive' part of its behavior and responds to asynchronous one-way simuli i.e. signals that may carry parameters. Further, as in UML, a statemachine may specify the complete behavior of an object or only its protocol, case in which its states are used as 'activation conditions' for the object methods. To solve the ambiguity that exists in UML about the way CallEvents are processed, method invocations sent towards an object do not pass through its state machine and only *signals* are queued and processed one-by-one by the machine. Thus, a method invocation will always result in the execution of the method.

```
Class ThreadScheduler (ActiveObjectSupport):

                methods = [‘InsertThread’, ‘Schedule’, ‘EndThread’, ‘AlertAdmin’,
                                              ‘RecoveryProc’]
                events =[‘Recover’]
                conditions = {
                               ‘InsertThread’ : ‘not self.inState (‘Overloaded’)’
                               ‘Schedule’:’not self.inState (‘Overloaded’)’ }
                def  InsertThread (self, thread, priority): …. method bodies omitted
                statechart = { --- given below graphically }
```

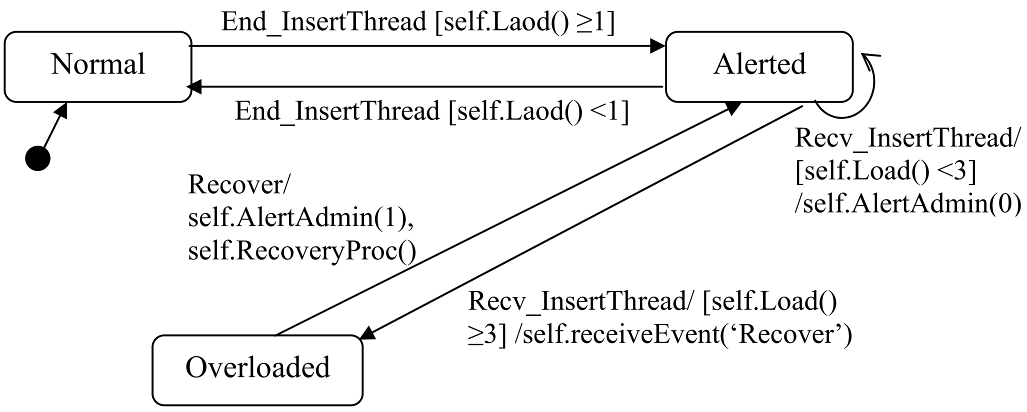

End_InsertThread [self.Laod() ≥1]
Normal
Alerted
End_InsertThread [self.Laod() <1]
Recv_InsertThread/
[self.Load() <3]
/self.AlertAdmin(0)
Recover/
self.AlertAdmin(1),
self.RecoveryProc()
Recv_InsertThread/ [self.Load()
≥3] /self.receiveEvent(‘Recover’)
Overloaded

Figure 2.4. A Simple Thread Scheduler in ATOM-S in Extended Python

[Source: Ober 1999]

The operational semantics of state machines in ATOM-S is based on the fact that the state machine of an active object runs quasi-concurrently with its methods. The state machine of an object is notified when a message call is received, when a method starts executing or finishes. Note that, like in ATOM, the moment when the method is received may not coincide with the moment when it starts executing. The introduction of these implicit messages augments the expressive power. The features of ATOM-S can be exemplified using a thread scheduler class (given in figure 2.4).

The cmUML framework of the thesis shares certain operational semantics of ATOM-S: separation of method calls from statechart of an object, special events to notify the statechart regarding the execution status of the invoked methods, etc. But, being a specification language cmUML does not use the notion of active/ passive design paradigm i.e. both active or passive 'kind' of objects can specify internal concurrency as well as reactive behavior (left to the specifier). Further cmUML is a comprehensive formal approach using live sequence charts as part of component interface specifications.

### 2.3.3 SPECIFYING CONCURRENT SYSTEM BEHAVIOR IN UML/ OCL

Shane Sendall and Alfred Strohmeier proposed a UML based approach in specifying concurrent, reactive behaviors and timing constraints using OCL [Sendall 2001, Kleppe 1999]. Recognizing the UML's limited support for specification of timing constraints and mechanisms for synchronization of concurrent activities, the approach extends the operation schemas in OCL i.e. pre, post assertions with constructs for specifying timing constraints, and asserting synchronization on shared resources. The approach has three principal views:

- a model composed of descriptions of the effects caused by operations, which uses pre- and postcondition assertions written in OCL, called operation schemas;
- a model of the allowable temporal ordering of operations, called the system interface protocol (SIP); and
- a model that describes the system state used in the operation schemas, called the analysis class model (ACM).

The analysis class model (ACM) represents all the domain concepts and relationships between them, the combination of which provides an abstract model of the state space of the system and defines the system boundary. This model is used as the basis for writing operation schemas, i.e., pre- and postcondition assertions for each system operation. The System Interface Protocol (SIP) defines the temporal ordering of system operations, one aspect of the behavior model of the system. SIP is described with a UML state diagram. A transition in the SIP is triggered by an input event only if the SIP is in a state to receive it, i.e., there exists an arc with the input event and the guard evaluates to true. If not, the input event that would otherwise trigger an operation is ignored. The SIP for sequential and trivial systems can normally be described with a single statemachine. We have made the observation, however, that concurrent systems are better described with multiple views, one view per perspective on the concurrency.

An operation schema declaratively describes the effect of the operation on a conceptual state representation of the system and by events sent to the outside world. It describes the *assumed* initial state by a precondition, and the required change in system state after the execution of the operation by a postcondition, written in UML's OCL formalism [Kleppe 1999]. The syntax of operation schemas consists of several clauses e.g. Description, Use Cases, Scope, Declares, Sends, Pre, Post etc to convey the semantics of the operation behavior. To highlight the constraint on shared resources in presence of concurrency, a clause 'Shared' is added in operation schemas to imply that the resources listed in this clause are constrained to be updated in mutual exclusion by the operation execution. Further as pre, post conditions are not sufficient in presence of concurrency suffixes @preAU, @postAU are added to shared variables to refer to the state of the resource immediately before and after an 'atomic update' by the operation. Further the suffix @rd indicates the consistent value of a shared resource without parallel updates within a specific period e.g. operation execution. Semantics of branch condition if-then-else assumes atomicity to avoid race conditions. The 'rely' block states a condition that must be true immediately before, immediately after, and during the execution of the body of the block for the body to take effect. If the rely condition does not stay true throughout execution, then the effect of the *fail* part of the rely block is observed to execute instead. The 'rely' block imposes neither immediate nor wait semantics on the condition. All it requires is that if the condition remains true, then the effect described by the body of the block will hold.

Though the approach described above provides a precise specification of concurrency and timing aspects, it is not well integrated with development process of systems. Tts heavy dependence on OCL may not appeal to ordinary developers and non-OCL specifiers. It also requires a complete class model to precisely define operation schemats. Further, the focus of the concurrency is at higher conceptual level, more suitable to describing distributed application behavior, and does not handle issues of concurrent operations and behavioral statemachines which are central to reactive systems. Additionally a multi-view SIP requires rules for composition, completeness, consistency, etc to form the ensemble from the different SIP views, a complex process. However the operation schema approach and semantics of related constructs in describing effects of concurrent operations is similar to that of cmUML specification framework i.e. activity specification of component services.

### 2.3.4 SDL/ UML AND UML-RT

ITU-T defines a one-to-one mapping between a subset of SDL and a specialized subset of UML. With this mapping it is possible to use UML (for multiple views of the same system, informal object models) and SDL (for detailed and formalised object models, with execution semantics) [ITU 1994, ITU 2000, Glasser 1997]. The main differences between UML and SDL are that

- UML is a collection of concepts and notations for several 'views' of the same system: e.g. Object-, State Machine-, Use Case-, Collaboration and Interaction

- SDL is a language (with concepts, abstract grammar and graphical/textual grammars) focusing on the Object- and State Machine views of a system. For these views, SDL is however a complete language with static and dynamic semantics and with concrete syntax (graphical/textual) for the specification of actions. Users of SDL rely on other languages like MSC for specification of interactions between instances.

- UML has a weak semantics with many variation points, while SDL has a complete semantics, including execution semantics for state machines.

An SDL System consists of Agents that are connected by means of Channels. Agents may communicate by sending Signals or by requesting other Agents to perform Procedures. An Agent may have both a StateMachine and an internal structure of Agents (a composite Agent). The internal Agents and the StateMachine are connected by Channels. The connection points for Channels are Gates. Agents come in different *kinds* with different execution semantics: Block Agents are *concurrent* Agents with possibly interleaved execution of the transitions of the state machines, while Process Agents are *alternating* Agents with run-to-completion execution of transitions. The overall system is a special System Block Agent.

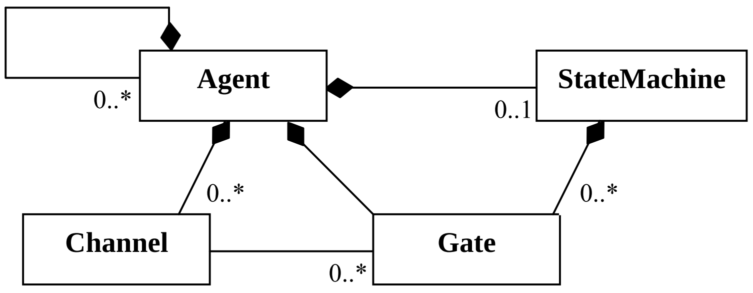


Figure 2.5. UML Conceptual Model of basic SDL Concepts and Relationships
[Source: Glasser 1997]

In SDL, internal structure of agents takes explicit form in terms of interfaces and gates. An Interface defines Signals, Variables, RemoteProcedures and Exceptions. Interfaces are associated with Gates. Gates are connection points for Channels connecting Agents. Communication between Agents takes place via Channels. Gates are mapped to UML by means of a combination of interfaces and associations to other classes. Statemachines of ‘block’ agents execute concurrently (i.e. interleaving semantics) where as ‘process’ agent statemachines (with in a ‘block’ agent) executes in run-to-completion steps. SDL statemachine transitions are triggered by events like input of a Signal or a remote Procedure call.

UML-RT is similar to SDL/UML except it defines only one type of ‘active’ system entity called ‘capsule’ and also defines ‘ports’ and ‘interfaces’ [Selic 1998]. In terms of composition, system entities, and system architecture the proposed cmUML framework is similar to SDL/UML (or UML-RT). But being behavioral specification language, there is no emphasis on static aspects of associations, classes etc in cmUML. Further external and internal associations are similar (take implicit form i.e. no notion of channels and gates). Further the focus of ‘interface specification’ in UML is different as it is an abstract form of more detailed ‘internal specification’ and meant for certain kinds of verification purposes. SDL/UML and UML-RT are heavily dependent on statemachine formalism (where transitions are triggered by both signals and procedure calls) where as cmUML leaves the choice of combinations of both statemachine and flow diagrams to the specifier giving more expressive power and flexibility (in cmUML operations are delinked from statemachine by default). cmUML integrates sequence diagrams into specifications with liveness constraints on the execution behaviors of system entities. SDL/UML and UML-RT target designs with precise operational semantics.

### 2.3.4 KRTUML

krtUML, a subset of UML, is rich enough to express all behavioural modelling entities of UML used for real-time applications, covering such aspects as concurrency and communication [Damm 2002]. A formal interleaving semantics for this kernel language is defined by associating with each model M of krtUML a symbolic transition system STS(M) [Manna 1991]. This provides the semantical foundation for formal verification of real-time UML models [Damm 2003]. In this semantic framework, the state-space of the transition system is given by valuations of a set of typed system variables, and initial states and the transition relation are defined using first-order predicate logic. A complete snapshot of the dynamic execution state of a UML model is captured using unbounded arrays of object configurations to maintain the current status of all objects, and a pending request table modeling the status of all submitted, but not yet served operation calls. Object configurations include information on the valuation of the object's attributes, the state configuration of its state-machine, as well as the pending events collected in an event-queue. Central to a krtUML specification are ‘krtUML omponents’ where a component is a collection of an active object and a set of passive objects. With in a component all passive objects delegate their event handling to corresponding active object. An active object is like an event-driven task which processes its incoming requests in a first-in-first-out fashion. It comes equipped with a dispatcher, which picks the top-level event for the event-queue, and dispatches it for processing to either its own state-machine, or to one of the passive reactive objects associated with this active object, inducing a so-called run-to-completion step. ‘Triggered operations’ i.e. operation calls, whose return value depends on the current state of the system, are distinguished from ‘primitive operations’. For triggered operations the willingness of the object to accept a particular operation call in a given state is expressed within the corresponding statemachine.

krtUML semantics is defined using symbolic transition systems. A symbolic transition system (STS) $S = (V;\theta;\ \rho)$ consists of V, a finite set of typed system variables, $\theta$, a first-order predicate over variables in V characterizing the initial states, and $\rho$, a transition predicate, that is a _rst-order predicate over V; V 0, referring to both primed and unprimed versions of the system variables (their current and next states)

krtUML is targeted for hard real-time systems as the corresponding semantics enforces that at most a single thread of control is active within one component. cmUML adopts the semantical definition framework of krtUML but its components possess heterogeneous semantics as they correspond to different kinds of behaviors e.g. finite statemachine, flow diagram, or different combination of both. Further as cmUML is not tied to a specific application domain the semantic foundation is more generic and flexible. cmUML adopts more structured ADL approach towards composition of systems in terms of cmUML components. Both the frameworks aim towards LSC based formal verification [Damm 2003].

## 2.4 RESEARCH GAPS AND PROBLEM DEFINITION

A formal specification framework in UML towards specification phase in the development of systems with concurrent and reactive behaviors is required. The principles and semantics of formal specification approaches for concurrent and reactive systems as described in [Wing 1990, Lamport 2000] can be adopted. It is required to clarify inconsistencies and ambiguities in the usage and semantics of UML diagrams through an integrating framework based on higher level conceptual semantics. The standard UML/ SPT Profile [OMG 2002] can be used as the foundation for the proposed frameworks as the profile defines the concepts of events, causality, concurrency, and resource without formal semantics. The framework should form a basis for explicit introduction of concurrency in UML. Further it should be independent of implementation level primitives and associated semantics.

A unifying framework for UML diagrams e.g. activity, state, and sequence with precise conceptual semantics and well defined context is required. The formal semantics should be defined in an intuitive way without extensive use of OCL [Kleppe 1999], the standard constraint language of UML. Further the proposed framework of the thesis should also integrate principles and techniques of formal methods and those of ADL community towards compositional specifications in terms of components, interfaces, ports etc. As UML does not define a development process and leave it to the profile developers as required for the domain under consideration, a suitable specification process for use of the framework can be defined. Further the framework can be validated against the current approaches [e.g. Goni 2004] in UML using classical problems of concurrency patterns, or example, or case studies or all of them.

The specification framework should adopt the separation-of-concerns approach in specification of system components through separation of interface and internal specifications where the latter is more detailed version of previous. Further suitable semantic foundation for the proposed framework should be investigated. The semantics should handle the liveness issues of executions of the system. For this, the formalism of

LSCs i.e. live sequence charts [Damm 1999] can be integrated in to the proposed framework. The research work can further explore the suitable formal verification techniques e.g. model checking, industry tools, ADL tools with formal basis for integration with the proposed framework.

Thus the framework should provide explicit, precise, yet intuitive means for specification of concurrency, reactivity, exception handling, synchronization features which are externally visible, and hence verifiable, of the system components.

## 2.5 SUMMARY

This chapter provided the background and literature survey which forms the foundation of the thesis work carried out. Formal specification approaches, overview of UML and related issues are presented. Further related works in UML are described. The chapter concludes with the research gaps identified through literature survey and the formulated research problem for the thesis work.

# CHAPTER 3
# FRAMEWORK DEFINITION

UML has defined an important profile, known as Profile for Schedulability, Performance, and Time (UML/SPT), for modeling Real-Time systems. In an attempt to provide a flexible open framework towards the exchange of models, the SPT profile has not defined a formal semantics. This limits the understanding of semantics of the specifications. A better approach to providing a open framework together with semantically rich specifications is by describing the relevant semantic information at a higher level of abstraction. The thesis defines such a framework over concepts of SPT profile. This is achieved by adding an abstract specification layer with precise conceptual semantics, on top of the UML/SPT profile, towards explicit specification of concurrency, reactivity, exception handling, and synchronization.

This chapter presents the first part of the thesis contribution i.e., the definition of a UML based specification framework, namely cmUML, and associated specification language, namely cmUML Profile. The profile is based on the standard light weight extension mechanisms of UML. The framework defines higher level abstractions with associated conceptual semantics as necessary building blocks for specification of system components. In this chapter, the relevant features of the SPT profile are described. Then, a conceptual model of the framework is presented. A mapping strategy between the elements of the conceptual model and those of SPT profile and UML meta model is defined. The chapter concludes with an informal description of behavior and semantics of the proposed specification constructs of the proposed cmUML framework.

## 3.1 UML SPT PROFILE

UML SPT profile offers a common framework for real-time modeling that unifies the diversity of techniques, terminologies and notations existing in the real-time software community, while still leaving space for different kinds of specifications [OMG 2002]. In its current form, the main focus of the SPT profile is on time and time-related concepts. It offers a terminology for modeling real-time systems: defines a set of concepts and some relationships between these concepts, as allowed by the meta-modeling technique used for the definition of the profile. Clearly, the meta-modeling techniques can carry only superficial semantic information. At a first sight this may be argued by its aim to address the needs of various real-time modeling techniques. However, a closer look to the SPT definition itself shows that such a definition is insufficient, in particular for promoting common understanding of specification and exchange of specifications between tools [Ober 2004].

The approach used in SPT to deal with the variability of concepts is to add attributes in the form of keywords and to abandon the idea of fixing a semantics (left to the tools). Indeed, fixing semantics has the inconvenience that there will always be some domains in which slightly different concepts are needed. The approach of the SPT was to provide an answer to the main question: *how to provide a flexible and relatively open framework and still be able to exchange models with their semantics?* But, the better alternative approach in providing open framework with semantics is to provide a standard way to describe semantic information at a higher level of abstraction [Ober 2004]. For the general dynamic semantics, the main issues are the choice of the granularity and communication and execution mechanisms i.e., the possible choices between several concurrently enabled steps and the granularity of the observed steps. The number of reasonable communication modes is relatively small, and in particular in the context of SPT, an effort can be made to provide attributes with widely accepted interpretation.

### 3.1.1 GENERAL RESOURCE MODELING FRAMEWORK

The SPT Profile is partitioned into a number of packages dedicated to specific aspects of real-time system modeling and analysis techniques. At the core of the profile is a set of sub-profiles that represent the general resource modeling framework. This provides a common base for all the sub-profiles. However, it is anticipated that future profiles may need to reuse only a portion of this core. Hence, the general resource model is itself partitioned into three separate parts (figure 3.1).

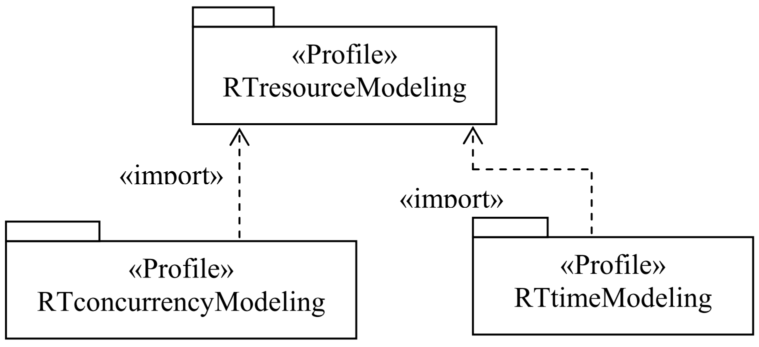


Figure 3.1 General Resource Modeling Framework Packages of SPT Profile
[Source: OMG 2002]

The core resource modeling framework is further divided into sub packages (figure 3.2). Of these packages **CoreResourceModel** package forms the core part and defines the abstract concept of a *Resource*, *ResourceService* and *Instances*. Because concurrent, reactive systems are best specified in terms of instances of their behavior parts, the rest of this description will focus on the behavioral entities corresponding to proposed cmUML constructs.

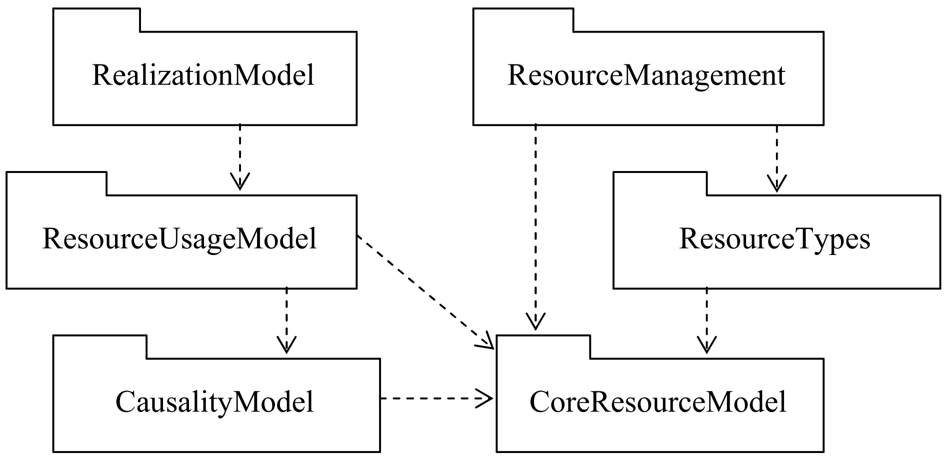


Figure 3.2. Sub packages of Resource Modeling Package of SPT Profile
[Source: OMG 2002]

Of all the sub packages of SPT profile the packages **CoreResourceModel**, **CausalityModel**, and **RTconcurrencyModelling** form necessary elements for defining the proposed cmUML specification framework.

### 3.1.2 CAUSALITY SUB-PACKAGE

This is an important model that is used as a basis for any dynamic modeling associated with the SPT profile. It captures the essentials of the cause-effect chains in the behavior of run-time instances of the model. The model is based on the dynamic semantics of UML but is more detailed and precise.

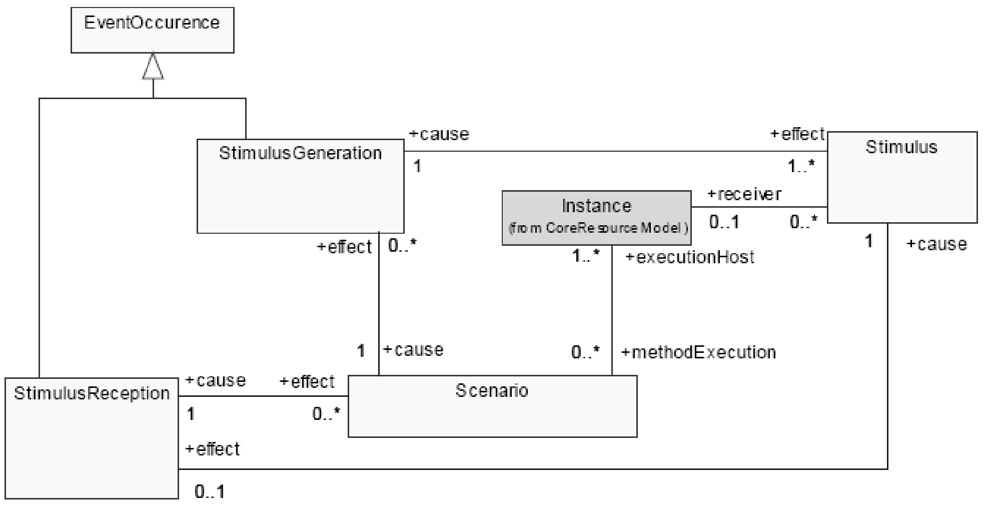


Figure 3.3. The basic Causal Loop Model in UML [Source: OMG 2002]

A fundamental concept in the causality model is the notion of an *EventOccurrence*. This corresponds to an instance of the UML *event* notion. There are many different kinds of event occurrences, but the most interesting ones are the *StimulusGgeneration* and *StimulusReception* events. A *stimulus* is an instance of a communication in transit between a calling object and a called object. A stimulus generation event occurs when an object executes an action that invokes an operation on another object (the *receiver*) or sends a signal to it. The effect of the stimulus generation event is the creation and dispatching of a stimulus that identifies the parameters of the communication (the operation invoked, the values of the parameters, etc.). The stimulus will eventually result in a stimulus reception event. This event occurs when an object executes some kind of reception operation. The occurrence of this event will either trigger a transition in the receiver or result in the execution of a method (**details of how the event is received, scheduled, and dispatched can be defined as part of the formal semantics of the sublanguage, here it is cmUML**). This, in turn, leads to a scenario execution (or simply, *scenario*). A scenario execution may result in the execution of an ordered set of actions, some of which may generate further stimuli, and so on.

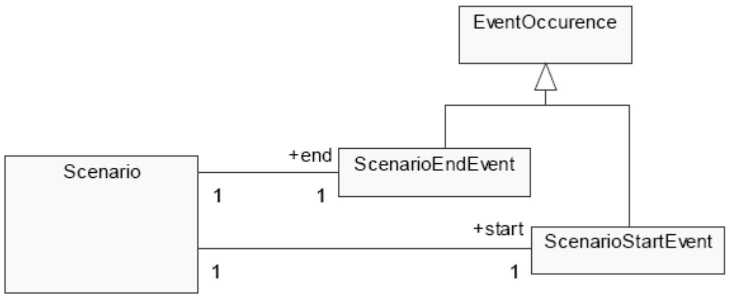


Figure 3.4. Scenario Event Occurrences [Source: OMG 2002]

### 3.1.3 CONCURRENCY SUB-PACKAGE

The general concurrency model in SPT profile is based on its causality model (as described in previous section). As ‘actions’ (which are parts of scenarios) execute, they generate ‘stimuli’. In the concurrency model we specifically identify so-called *message actions*. These are action executions that generate one or more stimuli. Following the standard causal loop, a stimulus targets a particular service instance of a specific object instance. This causes the execution of the scenario corresponding to the method associated with the resource service instance. This leads to further action executions, and so on. For the concurrency model, of particular interest is the notion of a *ConcurrentUnit*, an ‘active’ resource instance that executes concurrently with other concurrent units.

Ultimately, *all behavior in the system is a consequence of actions executed by concurrent units*. Following creation, each concurrent unit commences to execute one *main* scenario. This scenario executes until the concurrent unit is terminated. During its execution, the *scenario may perform explicit receive actions* in order to accept any stimuli sent to it. A receive action by a concurrent unit leads directly to the activation of the appropriate service instance and its service method. During the execution of the service method the main method may either be blocked, the so-called *run-to-completion* step, or it may proceed executing concurrently. Of course, a stimulus may arrive before the targeted concurrent object is ready to receive it. In such situations it may be necessary to defer the response to the stimulus until the corresponding 'receive' action is executed. For this reason, a concurrent unit needs one or more *queues* for holding deferred stimuli (multiple queues may be used to differentiate between stimuli of different priorities or sources). There are two choices at either end of the communication, which affect the detailed causality between concurrent threads of control. At the server end, the service request may either be handled *immediately*, or *deferred*. In the immediate case, a further property describes whether the receiving instance creates its own concurrent execution thread to handle the service request (the so-called *local* option), or assumes that there is an existing thread available (the *remote* option). At the receiver end, the message action may either represent an *asynchronous* or *synchronous invocation* of the service. If the request is asynchronous, then execution proceeds immediately; if the request is synchronous, then the client is blocked until a response is received from the receiver. Instances that are not concurrent do not have a main method and, hence, have no direct choice in controlling how a service request is handled. The concurrency model of SPT profile is described in figure 3.5.

## 3.2 EXTENSION MECHANISMS IN UML

The UML profile mechanism provides a way of specialization of the concepts defined in the UML standard. A stereotype can be viewed as a subclass of an existing UML concept. Most domain concepts (e.g. cmUML concepts, which in turn are based on UML SPT and UML metamodel) map directly into a stereotype with additional attributes of such concepts, specified using appropriately typed tags for each attribute. However, the domain model often shows associations between domain concepts, and, since the UML extension mechanisms do not provide a convenient facility for specifying new associations in its metamodel, such domain associations have to be represented in a variety of different ways, depending on the case at hand. The following three general techniques are used to capture associations between domain elements:

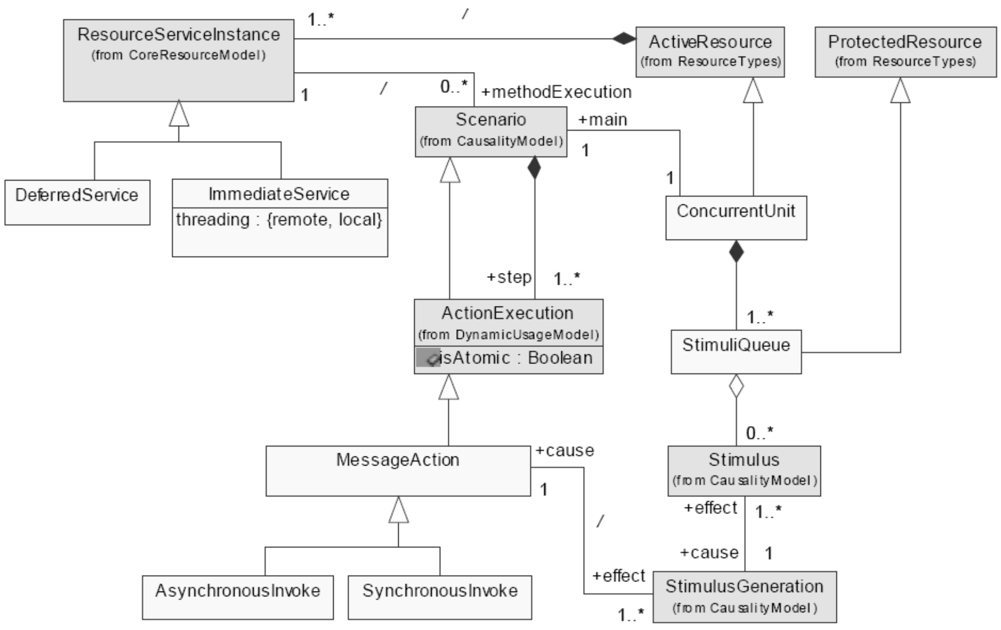


Figure 3.5. General Concurrency Modeling Concepts in SPT Profile
[Source: OMG 2002]

- Some domain associations map directly to existing associations in the metamodel.
- Some domain composition associations map to tags associated with the stereotype.
- In few cases, a domain association is represented by using the <<taggedValue>> relationship provided by the UML profile mechanisms.

The concept of an action execution figures prominently in the SPT profile. This represents the run-time execution of some action (the UML metamodel does not provide such a concept). The UML notation definition document finesses over this by mapping activation to the action whose execution is represented. Unfortunately, this is not adequate for specification of instance behaviors. Sometimes it is necessary to differentiate between an action, from an instance of the *execution* of that action. What is crucial is that different executions of the same action specification are different. Therefore, what is required is an extension to the UML metamodel Common Behavior package. The new concept, called *ActionExecution*, is integrated into the current metamodel as indicated in figure 3.5.

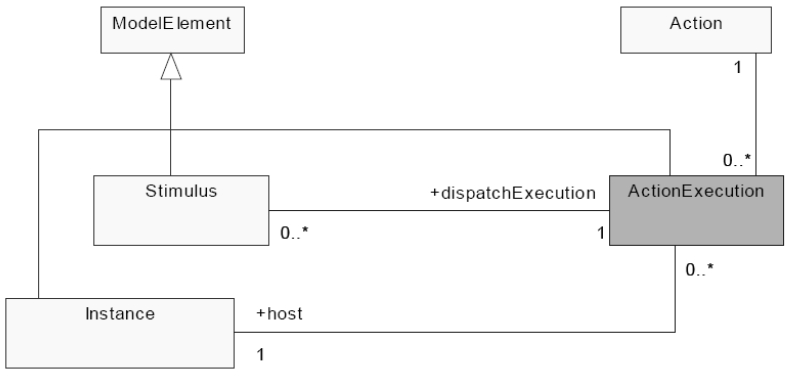


Figure 3.6. Proposed UML metamodel Extension in SPT Profile (ActionExecution) [Source: OMG 2002]

## 3.3 CONCEPTUAL MODEL OF cmUML

SPT profile approach is followed in defining the conceptual framework and the corresponding UML mapping: first conceptual elements of the framework are introduced in a class diagram notation (not related to UML metamodel) and then mapped onto UML metamodel using standard extension mechanisms *stereotypes*, *tags*, and *constraints*. In cmUML, a 'component' (different from UML components) is a generic entity representing the type or 'descriptor' of corresponding runtime behavior instances. The specialized components are defined with specific functionality and behavior specified in terms of actions or activities combined in reactive or flow semantics. Further these components may be 'concurrent' or 'sequential'. The internal concurrency is specified in two ways as *interleaved executions* or *alternating executions* in run-to-completion semantics.

The conceptual model of proposed cmUML specification framework is presented in figure 3.7. The core abstractions are represented in the conceptual model. Based on the functionality and related conceptual semantics (as explained in detail later in this chapter), the components are further classified as *system, state, port, service*, and *resource*.

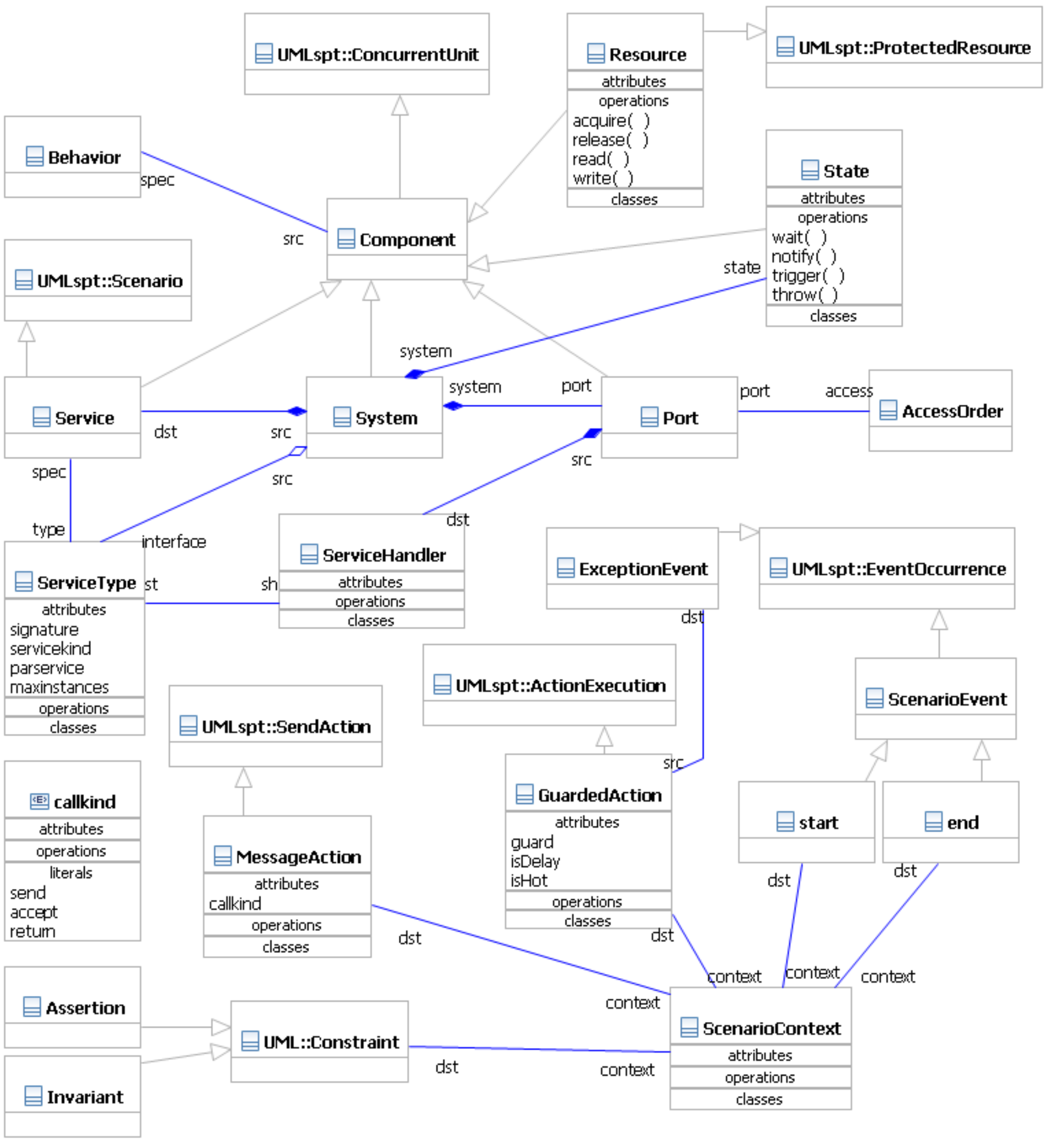


Figure 3.7. Conceptual Model of cmUML Framework

A *system* component contains other components and responsible for their initialization. Intuitively a system defines the scope of containing behaviors and represents a single cohesive behavioral unit suitable for compositional specification, verification, analysis as well as synthesis (not all of them are within the scope of this thesis). It may further contain other *system* components hierarchically.

*Resource* components with abstract operations *acquire(), release(), read() and write()* represent passive, protected data or hardware resources. *Resource* components with complex concurrent behaviors may be specified as *system* components. *State* represents *reactive, synchronization,* and *exception handling semantics* of internal executions. *Port* represents interface specification with concurrency aspects, service access order, and inter-component communication. *State* component can be considered as an extended form of *Port* behavior with sub states and sub transitions corresponding to an abstract implementation behavior (this may be comparable to so-called *stuttered* transitions in Lamport's transition axiom method approach [Lamport 1989]).

*Service* components represent dynamically created 'sequential' executions in interleaved or run-to-completion steps in response to external events (in contrast to asynchronously executing *State* and *Port* behaviors). An instance of a *Service* may execute concurrently with itself and other compatible services as specified in corresponding *Port* specification. Action and activities are *simple* or *guarded* (with precise semantics regarding atomicity, synchronization, and exception handling). The *guard* conditions are local *assertions* or global *invariants* representing synchronizations. Failure of guards associated with a guarded action or activity may result in either wait semantics or a raised exception causing the termination of the corresponding activity (similar to the java's try-catch block []).

cmUML framework defines 'conditions' as first-class entities facilitating precise specification behaviors at interface level (comparable to *conditions* in LSC semantic framework [Damm 1999]). Further the interface 'conditions' and related semantics correspond to those of internal behaviors e.g. *guarded* actions. Thus cmUML provides consistent specifications both at interface and internal specification level. A *ScenarioContext* represents interactions of component executions with liveness semantics, in response to external events. These contexts correspond to component use-cases and are triggered by specific events or sequence of events.

## 3.4 DEFINITION OF cmUML PROFILE

The goal of the proposed stereotypes is to provide specification constructs with conceptual semantics towards precise behavioral specification of concurrent, reactive systems hierarchically. Further the proposed language integrates various modeling formalisms of UML (i.e., statechart, activity, sequence diagrams) with concurrency semantics of underlying object model and the action semantics of UML. The UML profile, corresponding to the conceptual model of cmUML, is defined using UML standard extension mechanisms *stereotypes, tags* and *constraints*.

The *cmUML* profile uses flat structures of behavioral specifications i.e., activities, statecharts, and sequence diagrams without hierarchy as the latter can be easily translated into equivalent flat versions. Also a few meta-level *abstract* methods are defined for some abstractions (e.g., state, resource etc) to simplify the semantics description and make the specifications intuitive to system or tool developers. The design rationale behind the proposed stereotypes of cmUML is described below by related mapping on to elements of UML metamodel and the UML/SPT profile. This also explains the related *tag-value* pairs and cosntraints of the corresponding stereotypes (table 3.1-3.3).

***Component***: a generic behavioral specification unit in cmUML. Associated with *ConcurrentUnit* in UML/SPT profile, it represents a concurrent activity. As *behaviors in UML* can take *classifier* form [Selic 2004], it is also associated with UML metamodel element *Class* (with *isActive=true*). This mapping is further consistent with general notion of ‘threads’ associated with an active object in design paralance (but cmUML is independent of design notion of ‘threads’). Thus behavors specified as cmUML abstractions may be regarded as ‘Active’ in generic specification interpretation.

| **cmUML Stereotype** | **UML/ SPT Concepts** | **UML Metamodel Element** |
|---|---|---|
| Component | Descriptor, ConcurrentUnit | Class [isActive = 'true'] |
| System | Specialization: Component | |
| Port | Specialization: Component | |
| State | Specialization: Component | |
| Resource | ProtectedResource | Specialization: Component |
| Service | Scenario | Specialization: Component |
| ServiceType (ST) | | Operation |
| ServiceHandler (SH) | | Classifier |
| ActivityExecution (AE) | ActionExecution | |
| ScenarioExecution (SE) | Specialization: ActionExecution | |
| GuardedAction (GA) | | Action |
| MessageAction (MA) | SendAction | |
| Exception | | Stimulus |
| ScenarioEvent | EventOccurence | Event |
| Start | Specialization: ScenarioEvent | |
| End | Specialization: ScenarioEvent | |
| AccessOrder (AO) | | (Protocol)StateMachine |
| Reactive | | StateMachineDiagram |
| Flow | | ActivityDiagram |
| ScenarioContext (SC) | | SequenceDiagram |
| PrimaryContext (PC) | Specialization: ScenarioContext | |
| SecondaryContext (SC) | Specialization: ScenarioContext | |
| Condition | | Static Feature |
| Assertion | | Specialization: Condition |
| Invariant | | Specialization: Condition |

Table 3.1. Stereotypes of cmUML and the mapping into UML and UML/SPT Profile

**Specializations of *Component***: taking into consideration the specification needs, several specializations of *Component* are proposed. *State* represents a behavior with reactive semantics. *Port* represents an interface specification and its abstract behavior for e.g. the interface concurrency control, global synchronization patterns. *Resource* represents a 'passive' entity with possible 'internal' concurrency. A resource entity may not be specified further i.e., decomposed in terms of internal specification entities. *Services* represent *dynamically* created behavior executions in response to external events (i.e. service requests). *Service* components are associated with static descriptors defined as *ServiceTypes* (specification of paramerters, reutrn values, pre-condition, post-condition etc) and dynamic descriptors defined as *ServiceHandlers* (with incarnation counters corresponding to each *ServiceType*). Thus service handlers represent the dynamic information of service instances, corresponding to a specific ServiceType i.e., the necessary information required of the executing status of intances of a service type. The *ServiceHandlers* are analogous to implementation level system variables like call stack, program counter etc which represent important information of executions that can be used to specify execution behaviors at higher level without suffering from any implementation bias [Lamport 2000].

**Specializations of *Action* and *ActionExecution***: an 'action' in UML represents a fundamental unit of execution without precise behavioral semantics. cmUML proposes *GuardedActions* as an extension of basic actions with concurrency semantics of *atomicity, synchronziation,* and *exception handling*. *GaurdedAction* precisely conveys the execution semantics of corresponding actions as defined in UML (not compromising the general action semantics defined in UML). SPT profile has proposed the concept of *ActionExecution* towards the metamodel extension of UML. This helps distinguish different executions of an action which is in perticular useful to specify concurrent executions of same action, or service type. As 'activity' is basically defined as an 'action' or a sequence of actions at UML metamodel, cmUML extends the *ActionExecution* to more specific concepts of behavior i.e., instances of executions e.g. *ActivityExecution* and *ServiceExecution* corresponding to various instances of methods of 'operations'

(*ServiceTypes* in cmUML). With this construct it is possible to refer, distinguish and specify the different invocations of component services and their execution status either in interface or internal specifications.

**Specialization of *EventOccurence***: executions in concurrent, ractive systems are related to causality of actions. UML has defined certain kinds of events e.g. callEvent, changeEvent, etc. As these events are not sufficient to specify exact start, and termination of behavior executions, cmUML identifies two special events *Start*, End corresponding to start, and termination of every instance of an 'ActivityExecution' or 'ServiceExecution'. These events help specify the necessary synchronization behavior (e.g. in interface specification) as well as flexibility in defining the semantics of the execution models (e.g. for internal specifications).

**Specialization of UML *feature***: As defined in UML metamodel, a ***classifier*** can be associated with various kinds of ***features*** e.g., static and behavior. cmUML extends the *StaticFeature* to define 'conditions' as first-class entities that can be associated with behaviors. As identified in LSC semantic framework, conditions as first class entities are useful for specification of externally observable behaviors. Further these 'conditions' can be associated with specific semantics and interpretations having a bearing on the acceptable 'runs' i.e., executions of the system components. cmUML further distinguishes two kinds of *conditions* i.e., *assertion* and *invariant*. An *assertion* corresponds to a condition based on local instance variables where an *invariant* may contain other sytem level variables e.g. attributes of *ServiceHandlers* (this may be useful to specify various kinds of synchronziation patterns [Mizuno 1999, Jagadish2006(a)]. Further a *condition* is associated with the tag 'isDelay' which specifies the *wait semantics* in addition to optional (*cold*), or mandatory (*hot*) semantics as defined in LSC semantic framework [Damm 1999].

Table 3.1 lists the stereotypes defined in cmUML extension. UML name as tag type in the table indicates reference to the corresponding instance. Also absence of multiplicity indicates 0 or 1 where as * indicates 0 or more.

| **Stereotypes** | **Associated Tags** (if any):<br>Name [Type] {values} (multiplicity) |
|---|---|
| Component | spec[Behavior](*); root[«system»];<br>concurrencyKind={*concurrent*, *sequential*};<br>evBuffer[«resource»] |
| System | port[«port»]; state[«state»];<br>service[«service»](*); |
| Port | interface[«serviceType»](*);<br>spec[«AcessOrder», «ScenarioContext»(*)];<br>handles[«serviceHandler»](*);<br>policy={FIFO, Priority} |
| State | spec [«Reactive»]; |
| Service | spec [«Flow»]; |
| ServiceType (ST) | max[integer]; serviceKind={*read, write*};<br>parService[«serviceType»](*); |
| ServiceHandler (SH) | execs[«service»](*);<br>in(integer); out(integer); |
| GuardedAction (GA) | guard[boolean]; isDelay[boolean];<br>isHot[boolean]; exception[«exception»];<br>isAtomic[boolean]; |
| MessageAction (MA) | synchKind={send, accept, return} |
| AccessOrder (AO) | scope: {local, global} |
| Condition | isDelay[boolean] |
| PrimaryContext | secondary[«SecondaryContext»](*) |
| SecondaryContext | primary[«PrimaryContext»](*) |

Table 3.2. Tags for Proposed Stereotypes of cmUML

| **Stereotype** | **Associated Constraint** |
|---|---|
| Component | Abstract |
| System | *port*, *state* are not 'null' |
| Port | concurrencyKind = 'sequential';<br>*port*, and *state* are 'null' |
| State | concurrencyKind = 'sequential' ;<br>*port* and *state* are 'null' |
| Service | concurrencyKind = 'sequential'<br>*port* and *state* refer to those of 'root' |
| Resource | To be atomically acquired and released |

Table 3.3. Constraints Associated with the Stereotypes of cmUML

## 3.5 INFORMAL SEMANTICS OF cmUML PROFILE

This section describes the informal semantics of the cmUML constructs as defined in the profile in previous section (formal semantics is described in the following chapter).

One of the important constructs defined in the profile is *ActivityExecution*, as a generalization of SPT Profile's *ActionExecution*, consistent with UML definition of activity as action. Activities are at a higher granularity than actions and can represent an instance of a *Service* corresponding to a *ServiceType*. A service instance is associated with a run-time handler *ServiceHandler* (analogous to run-time system variables, call stack, program counter etc) with dynamic information regarding service instances that (using incarnation counters 'in' and 'out'). This information can be used to specify complex synchronization patterns in the form of global invariants representing safety conditions in a simpler way [Jagadish 2007, Mizuno 1999]. A set of useful global invariants were proposed by Mizuno which work as basic patterns to compose appropriate global invariants for specifications. Translations exist from global invariant based coarse-grained specifications to fine-grained synchronization code using semaphore, monitors etc.

Another important construct defined in the profile with respect to concurrent semantics is *GuardedAction*. This allows specifying precise semantics corresponding to the execution of the corresponding action or activity. The *GuardedAction* specifies synchronization and exception handling aspects through specified tags 'guard', 'isDelay', 'isHot', 'Exception', and 'isAtomic'. Various combinations of these tag values specify the precise semantics of the construct as described below (other combinations may be termed invalid).

| **Tag Value Combinations** | **The Implied Semantics** |
|---|---|
| isHot = *true* and guard = *false* | termination of the service execution that raised the specified *exception* |
| isHot = *false* and guard = *false* | no effect on the corresponding service execution. The action is only skipped |
| isHot = *true* and guard = *true* | Desired effect of the action on the run |
| isDelay = *true* | wait semantics until *guard* is true |
| isAtomic = *true* | Atomicity of guard evaluation and action execution, with no interleaving step in between, in the corresponding execution of the service |

Table 3.4. Summary of GuardedAction Semantics

Further the exceptions are handled by corresponding *state* component or thrown into higher level containing *system* components (comparable to the semantics of Java's *try-catch* block [Gosling 1996]). Thus *GuardedAction* provides much needed specification construct for synchronization, exception handling behavior of sequential executions in concurrent environment [Lohr 1992]. *Further these 'visible' effects of 'guards' may correspond to the semantics of 'conditions' specified on LSCs of corresponding interface specification*. Thus cmUML relates constructs and semantics regarding synchronization, exception handling in both interface and internal specifications.

The semantics of main abstractions of cmUML are described below:

- **System**: the main abstraction which contains other components compositionally and associated with their initialization behavior of its *Port* and *State* (if exist). Further it defines a 'scope' for containing components defining 'runs' of the executions as a whole which can be verified against the specified LSCs as part of its interface specification i.e., *port*. If not decomposed further, a *system's* internal behavior can be completely specified in terms of *state* and *service* components based on required implementation semantics [Girault 1999]. Thus, this internal specification *corresponds to an abstract implementation and a higher design specification of a component* towards easier synthesis. The *port* and *state* are asynchronously executing behaviors where as *service* instances are dynamic components, initialized in response to external requests. A *system* component may also contain *resource* type components to specify simple, protected, shared resources with no reactive behaviors but may have internal concurrency. The static composition aspect of system components can be specified using structure diagrams like UML component diagrams (i.e., no separate concrete syntax).

- **Resource**: represents a simple protected shared resource with methods *acquire(), release(), read(),* and *write()*. A resource instance may be explicitly *acquired* and *released* (atomically). Instances of these components do not possess reactive behaviors but can have internal concurrency which is specified as part of its *ServiceTypes* (using tags serviceKind and parService ) of corresponding *port* specification.

- **Service**: the behavior corresponding to an interface of a component i.e. a collection of *ServiceTypes* is specified with data and control *flow* semantics (using UML activity diagram with UML2.0 semantics).The concurrent nature of a *ServiceType* with itself and other compatible *ServiceTypes* is specified by associated tags 'parService', 'serviceKind'. The tag parService represents the collection of other ServiceTypes whose instances can execute concurrently with this service instance. serviceKind tag specifies whether instances of the same *serviceType* can execute concurrently (*read* indicates multiple instances and *write* indicate single instance only) [Lohr 1992]. Further, events *start* and *end* are generated corresponding to a service execution (event *end* not generated if the service is terminated due to a raised exception). These events are broadcasted to all state components with in the scope of the containing *system* component.

- **Port**: represents the interface specification of component *behavior* as observable *externally*. As recommended in Lamport's transition axiom method approach, *the interface need to be specified with precise operational semantics, if required using the necessary state variables corresponding to execution environment* [Lamport 1983, 2000]. This principle is adopted in the specification of interface including concurrency control using *AccessOrder*, a protocol statemachine (for temporal ordering dependencies among the invoked services). *AccessOrder* may represent an abstract version of the more detailed internal *State* component. *ServiceHandlers* and associated execution information can be used for the detailed specification (fine-grained) of the concurrency control at the interface. Further, a *port* specification *exports* a collection of *ServiceTypes* with concurrency annotations through associated tag values for specifying concurrent semantics of their invocations (while this may specify the same information as that of *AccessOrder* specification, the latter may additionally specify fine-grained concurrency control). The *AccessOrder* is an important abstraction addressing many issues of concurrent systems [Jagadish 2007(a)]. For a concurrent component, this also aids in identifying concurrent sub components [Jagadish 2007(b)], as described in next chapter.

- **State**: specifies the reactive, coordination, exception handling aspects of internal behaviors of *system* components. The associated *Reactive* behavior (specified using behavior statemachines) executes asynchronously with respect to instances of *services* of a component (this behavior can be restricted for an implementation environment). To understand the concurrency semantics intuitively, a *system* component associated with a *state* behavior is analogous to an operating system **monitor** with concurrent threads of control which synchronize on need (classical monitors cause unnecessary mutual exclusion [Jagadish 2007a]). Thus UML approaches that use low level constructs like semaphore, monitor to specify concurrency behavior suffer from *implementation bias* and prohibit legal implementations that 'could' satisfy the specification. Though a *state* behavior corresponds to corresponding *AccessOrder* behavior of the interface specification the corresponding internal specification may contain additional states, transitions, and activities (comparable to Lamport's notion of *stuttered* transitions [Lamport 1983, Lamport 2000]). The meta-operations *wait()* and *notify()*, associated with a *state* behavior facilitate 'explicit' form of service synchronization (comparable to classical monitors). Further a *state* component receives events *start*, *end* indicating the execution status of instances of 'services' of the corresponding *system* component.

- **ScenarioContext**: corresponding to each use case, *ScenarioContexts* represent interactions of internal behaviors, as observed externally, with precise notions of compulsory, optional, and wait semantics. These contexts are specified using live sequence charts (LSC) [Damm 1999]. cmUML extends the LSCs to specify 'wait' semantics of *guarded* actions, and other conditions. In concurrent environment, certain executions may need to 'wait' for certain events, or conditions to happen. Though LSCs are specified for externally observable behaviors in response to use cases, certain synchronization aspects may be externally visible (even otherwise it may be necessary to make them visible for the sake of verification of certain aspects). Usually these contexts represent the principle behavior of the system without considering error scenarios. But as cmUML provides explicit exception handling mechanisms, certain important exceptions (which are visible externally) and expected system response can be specified (and verified) using the secondary scenario contexts i.e., *SecondaryContexts* triggered by corresponding *exceptions* (raised by the *PrimaryContexts*).

## 3.6 SUMMARY

This chapter provided the overview, design rationale, and definition of cmUML, the proposed specification framework and the associated specification language through lightweight extensions in UML. Based on the conceptual framework of UML/SPT profile, cmUML provides the constructs, and concurrency semantics towards precise behavioral structuring and specification of system components. Thus cmUML complements the existing UML framework with latter’s design rational in tact. cmUML framework further integrates the higher level formalisms of UML with underlying object model and UML action semantics. Further the framework provides constructs and mechanisms for explicit specification of concurrency, reactivity, exception handling, and synchronization.

# Chapter 4
# SPECIFICATION METHODOLOGY

Software development methodologies and processes play a major role in software engineering. They specify a systematic procedure of applying principles, tools, techniques, and heuristics in rapid development of software systems. Though UML has become the *de facto* industry standard language, it does not prescribe any standard development process and leaves the task to profile developers or domain experts or tool developers. Hence the thesis proposes a specification methodology for application of the proposed cmUML framework and its UML extensions. As the framework is defined using light weight extension mechanisms of UML, it can be used along with other UML based methods, and tools. The thesis proposes a specification refinement for application of the proposed cmUML framework and proposed UML extensions.

This chapter presents the second part of the thesis contribution i.e., a step-wise specification process that can be applied to develop hierarchical specification of concurrent, reactive systems (or subsystem components) using proposed cmUML framework and its profile. The process is demonstrated using the well known problem of a Vending Machine specification. Further the advantage of the cmUML approach is demonstrated by comparing and validating it with other approaches, formal as well as semi-formal. For this, classical problems of concurrency i.e., readers-writers problem and producer-consumer problem are specified and compared against their specifications in UML and formal approaches.

## 4.1 SPECIFICATION PROCESS WITH cmUML

In this section, a step-wise specification methodology for the application of the cmUML framework is proposed. The methodology assumes use case based requirement analysis. Also a higher level decomposition strategy is assumed for arriving at the initial subsystems i.e., the large-grained architectural components that may be deployed on a single physical computing node [Gomma 1993, Gomma 2000]. For the simple case study of Vending Machine specification below, there is only one subsystem that can be considered as the initial *System* component. For a complex system there may exist many subsystems for which the methodology can be applied independently. We describe the specification approach in terms of the following tasks.

**Interface Identification (Task1)**: Identify the *offered* 'services' of the *system* component through its interface. The information can be obtained from requirement artifacts like problem statement, use cases, and context diagrams for the subsystem under specification process.

**Interface Specification (Task2)**: The detailed *Port* specification of a *system* component. This includes the specification of service types, and related tags as well as protocol statemachine like behavior. Determine the concurrent execution behavior of *offered* interface services (*serviceKind* and other tags). For interface *ServiceTypes*, the temporal ordering dependencies, if any, as observable externally, can be specified as the *AccessOrder* part of the *Port* specification. *AccessOrder* is a protocol statemachine, but its transition guards may include expressions over incarnation counters of *ServiceHandlers* corresponding to each *ServiceType* defined. This detailed *AccessOrder* specification is also useful for a suitable decomposition of a *system* component i.e., identification of concurrent subcomponents as described in next task.

**System Decomposition (Task3)**: Considering the *Port* specification obtained in previous task, i.e., the externally observable *concurrent* component services, a suitable component (or subsystem) decomposition needs to be performed. This can be done by dividing the interface services into a set of concurrent 'groups' of services. This decomposition can be fine tuned by applying the general task cohesion principles from OOAD approaches (e.g. functional cohesion) [Gomma 200]. Task cohesion principles, largely design heuristics, help identify optimum number of concurrent tasks in a system. For example, sequential cohesion and functional cohesion can be applied to group two services of different concurrent components into a single component as sequential cohesion implies sequential execution among the identified tasks and functional cohesion implies the tasks are related into performing similar services. Each of these concurrent groups, identified by applying simple design heuristics to the artifact of previous task, can be specified as a *system* subcomponent. For simple components with no internal structure, this step may be skipped.

**Internal Specification (Task4)**: For each *system* sub-component, repeat the tasks 1, and 2. Once the decomposition process is over, each sub-component specification (currently consisting of interface part only) may be refined by providing the necessary internal specifications i.e., *State* and *Service*. As explained in previous chapter, this depends on the behavior semantics of the *System* (sub) component under consideration. For example, a component with state-based internal concurrency represents a reactive behavior and hence requires an internal *State* specification for necessary synchronization, exception handling aspects. For a component with state-less internal concurrency, the internal specification may not contain a *State* specification (even with this, a fine-grained concurrency, if required, may be specified using *Port* specification alone, based on the attributes of *ServiceHandlers*).

**Interaction Specification (Task5)**: Corresponding to each use case of a *System* component under consideration, specify one or more *ScenarioContexts* involving interaction between its sub-components. These interactions may also involve the corresponding *Port* and *State* components if required. *ScenarioContext* are UML sequence diagrams with explicit liveness semantics and event orderings borrowed from Live Sequence Charts [Damm 199, Damm 2003]. Further the notion of 'pre-charts' as defined in LSCs are very useful to specify a generic pre-conditions, as a sequence of messages exchanges, for triggering above *ScenarioContext*. Thus a *ScenarioContext* specifies primary activities of *System* components in response to external requests and events. These contexts can be further extended, as explained in Task6, with *exception* conditions ‘triggering’ *SecondaryContexts*, the latter thus specify much needed exception handling features (important characterisitics of concurrent and reactive systems).

**Service Specifications (Task6)**: UML2.0 has replaced the statemachine like semantics of activity diagrams with control and data flow semantics [Selic 2004]. These diagrams are very useful to specify sequential executions and local actions internal to concurrent component executions. As explained in previous chapter, cmUML extends the UML activities and actions to specify the semantics of synchronization, atomicity, and exception handling. Thus internal specifications of services (i.e., *ServiceType*) in cmUML provide unambiguous specification of sequential executions in concurrent and reactive environments.

**Synchronization, Exception Handling Features (Task7)**: Identify the different synchronization, and exception handling situations. This may be done by identifying *GuardedActions* in *Service* and *ScenarioContexts* specifications. Further refine the internal specifications i.e., *Service*, *State* (if exist) and *ScenarioContexts* specifications by identifying synchronization, exception handling aspects among the concurrently executing behaviors of the component. This leads to identification and specification of *SecondaryContexts* i.e., *responses* to the raised exceptions. The task also includes identification of appropriate invariants in the *Port* specification (e.g., fine-grained synchronization patterns using attributes of *ServiceHandlers*).

**Task8**: Repeat above Tasks1-7 for all *System* components and also *System* sub components identified in Task3.

## 4.2 CASE STUDY: VENDING MACHINE SPECIFICATION

### 4.2.1 PROBLEM STATEMENT

The specification methodology as described in the previous section is further elaborated through a case study, the well known problem of Vending Machine specification [Schinz

2004]. A vending machine (VM) accepts coins from users to dispense a drink of chosen choice. The user gives coins, one at a time, and while the sum is sufficient enough the corresponding choices of available drinks are updated. The user can select any of enabled choices. The drink and the extra coins, if any, are dispensed (for simplicity, we assume that the VM doesn't remember the coins of previous transactions). Also the user's request to cancel the transaction may be considered for e.g. before the drink or coins are dispensed.

### 4.2.2 SPECIFICATION PROCESS

The specification process assumes a suitable subsystem identification strategy as discussed in [Gomma 1993, Gomma 2000]. This allows identification of large grained architectural components in the system usually corresponding to physical architecture of the systems, which can be best specified using UML deployment diagram.

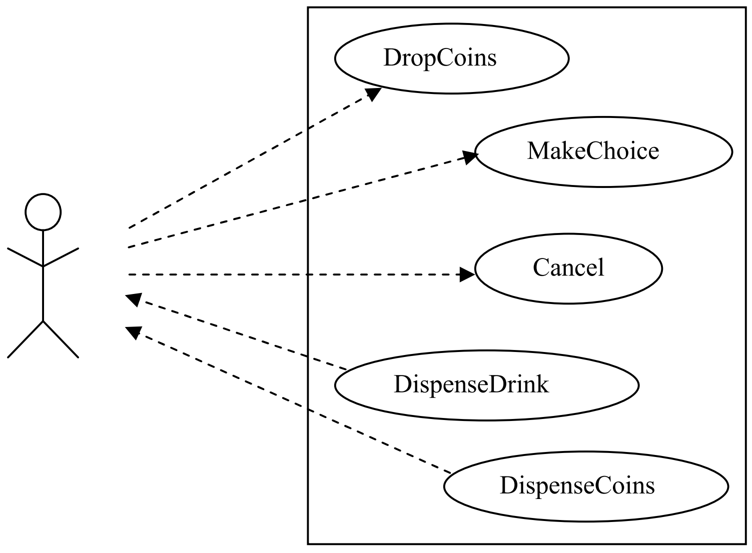


Figure 4.1. General Use Cases of a Vending Machine

cmUML specification process (Task1-8 as explained in the previous section) is based on artifacts of requirements elicitation phase. In UML, this activity is commonly performed using the informal design notation of Use Cases. A use case corresponds to a specific system functionality involving actors of the subsystem i.e., external entities which include both persons and other sub systems. In the current case study of the vending machine, there is at most one actor interacting with the system. Further as the case study is a simple, we assume there is only one subsystem comprising a vending machine system. The general use cases of a vending machine are given in figure 4.1.

**Identification of System Interface (Task1)**: Interactions of the system (vending machine) with its environment (user), i.e., externally observable behaviors can be understood from system requirements diagram (use cases in UML). In case of vending machine's external behavior, the user *inserts* sufficiently more coins and when prompted by the VM *selects* his choice of the drink. The VM, after *validating* the choice and the received coins, *dispense*s the 'drink' as well as the balance coins if any. From the initial analysis of these interactions, we can observe four principle services of the VM, involving its environment (user): *ReceiveCoins*, *ReceiveChoice*, *DispenseDrink*, and *DispenseCoins* (denoted as *R-Coins, R-Choice, D-Drink, D-Coins*). During this task, the error or exception handling situations are not considered.

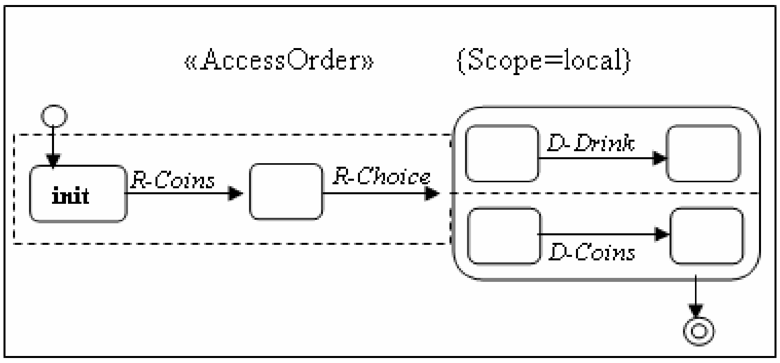


Figure 4.2. *AccessOrder* Specification of «system» VM

**Specification of System Interface (Task2)**: Now, we specify the interface specification i.e., *Port* component of the VM system. This includes both static aspects (i.e., *ServiceTypes* and attributes) and the dynamic aspects (e.g., temporal dependencies at run-time using *AccessOrder*). As in figure 4.2, *AccessOrder* specifies concurrency and temporal dependency between the interface services, as can be observed externally. The specified tag value *scope* is redundant in this example as there exist at most one user at a given time interacting with the VM (*scope* tag indicates whether the interface behavior is applicable locally per a given actor or global invocation order among all actors [Jagadish 2006a]). The complete interface specification («port») includes concurrency aspects of all *ServiceTypes* of the component (again as observed externally). For VM, all service types have same tag values {isAtomic=*false*; serviceKind=*write*; max=1} with additional information that *D-Drink, D-Coins* may execute in parallel.

**System Decomposition (Task3)**: From the *AccessOrder* specification of «system» components it is possible to identify concurrency or sequential dependency among interface services. For VM component, from figure 4.2, a concurrent region and a sequential region in dashed border indicates that the services **DispenseDrink** and **DispenseCash** may execute concurrently while **ReceiveCoins** and **ReceiveChoice** have sequential dependency. Now following task cohesion principles of OOAD approaches [Gomma 2000], we can identify two concurrent components with functionally related services; a **CashExchanger** component (CE) (that handles interface services *R-Coins, D-Coins*) and a **DrinkDispenser** component (DD) (that handles interface services *R-Choice, D-Drink*). With the identification of internal components (concurrent, or sequential), the internal structure of the VM system can be specified using UML structure diagram (figure 4.3). In cmUML a component represent a collection of behaviors at an abstract level (and independent of class or object diagram approaches of usual UML approaches).

**Internal Specifications (Task4)**: If a *System* component is decomposed its interface may be completely delegated to its sub components (as in the current case of VM system). Even so, the *System* component may contain the *State* specification for necessary coordination, synchronization or exception handling aspects. For example, for VM system component, the temporal dependencies specified as part of interface specification must be preserved in the internal specification. The internal *state* specification VM system (figure 4.3) specifies externally observable states (it may also contain other states) as well as the coordinating or synchronization of internal behaviors. The transitions are augmented with new kind of events e.g. *start*, *end*, and *exception* towards specifying the necessary coordination and exception handling mechanisms. There are no *Service* specifications at the top-level of the VM system. The **Cancel** use-case i.e., the possible cancellation of the transaction is specified as an *Exception* as it requires coordination of the sub-components rather than a sequential *Service* execution.

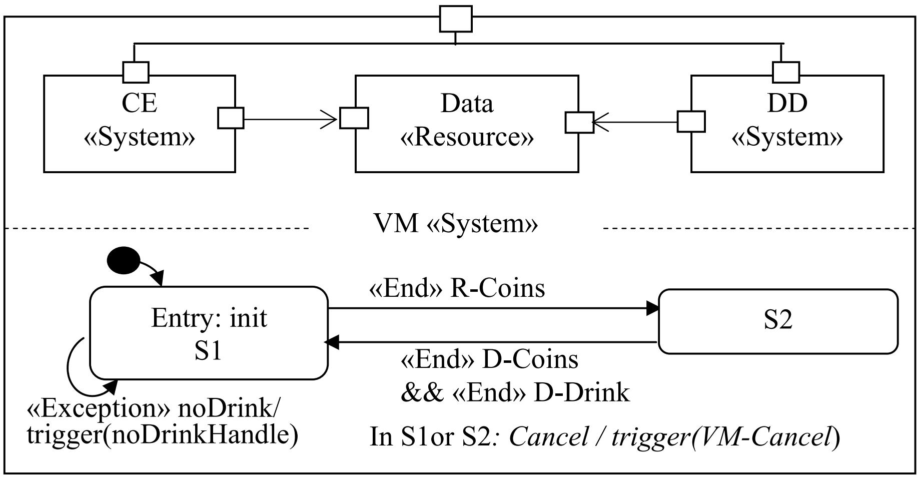


Figure 4.3. Specification of internal structure and *State* of «system» VM

**System Interactions (Task5)**: During this process of specification, primary interaction scenarios i.e., *ScenarioContexts* are specified using sequence diagrams. The primary *ScenarioContexts*, also *PrimaryContexts*, correspond to the main functionality or use cases of the *System* component. As UML sequence diagrams are not associated with precise semantics, cmUML adopts the LSC formalism towards specifying **optional** and **mandatory** behaviors through interaction with external environment [Damm 1999]. LSCs extend sequence diagrams by associating all the elements of sequence diagrams e.g. lifelines, messages, conditions etc with two kinds of annotations i.e., *hot* and *cold*. The *hot* annotations (graphically solid lines) indicate mandatory or compulsory behavior indicating liveness where as cold annotations (graphically dotted elements) indicate optional semantics. Further the cmUML profile extends the LSC framework to indicate wait semantics for certain conditions specified on LSCs (this feature relates the interface and internal specifications for consistency). This may be alternatively specified using explicit synchronization messages with *State* component.

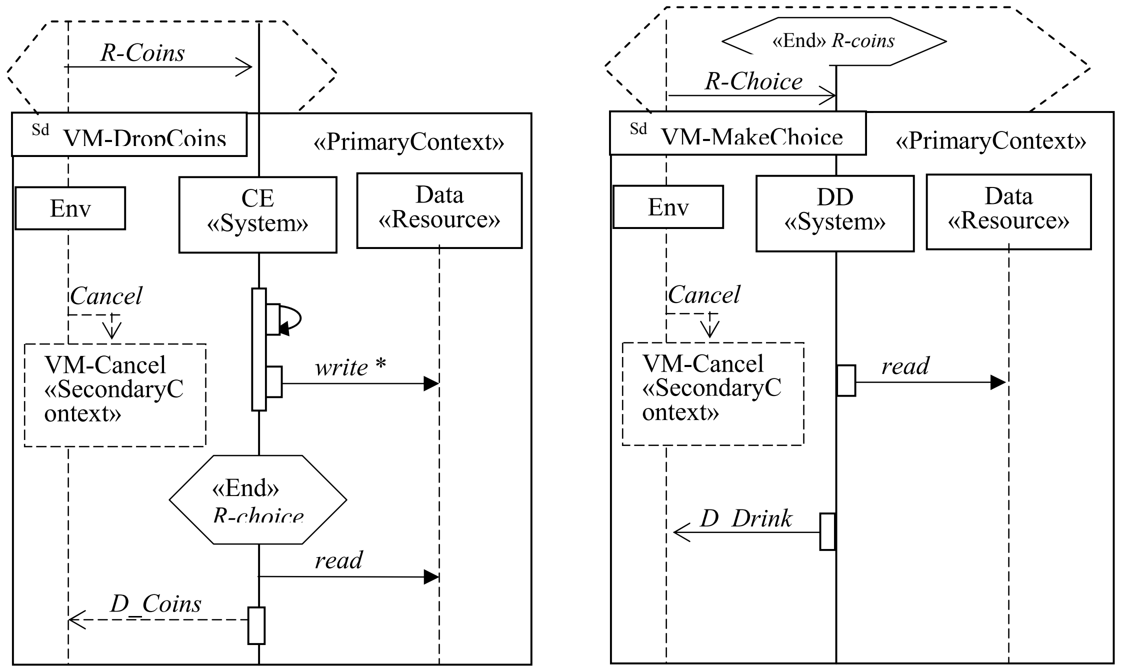


Figure 4.4. «ScenarioContext» Specifications of Vending Machine Use Cases

*ScenarioContexts*, as LSCs, can be associated with *pre-charts* indicating optional precondition behavior expected of environment towards execution of a use case and are comparable to specification of pre-conditions of procedures in sequential execution environment. The environment including actors, and other hardware interfaces e.g. **choicePanel**, display interfaces etc is denoted by **Env** in scenario context specifications. Further these contexts are useful towards simulation or formal verification of correctness, liveness, and property verification etc as described in later chapter. Only specified events under given liveness constraints are of interest to the context/ use-case under consideration with respect to overall system behaviors (which may include other 'unspecified' internal events, actions etc comparable to the Lamport's 'stuttered' transitions [Lamport 2000]).

Figure 4.4 specifies the *ScenarioContext* corresponding to **DropCoins** use case. Pre-chart, an optional behavior, acts a 'triggering' mechanism for execution of corresponding *ScenarioContext* (pre-charts with single message can also be specified within the scenario context using the 'trigger' stereotype. For example, an important *SecondaryContext* VM-Cancel is invoked if the user presses 'Cancel' at any point of time.

**Service Specifications (Task6)**: The computational aspects of *ServiceTypes* of a component are specified using activity diagram (with data and control flow semantics). The component VM System has no service specifications as its *ServiceTypes* are all delegated to its sub-components. For description purpose we chose a service of its subcomponent. Figure 4.5 specifies the sequential execution behavior of the *service* **R-Coins**. The specification contains *GuardedActions*. The associated tag values specify that the service execution does not wait for guard value to become true and terminate by raising an exception. *GuardedActions* are useful to specify atomic update of shared data values (isAtomic=*true*) or synchronization semantics regarding guard evaluation. In this context atomicity indicates that the guard value can not change during execution of the action(s) (for example, in fig.5 the outer guard corresponding to drinks availability cannot change during execution of R-Coins behavior). Also a guard expression may declaratively specify a condition referring to old and new values w.r.t action under consideration, using notation e.g. x@preAU and x@postAU enhancing expressiveness of specifications [Lam00].

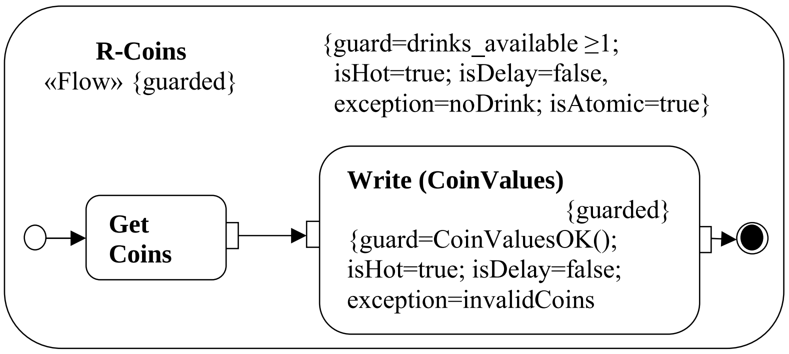

R-Coins
«Flow» {guarded}
{guard=drinks_available ≥1;
isHot=true; isDelay=false,
exception=noDrink; isAtomic=true}
Get Coins
Write (CoinValues)
{guarded}
{guard=CoinValuesOK();
isHot=true; isDelay=false;
exception=invalidCoins

Figure 4.5. Activity Specification of a Service with *guarded* Semantics

**Synchronization and Exception handling (Task7)**: Both interface and internal specifications can be examined for further synchronization and exception handling involved. For example the interface specification can be augmented by global invariants over incarnation counters *in, out* of *ServiceHandlers* of corresonding *ServiceTypes* specifying global synchronizations if any (e.g. fine-grained concurrency control). A *State* specification can be easily extended to specify new exceptions and corresponding exception handling contexts or services (which executes synchronously with the *State* behavior). New events, both external and internal, may be identified. In figure 4.3, the *State* specification is extended with new exception **noDrink,** handled synchronously.

```
context Semaphore inv:

context Semaphore
 def:       waiting       :   Sequence{ }

context Semaphore::P(ID:T) : void
 body :
           if  self.s@pre > 0 then
                     self.s = self.s@pre - 1
           else
           self.synchronizedProcess->select ( p: Process |
                      p.ID =ID).semSynch.isActive = false
           self.synchronizedProcess -> select (p:Process |
                     p.ID=ID). semSynch.isBlocked = true
           self.waiting = self.waiting@pre -> append (ID)
           endif


context Semaphore:: V() : void
 body:     if self.s @pre = 0 then
           if self.waiting @pre -> isEmpty() = false then
           self.synchronizedProcess -> select (
           p: Process | p.ID = self.waiting @pre ->
                     first()).semSynch.isBlocked = false
           self.synchonizedProcess -> select(
           p : Process | p.ID = self.waiting @pre ->
                     first()).semSynch.isActive = true
           Self.waiting = self.waiting@pre ->
           Subsequence (2, self.waiting@pre -> size())
           endif
           endif
           else
                     self.s = self.s @pre +1
```

Table 4.1. An OCL based Specification of Semaphore Semantics

In figure 4.4, the *ScenarioContexts* are extended with possible external event **Cancel** corresponding to user cancellation of the transaction. This is handled as an exception even though specified as a use-case of the system. The corresponding exemption handling is specified as a *SecondaryContext* i.e., VM-Cancel as it requires specific interactions among the sub-components.

**Sub-Component Specifications (Task8)**: We can complete the specification of VM by specifying each «system» sub component (CE, DD) following Tasks 1-7. We skip the description of these steps here. The complete specification of VM system is presented in the appendix.

## 4.3 COMPARISON AND VALIDATION

In this section we validate cmUML framework through specification of well known classical concurrency problems i.e., readers-writers, and producer and consumer problems. We compare and validate the specifications against current UML approaches for the same, e.g. as given in [Goni 2004]. Generally these approaches use low-level primitives like locks, semaphore, and monitors etc to describe concurrent behavior where the semantics of these constructs are either not specified or specified in complex program-like OCL statements.

A. Goni and Y. Eterovic, in [Goni 2004], provided precise OCL semantics to low level concurrency mechanisms like semaphores, and monitors. Using these mechanisms they have shown that certain concurrency problems e.g. Sleeping Barber problem can be precisely specified. Their OCL based specifications for 'Semaphore' is reproduced in table 4.1. A similar OCL specification is defined for 'monitor' construct. The specifications are restricted by the semantics of these low-level primitives [Jagadish 2007]. In these approaches, though higher level diagrams are used, no precise semantics can be inferred about behavior of the specification. In contrast, *cmUML* specifications convey precise semantics without using low level primitives or complex OCL statements.

### 4.3.1 CLASSICAL PROBLEMS OF CONCURRENCY

Current UML approaches are not precise regarding semantics of concurrency in specifications. Most approaches are based on low-level implementation constructs like semaphore, and monitor to convey the semantics of concurrent behavior thus suffering from implementation bias. Further there are certain inherent problems associated with these low-level mechanisms for e.g., sometimes the unnecessary mutual exclusion imposed by a 'monitor' is not suitable to specify the required fine-grained concurrency semantics [Jagadish 2006(a)]. Below we specified two well known problems for concurrency; readers-writers problem and producer–consumer problem. These problems are first specified using existing UML approaches and then in the proposed cmUML specification framework. It should be easy to see that the higher level semantics precisely specified by the cmUML approach is superior to OCL and low-level primitive based approaches. To compare the formality and specification power of cmUML, the specifications of these problems in a formal language are presented and compared.

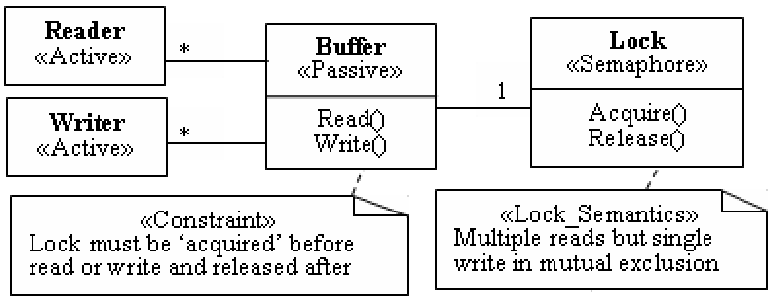


Figure 4.6. Specification of Readers-Writers Problem in UML

### 4.3.2 READERS-WRITERS PROBLEM

Readers-writers problem is a well known concurrency pattern and often used to demonstrate the concurrency aspects of a language. It states situation where there is a collection of 'reader' entities (processes, or actors), and a collection of 'writer' entities which access a shared resource. Readers can access the resource in parallel while writers access in mutual exclusion with other writers and readers. This problem is a very simple pattern and many solutions exist based on suitable read-write lock like 'semaphore'. But the specifications in UML also use these mechanisms to convey the required semantics of the pattern i.e., readers could be accessing simultaneously and writers in mutual exclusion with all other entities.

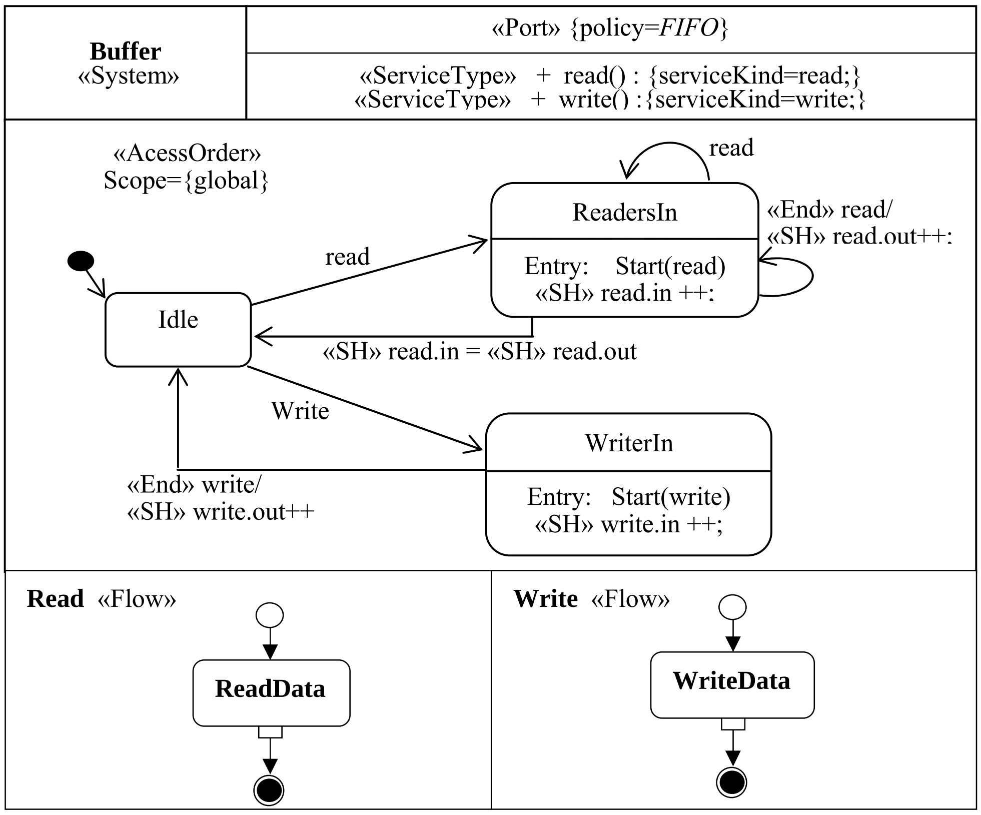


Figure 4.7. Specification of Readers-Writers Problem in cmUML

Figure 4.6 shows the specification of the problem using one of current UML practices (semantics of 'semaphore' lock, as OCL statements, is presented in table 4.1) and in figure 4.7 using the proposed cmUML approach. The proposed approach retains the abstractness of specifications yet provides the precise concurrency semantics. *AccessOrder* i.e., a UML statemachine specifies the interface concurrency. The transitions are labeled with *End* events of service executions as well as the 'in' and 'out' i.e., the incarnation counters of service handlers corresponding to a given service type.

```
      call resource.startread    call resource.startwrite
      read                                              write
      call resource.endread                 call
resource.endwrite


      // A variable 's' defines the current resource state as

      s=0 : 1 writer uses the resource
      s=1 : 0 processes use the resource
      s=2 : 1 reader uses the resource
      s=3 : 2 readers uses the resource
      ...  ...

      Process resource
      s : int

      Proc startread  when s≥1 : s:=s+1 end
      Proc endread    if   s>1 : s:=s-1 end
      Proc startwrite when s=1 : s:=0 end
      Proc endwrite   if   s=0 : s:=1 end
           s:=1
```

Table 4.2. Specification of Reader-Writer problem in *Distributed Processes* framework

The complete formal specification of the problem is given in table 4.2. It is based on **Distributed Processes** framework due to Brinch Hansen [Hansen 1978]. The constructs *when* and *if* represent a *guarded region* and a *guarded statement* respectively with associated formal semantics w.r.t delay and exceptions. It can be seen that the specification power of cmUML matches with that of formal approaches yet provides better intuitivity and abstractness.

The cmUML specification of readers-writers problem does not use many features of cmUML profile as it represents a simple *system* with no internal structure and has simple behaviors i.e., the sequential executions of the *ServiceTypes* i.e., Read(), and Write(). For example, the specification of the internal *State* component is not required as there is no synchronization, exception handling aspects among internal executions. It may be noted, that the 'policy' tag value '*FIFO*' with «Port», guarantees that there is no starvation of 'writers' as request events are only handled in FIFO order and events stay until completely processed (default behavior as per formal semantics of a 'Port' component).

### 4.3.3 PRODUCER-CONSUMER PROBLEM

The problem involves specification of a protected passive resource i.e., a bounded 'Buffer' accessed by operations put(), get(), in state-dependent synchronization fashion, and possibly concurrently. For simplicity, we assume only single instance of these operations execute at a moment. When the Buffer is 'full' an execution instance of 'put' can not proceed and similarly when the Buffer is 'empty' an execution instance of 'get' waits until the state of the buffer changes. Thus the concurrency and synchronization behavior of the problem depends on the state of the Buffer.

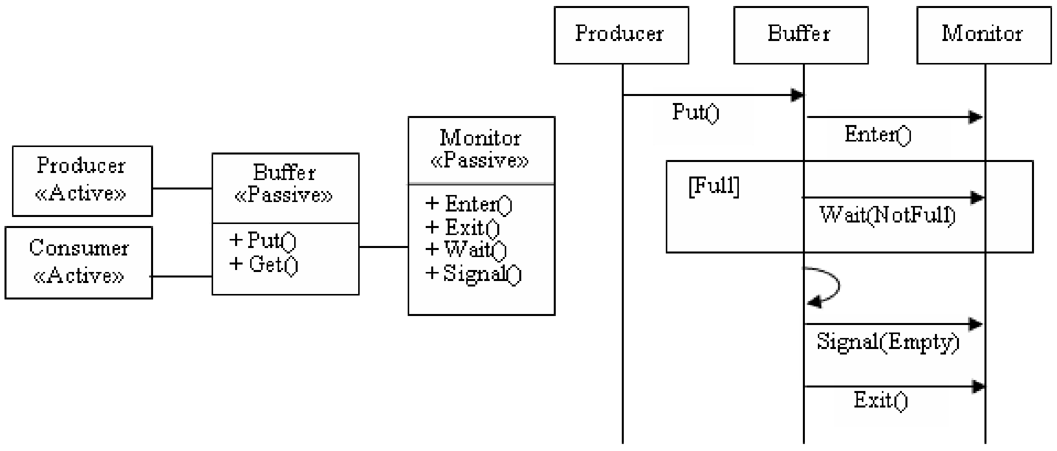


Figure 4.8. Specification of Producer-Consumer Problem in UML approaches

Figure 4.8 specifies the problem (invocation behavior of get() skipped) in current UML approaches (as in [Goni 2004]). This specification assumes monitor behavior as specified in [Goni 2004] (similar to that of semaphore using OCL statements as in table 4.1). The figure 4.9 presents the cmUML specification of the problem. The UML approach uses a sequence diagram to model the necessary state condition and related wait semantics. But the specified behavior is representative in nature and does not specify precise semantics.

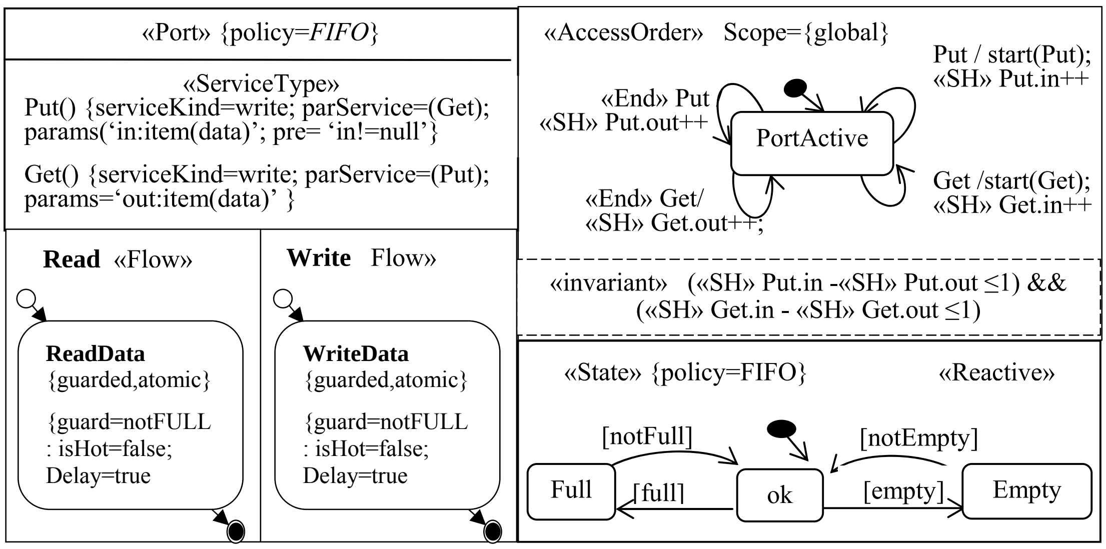

«Port» {policy=FIFO}
«ServiceType»
Put() {serviceKind=write; parService=(Get); params(‘in:item(data)’; pre= ‘in!=null’}
Get() {serviceKind=write; parService=(Put); params=‘out:item(data)’ }
Read «Flow»
ReadData {guarded,atomic}
{guard=notFULL : isHot=false; Delay=true
Write Flow»
WriteData {guarded,atomic}
{guard=notFULL : isHot=false; Delay=true
«AccessOrder» Scope={global}
«End» Put «SH» Put.out++
PortActive
Put / start(Put); «SH» Put.in++
«End» Get/ «SH» Get.out++;
Get /start(Get); «SH» Get.in++
«invariant» («SH» Put.in -«SH» Put.out ≤1) && («SH» Get.in - «SH» Get.out ≤1)
«State» {policy=FIFO}
«Reactive»
[notFull]
[notEmpty]
Full
[full]
ok
[empty]
Empty

Figure 4.9. cmUML Specification of Producer-Consumer Problem

The cmUML specification of the problem uses many abstractions of the proposed framework (but not *ScenarioContexts* due to simplicity of the *system*). In figure 4.9, «Port» specifies interface services *ServiceTypes* and their concurrency aspects through *AccessOrder* (the associated *invariant* specifies the single instance of each *ServiceType* to be executing at a given instant). Further the state-based synchronization behavior of interface operations is specified using internal *State* specification (the reactive semantics). Also the sequential behavior of interface *ServiceTypes* is specified using UML activity diagrams. These sequential behaviors contain *guarded actions* i.e., ReadData and WriteData which imply wait semantics until the corresponding guards are true. Further the spcified *atomicity* indicates that the guard value can not change during execution of the corresponding action.

Also, the complete formal specification of the problem is given in table 4.3. It is based on ‘Distributed Processes’ framework due to Brinch Hansen [Hansen 1978]. The constructs *when* and *if* represent a *guarded region* and a *guarded statement* respectively with associated formal semantics w.r.t delay and exceptions. It again validates that the specification power of cmUML matches with that of formal approaches.

```
call resource.put(x)                    call resource.get(x)

process resource
buff:[0..SIZE] of int;
head, tail: integer;

proc put(x :int)
     when tail ≤ head+SIZE : tail :=tail+1
     insert(buff,x)
     end
proc get (#x :int)
     when head < tail : head := head-1
     remove(buff,x)
     end

     begin head:=0 ; tail :=0 end
```

Table 4.3. *Distributed Processes* Specification of Producer-Consumer problem

## 4.4 SPECIFICATION OF LEADER-FOLLOWER CONCURRENCY PATTERN

There exist several design patterns for implementation of efficient concurrency models e.g. Active object, HalfSynch-HalfAsynch, HalfSynch-HalfReactive etc [Schimidt 1997]. The Leader/Followers design pattern provides a concurrency model where multiple threads can efficiently demultiplex events and dispatch event handlers that process I/O handles shared by the threads. The pattern is specified below using cmUML framework. Under the pattern, multiple former *leader* threads can process events concurrently while the current leader thread waits on the handle set. After its event processing completes, an idle follower thread waits its turn to become the leader. If requests arrive faster than the available threads can service them, it is assumed that the underlying I/O system can queue events internally until a leader thread becomes available.

Figure 4.10 presents the cmUML Specification of Leader-Follower pattern. The interface specification describes the concurrency aspect of lone *ServiceType* i.e., ‘HandleEvent'. This *ServiceType* executes concurrently but limited by the maximum number of instances that is specified (which specifies the number of threads that would be available in the equivalent implementation). The internal specification ‘State’ executes internal operations

‘PromoteLeader()’ and ‘JoinQ()’ (invoked by concurrently executing instances of the *ServiceType* HandleEvent). Also the sequential execution of the HandleEvent *ServiceType* is specified using UML activity diagram. It can be noted that this example does not require other features of the cmUML framework i.e., *ScenarioContexts* as the behavior is simple and the specified *system* does not contain any internal components (at specification level).

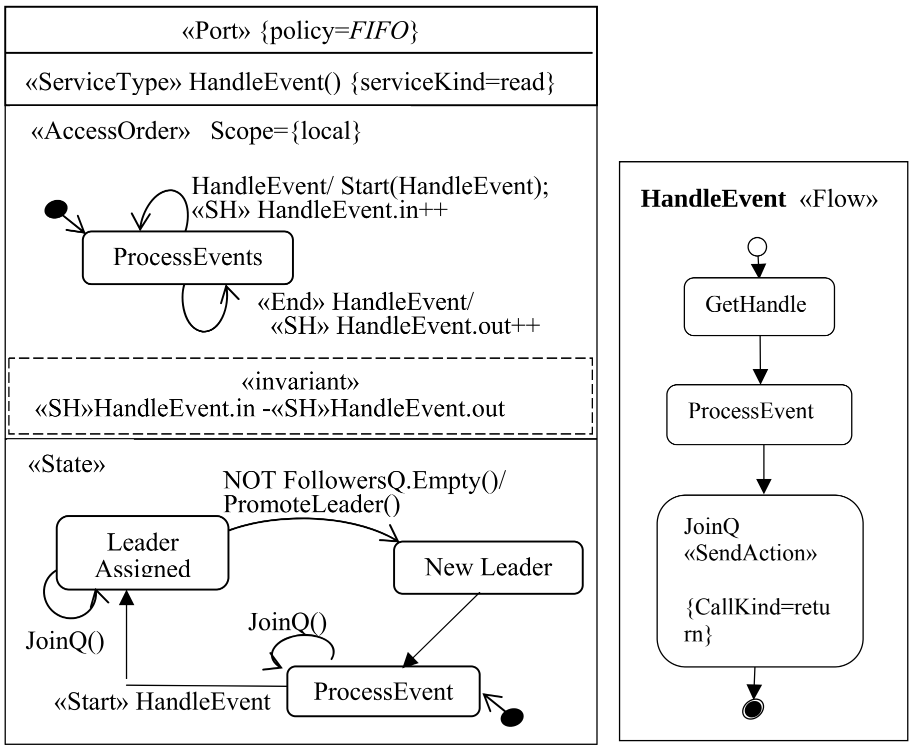

«Port» {policy=FIFO}
«ServiceType» HandleEvent() {serviceKind=read}
«AccessOrder» Scope={local}
HandleEvent/ Start(HandleEvent); «SH» HandleEvent.in++
ProcessEvents
«End» HandleEvent/ «SH» HandleEvent.out++
«invariant»
«SH»HandleEvent.in -«SH»HandleEvent.out
«State»
NOT FollowersQ.Empty()/ PromoteLeader()
Leader Assigned
New Leader
JoinQ()
JoinQ()
«Start» HandleEvent
ProcessEvent
HandleEvent «Flow»
GetHandle
ProcessEvent
JoinQ «SendAction» {CallKind=return}

Figure 4.10. cmUML Specification of Leader-Follower Concurrency Pattern

## 4.5 SUMMARY

This chapter presented the second part of the contribution of the thesis i.e., the specification methodology along with a case study of ‘Vending Machine’ specification. The chapter further discussed the comparison and validation of cmUML framework approach with existing UML based approaches using two well known problems of concurrency i.e., readers-writers problem and producer-consumer problem. Our demonstration shows that the cmUML approach provides an abstract yet precise specifications. Further the approach is very intuitive hiding the intricacies of formal semantics behind simple specification constructs and specification methodology.

# CHAPTER 5
# FORMAL SEMANTICS

Though UML has become the *de-facto* industry standard language for specification of software systems, the UML models are not suitable for formal analysis. This is mainly due to lack of formal semantics in UML. Several approaches were taken in providing formal semantics to UML, or suitable subsets of it, as required for the domain under consideration. This thesis has proposed a specification framework, namely cmUML, for precise and explicit specification of concurrency, reactivity, exception handling, and synchronization for system components. Further a specification language, namely cmUML profile based on UML extension mechanisms, has been defined. Towards rigorous and unambiguous specification framework, this thesis work further proposes an appropriate semantic foundation as described in this chapter.

The formal semantics of cmUML framework is largely based on so called informal semantics descriptions of UML and UML/SPT Profile. This does not cover all elements of UML and SPT metamodels but only those providing foundations for cmUML constructs. The formal semantics is described along two dimensions i.e., the interface, and internal specifications of system. For this existing semantic descriptions and formalisms are adopted with suitable extensions wherever necessary. The chapter also provides an overview of existing approaches of semantics descriptions in UML.

## 5.1 INTRODUCTION

The cmUML specification framework defines a two level specification process (figure 5.1) in terms of interface specification (based on LSCs and protocol state machines), and internal specification (based on activities and statemachine i.e., data/control flow semantics, reactive semantics respectively). Hence the formal semantics is described in two separate but related dimensions. For interface specifications, the semantic framework of LSCs [Damm 1999] is extended as required. The LSC semantics provide the liveness semantics of executions of the system as observable externally. The semantics of the internal specifications is described using ‘Symbolic Transition Systems (STS)’. For this purpose, the foundation part of krtUML approach has been adopted [Damm 2002]. STS based formal semantics of cmUML internal specifications is general enough to provide a flexible and heterogeneous implementations. This is particularly useful for internal specifications of basic i.e., non-composite *System* components in cmUML, as these specifications are close to implementations. For example, the internal concurrency and non-determinism as provided by the semantics may be compiled away by specific implementation semantics (e.g. sequential execution environment). Further formal semantics preserves the semantics of multiple formalisms associated with cmUML i.e., sequence charts, activity diagrams and statemachines, providing much needed consistent and unifying semantic foundation.

The separation of concern approach in formal semantics definition of cmUML framework facilitates the integration of various formal verification approaches. For example, the LSC based interface specifications can provide simulation or formal verifications of interface correctness, deadlock etc. Formal verification of internal specifications is achieved by translating the specifications to chosen semantic domain in consistent with adopted semantics and execution models of implementations. For example, in next chapter, we describe a CSP translation mechanism for formal verification of internal specifications of basic non-composite *system* components.

## 5.2 SEMANTICS DESCRIPTION APPROACHES IN UML

There exist several approaches in defining formal semantics for UML models [described in detail in section 2.2.3]. While a few of the approaches focused on some chosen subsets of UML and only static aspects, other approaches provided formal semantics of behavioral specification of UML with concurrency, and reactivity. Reggio et.al considered the problem of defining active classes with associated statemachines [Reggio 2000]. They gave a very fine interleaving semantics for state-machines in terms of labeled transition systems (used to model concurrent languages such as Ada, part of Java). The labeled transition system associated with an active class is presented using the algebraic specification language CASL [CoFI 1997]. This formalization of active classes and state machines has led to perform a thorough analysis uncovering many problematic points in the official informal semantics of UML. But, this work does not give precise semantics for statemachines, event queue handling, and action semantics.

Hussmann introduced the so called ‘loose semantics’ approach where the aspects of concurrency and object communication are not fixed to some design decision, but cover different implementations. Such loose semantics is not suitable for formal verification [Hussmann 2002]. The semantics of a UML class diagram is constituted by all object algebras that are type conformant to the class diagram. The approach combines the existing approaches of formal semantics e.g., set-theoretic, translation based, and meta-modeling, to give more abstract semantics. Hussmann’s approach provides semantics in two versions: first a direct semantics based on plain mathematical set theory, then a sketch of a meta-modeling approach to the same concept.

krtUML provides the semantical foundations for formal verification of real-time UML models [Damm 2003]. krtUML approach fixes one detailed formal semantics to support verification purposes. The subset krtUML of UML is rich enough to express all behavioral modeling entities of UML. A formal interleaving semantics of for this kernel language is defined by associating with each model M in krtUML a symbolic transition systems STS(M).

Thus many approaches try to formalize UML or a subset of it by using a particular semantic language, or a collection of them, or by fixing a detailed semantics. Each of these approaches has many limitations and represent heavy-weight extensions of UML not supported by UML and related tools. But the formal semantics of cmUML is close to UML core and so-called informal semantics.

## 5.3 SEMANTIC FOUNDATIONS OF cmUML FRAMEWORK

A layered approach is defined for semantic descriptions of the cmUML specifications (figure 5.1). The semantic approach separates externally visible interface behaviors from internal implementation behaviors facilitating different verification approaches. The formal semantics assume the consistency between the interface and internal specifications. As shown in figure 5.1, the semantic foundation of cmUML specifications is based on the precise action semantics of UML and related dynamic aspects e.g., events, causality, concurrency, and abstract resources as defined in UML/SPT metamodel. Further the classical semantics of the higher level formalisms e.g., Statecharts, Activities, and Sequence diagrams are retained as defined in original UML framework.

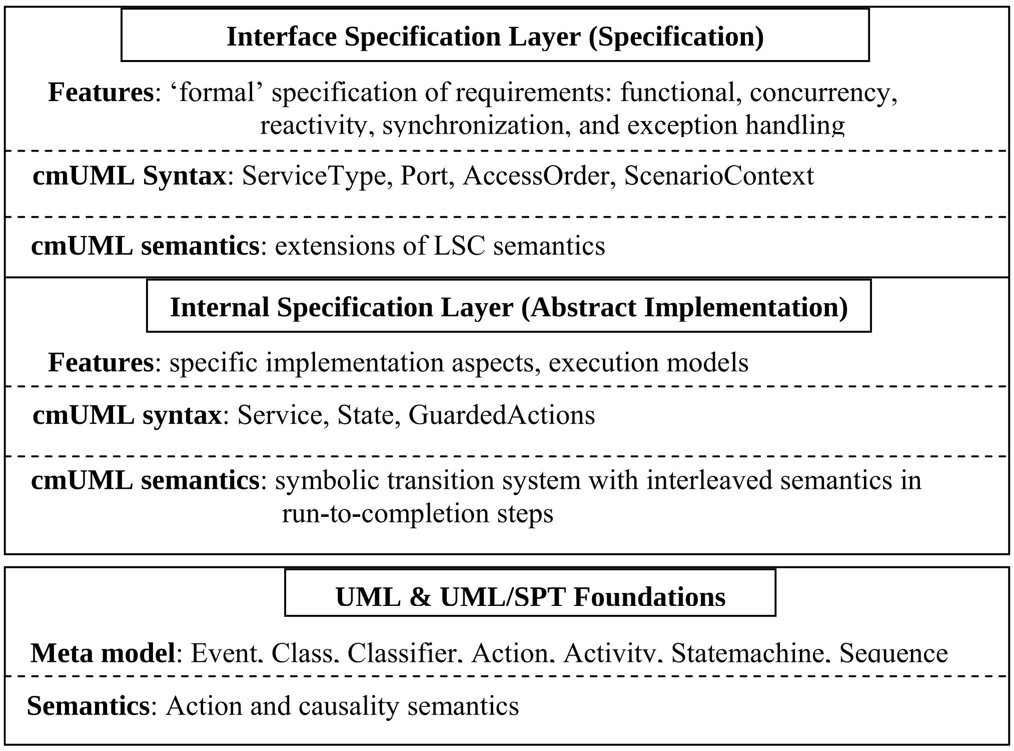

Interface Specification Layer (Specification)
Features: 'formal' specification of requirements: functional, concurrency, reactivity, synchronization, and exception handling
cmUML Syntax: ServiceType, Port, AccessOrder, ScenarioContext
cmUML semantics: extensions of LSC semantics
Internal Specification Layer (Abstract Implementation)
Features: specific implementation aspects, execution models
cmUML syntax: Service, State, GuardedActions
cmUML semantics: symbolic transition system with interleaved semantics in run-to-completion steps
UML & UML/SPT Foundations
Meta model: Event, Class, Classifier, Action, Activity, Statemachine, Sequence
Semantics: Action and causality semantics

Figure 5.1. The Layered Approach of cmUML Semantic Framework

### 5.3.1 INTERFACE SPECIFICATIONS

The top layer of the semantic framework (figure 5.1) represents the interface specifications in cmUML. This layer is close to the requirements phase of software development processes and hence represents the required interface behaviors associated with liveness constraints as well as synchronization and exception handling as observable externally of a *System* component. As the dynamic aspects of interface specifications are represented by live sequence charts (LSCs) and protocol statemachines, the related formal semantics is based on the semantic foundations of LSCs and its suitable extensions as required.

### 5.3.2 INTERNAL SPECIFICATIONS

The middle layer of the semantic framework (figure 5.1) represents the internal specifications of cmUML components in general or basic, non-composite, 'system' components in specific. This layer is close to the design or (abstract) implementation phase of software development processes and hence represents the implementation semantics, execution models *System* components. As the internal specifications consist of UML statemchines and Activities, the corresponding formal semantics is described as a Symbolic Transition System (STS). A simple C-like programs using with non-deterministic choice of actions, is defined for the formal semantics of main abstractions of the cmUML framework.

## 5.4 SEMANTICS OF INTERFACE SPECIFICATIONS

An interface specification of a *System* component in cmUML mainly consists of a collection of *ServiceTypes* (representing static aspects) and *AccessOrder* and *ScenarioContexts* (representing dynamic aspects of execution instances of *ServiceTypes*). The *ScenarioContexts* are further divided into *PrimaryContexts* and *SecondaryContexts* which represent the system functionality and exception handling aspects as observable exernally. These contexts are specified using an extension of LSC formalism as described in previous chapters (chapters 3 and 4). The live sequence charts represent a collection of represent a collection of partially ordered sets of events with associated liveness constraints. cmUML framework adopts the semantics foundations of LSCs as described in [Damm 1999], Also described in Appendix B.

### 5.4.1 LIVE SEQUENCE CHARTS

While message sequence charts (MSCs), sequence diagrams in UML, are widely used in industry to document the interworking of processes or objects, they are expressively weak, being based on the modest semantic notion of a partial ordering of events as defined, e.g., in the ITU standard [ITU 1994]. A highly expressive and rigorously defined MSC language is a must for serious, semantically meaningful tool support for use-cases and scenarios (as documented during requirements phase).

LSCs (live sequence charts) are extensions of MSCs towards precise behavioral specification of scenarios of the system. In fact, LSCs allow the distinction between possible and necessary behavior both globally, on the level of an entire chart and locally, when specifying events, conditions and progress over time. Thus, elements of MSCs e.g. lifelines, messages, locations etc are annotated with live ness constraints '*hot*' or '*cold'* (i.e., the mandatory or optional behavior respectively). The semantical basis of LSCs facilitates rigorous and complete consistency checks between the descriptive view of the system and the constructive or implementation view (e.g. statemachines). Thus LSCs allow integration of implementation models with the descriptive or requirements part of the system specifications.

The formal semantics of an LSC is described by a symbolic transition system (or a skeleton automaton as described in Appendix B). The transition system is described by its abstract states (active, terminated, or aborted) and associated atomic transitions (Figure 5.2, Figure 5.3). The detailed description of the formal foundation is given in ([Damm 1999]), also presented in Appendix B.

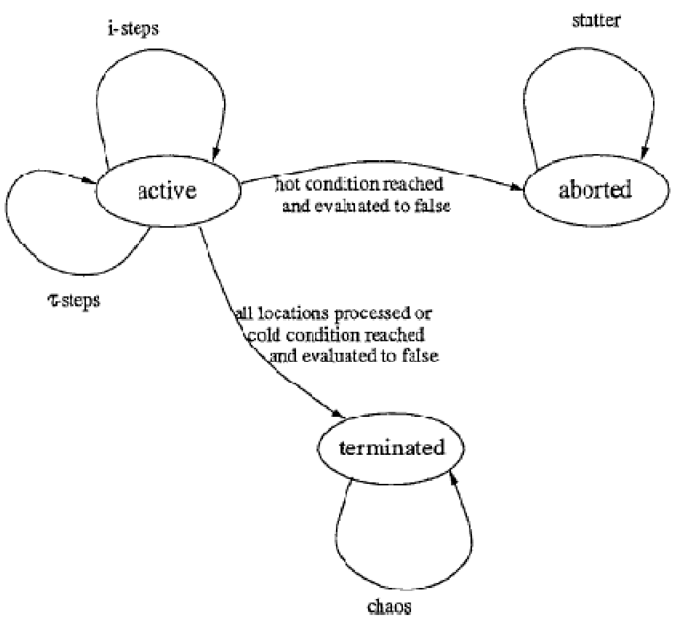

i-steps
stutter
active
hot condition reached and evaluated to false
aborted
τ-steps
all locations processed or cold condition reached and evaluated to false
terminated
chaos

Figure 5.2. Transition System Corresponding to a LSC [Source: Damm 1999]

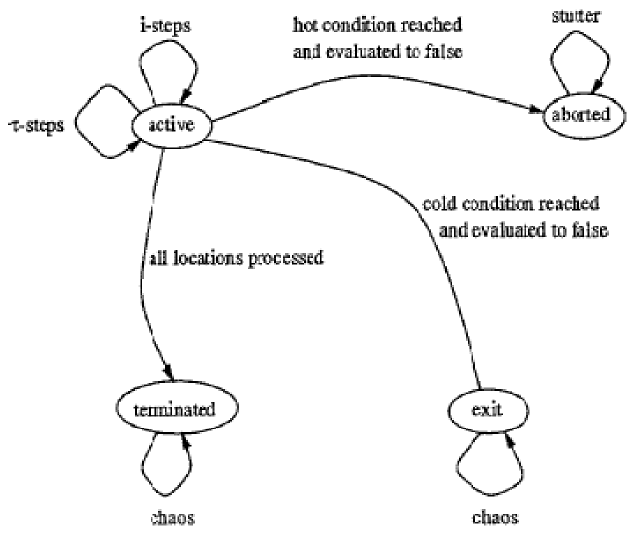

i-steps
hot condition reached and evaluated to false
stutter
τ-steps
active
aborted
cold condition reached and evaluated to false
all locations processed
terminated
exit
chaos
chaos

Figure 5.3. Transition System of LSC as a Pre-chart [Source: Damm 1999]

### 5.4.2 CMUML EXTENSIONS OF LSC SEMANTICS

The cmUML framework of the thesis adopts LSCs and the associated liveness semantics. In cmUML, the 'formal' part of interface specifications consist of a collection of *ScenarioContext* i.e LSCs. These LSCs are divided as *PrimaryContexts* and *SecondaryContexts*. The primary LSCs correspond to the principle behaviors of the *System* components and represent the corresponding use cases of the requirements phase. The secondary LSCs correspond to the necessary exception handling behavior expected of the *System* components in response to exceptions raised during its executions (or 'runs' in LSC parlance). In cmUML framework, the exceptions are first-class specification and behavioral entities. Thus the expected system *responses* w.r.t to raised exceptions during executions represent the part of system 'requirements'. The explicit separation of primary and secondary behaviors thus simplifies the specification, analysis, and development of systems. Initially only primary behaviors can be specified ignoring the secondary behaviors (or assuming no exceptions are raised). Though the thesis assumes the consistency between various parts of specifications these can also be easily verified (as described in next chapter).

The LSC semantics require extensions to incorporate the required semantics of the cmUML framework e.g. semantics of the synchronization and exception handling mechanisms as provided in cmUML. This can be done by adding necessary axioms in the transition relation of symbolic transition system that describes the formal semantics a LSC [Appendix B]. These extensions are described informally below:

- When a LSC is aborted due to violation of specified condition it may perform actions which have no effect on the 'runs' of the system (*stutter*) as defined in LSC framework or 'trigger' a 'secondary' LSC (specified in cmUML framework) using the exception raised in violation of the condition. The 'secondary' LSC specifies the expected exception handling behavior that should be observed in the 'run' of the component.

- When a condition 'C' is encountered along a single instance line, no instance progress steps are performed until the condition is satisfied. But the local steps can be performed which may actually effect the condition.

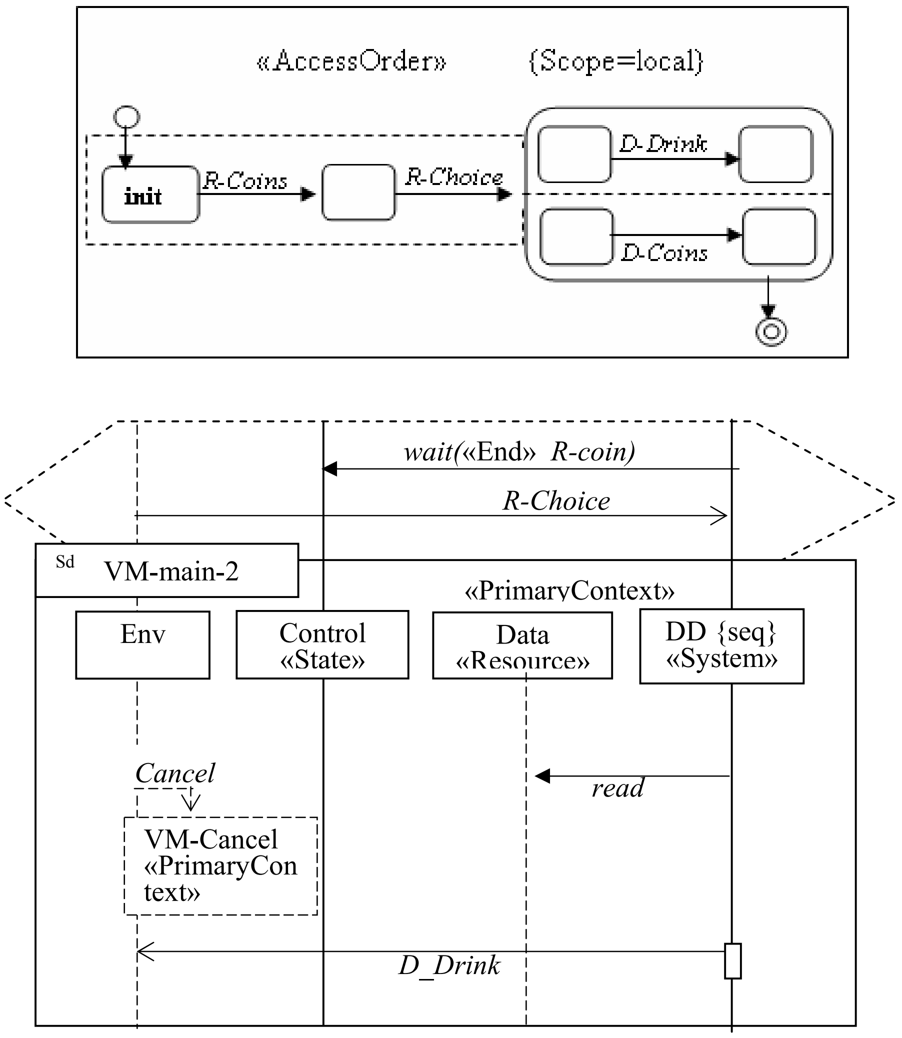


Figure 5.4. Partial Interface Specification in cmUML

- At a given instant, an execution of a cmUML component can be associated with a collection of 'active' *ScenarioContexts* to be satisfied. When all specified events i.e., both external and internal events of a partially ordered set satisfies the order, and liveness constraints as specified on a LSC it is considered 'complete' and removed from the collection. Thus a component starts with empty collection of LSC instances and said to have satisfied all specified behaviors when it becomes empty eventually while *AccessOrder* i.e., the corresponding interface statemachine reaches the final state or terminates.

## 5.5 SEMANTICS OF INTERNAL SPECIFICATIONS

The internal specification in cmUML is described by UML Activities and Statemachines. UML Activities represent sequential executions with data/control flow semantics while a statemachine possesses the reactive behavior. Thus the cmUML framework combines these formalisms to define a semantically precise execution or implementation models. Further the framework integrates an 'internal' exception handling mechanism in the fashion of Java's try-catch block [Gosling 1996] among the sequentially executing concurrent activities and the synchronized reactive behaviors. Further the semantic foundation of the internal specifications is extensible to customize for the adopted execution models and implementation environments providing flexibility to system design specifiers.

The cmUML framework adds precise semantics to UML Activities and Actions to specify semantics of sequential executions in concurrent and reactive environment. cmUML clarifies the semantics of executions of concurrent operations ( *Services* in cmUML) associated with a statemachine. As in [Ober 1999, Chrichton 2002] the execution of operations is de-linked from statemachine towards expressiveness in concurrency specification. Further the basic concepts of 'Action' and 'Activity' are extended as *GuardedAction*. The *GuardedAction* consists of various 'tags'. These tags and their values describe the liveness, synchronization, and exception behavior of actions or activities during execution [table 6.1, reproduced from chapter 3]. Further individual instances of *Service* executions (activities) can be referred in the specifications thus increasing the expressive power of the specification language. A *ServiceHandler* is associated with each *ServiceType* which contains the information about the current instances in execution using incarnation counters 'in' and 'out'. These *ServiceHandlers* help specification of complex synchronization patters, as described in [Mizuno 1999] e.g. barrier synchronization pattern etc.

| **Tag Value Combinations** | **The Implied Semantics** |
|---|---|
| isHot = *true* and guard = *false* | termination of the run raising the specified exception |
| isHot = *false* and guard = *false* | no effect on the 'run' of the corresponding *system* component |
| isHot = *true* and guard = *true* | effect of action on the run |
| isDelay = *true* | wait semantics till *guard* is true |
| isAtomic = *true* | atomic effect of guard evaluation and action execution with no interleaving step in between in the corresponding run of the *system* component |

Table 5.1. GuardedAction Semantics

An internal specification constitutes the implementation or detailed aspects of its interface specification [Lamport 2000]. cmUML follows a generic approach in semantics definition of its internal specification allowing different kinds of semantics specializations. For example, various combination of Finite State Machines, represented by UML behavioral statemachines and other kinds of concurrency models e.g. data/ control flow, synchronous reactive etc are described in [Girault 1999]. These execution models and associated semantics can be easily adopted in cmUML framework.

The formal semantics of the internal specifications is described using the formalism of Symbolic Transition Systems (STS) [Damm 2002]. The formal foundations of UML as defined in krtUML approach [Damm 2002] are adopted for the semantic description of cmUML. While the krtUML adopts the sequential execution in components, cmUML framework proposes a generic, extensible, multi-threaded semantics (though with reduced verification capabilities).

Further the semantic description is organized as *semantic modules* corresponding to the major abstractions of the cmUML framework (instead of first order predicate logic, as in krtUML approach, cmUML framework uses simple C-like programs for description of these semantic modules). These abstractions are namely PORT, SERVICE (referred as OP for brevity), STATE. The semantics of these abstractions are defined in terms of executions of instances of corresponding semantic modules (configuration instances e.g. PORTconf, OPconf, STATEconf with corresponding behavior types $T_{port\text{-}conf}$, $T_{op\text{-}conf}$, $T_{state\text{-}conf}$.

The semantic is defined in terms of atomic actions combined in imperative style. The instances of semantic modules execute in *run-to-completion* steps. The over all execution model of a 'system' component in cmUML is represented by an interleaving of the run-to-completion steps of corresponding semantic modules. Further two 'system' components (or sub-components) may execute concurrently with arbitrary interleaving.

## 5.5.1 SYMBOLIC TRANSITION SYSTEMS

Symbolic transition systems (STS) allow for purely syntactical description of a transition system over a set of typed system variables. A symbolic transition system (STS) S = (V, θ, ρ) consists of a finite set of typed system variables (V), a first-order predicate (θ) over variables in V characterizing the initial states, and ρ, a transition predicate. An STS induces a transition system on the set of interpretations of its variables as follows.

## 5.5.2 FORMAL REPRESENTATION OF INTERNAL SPECIFICATIONS

In general, defining semantics of a language L (cmUML) involves defining a mapping M between the syntactic structures of L and concepts of chosen semantic domain S (symbolic transition system). For this, a formal notation for specifications in L i.e. cmUML, is needed. An instance of a cmUML specification (internal) represents a collection of dynamically created sequential executions (*Services*) and a reactive (*State*) behavior. Services, corresponding to the triggered operations, represent independent threads-of-executions executing a sequence of actions (analogous to procedures in programming language frameworks). The reactive behavior represents an event-driven task, processing incoming messages/ events under a specified scheduling policy (default FIFO) by dispatching them for execution as specified by the statemachine in *run-to-completion* (*rtc*) steps.

Formally, an internal specification 'M' of cmUML framework is a 10-tuple:
M = (T, Act, Att, Expr, F, E, P, S, C) where

- T: A set of basic types and types for STATE, PORT, ENV and OP classes
- Act: The finite set of UML *actions*
- Att: A finite set of typed attributes of M
- Expr: A finite set of expressions *expr* over Att in first order logic defining the expression language for the model. An *expr* is a term defined in the scope of an object and used as transition guard, or invariant, or assertion
- F: $F_t \cup F_p$ contains the predefined types for triggered and primitive operations (defined below)
- E: A class of the type $T_{ENV}$, E = (e.Attr, e.Seq)
  - e.Seq is {<$mes_i$, $t_i$>} (i.e. a sequence of) represents messages $mes_i$ sent/ received at time $t_i$
- P: A class of the type $T_{PORT}$, P=(p.Attr, p.Seq, p.Acq)
  - p.Seq is {f} over $F_t$ defining the allowed temporal order of operation invocation
  - p.Attr includes the implicit attributes referring all OP objects and the STATE object
  - p.Acq is the set of acquaintances (i.e. references) representing external associations
- S: A class of the type $T_{STATE,}$ S= (s.Att, s.Expr, s.Act, s.Ops, *Assign*, Q, Tr)
  - Q is the set of finite states of s with $S_o$, $S_f$ as the initial, and final states
  - *Assign* represents a valuation of s.Att in a state

- s.Ops is the set of primitive operations
- Tr is the transition operation, $\subseteq$ Q x S.Expr x S.Act x Q
- s.Act is the set of UML actions specified on transitions as well as in primitive operations

- C: A finite, non-empty set of classes c, of the common super-type $T_{OP}$
  - c = (c.Att, c.Param, c.Ret, c.Tf, Pre, Post, L)
  - For each c$\in$C,$\exists$ corresponding type $t_c \in$T
  - c.Tf $\in F_t$ is the related triggered operation type
  - c.Param$\subseteq$c.Att, and c.ret$\in$c.Att represent the parameters and return values of c.Tf with corresponding types $type_{par}$(Tf) and $type_r$(Tf)
  - Pre, Post $\in$Expr are pre, post conditions of c.Tf
  - L is a finite {<$action_i$, $assert_i$,>} representing a flattened sequential method specification where $assert_i$ $\in$Expr and $action_i$ $\in$Act. $assert_i$, $assert_{i+1}$ represent the local pre, post assertions of $action_i$

For each operation p$\in$F, $type_{par}(p) = T_1 x \ldots x T_n$ denotes the parameter type where $T_i \in T(M)$ is the type of the i-th parameter and $type_r(p) \in T(M)$ is the type of the reply value. The type of p, $T_P$, is defined as $T_P = (type_{par}(p) \rightarrow type_r(f)$, *callKind*) where *CallKind*, is an enumerated value (of type $T_{CK} \in$T) defined by {*read, write, readPar, writePar*} representing the concurrency nature of the operation w.r.t itself as well as other operations (for e.g., *read* indicates multiple instantiations, *write* indicates single instantiation in isolation, and the suffix *par* groups a set of operations that can execute in parallel).

UML Actions are a finite set of fundamental actions (e.g object creation/ destruction, attribute assignment, operation calls etc). The definitions for expressions, guards and Actions can be given inductively as in [Damm 2002]. But, there is no need for association types in models of cmUML as all inter/ intra component associations are represented by the implicit variables. We also assume the following additional requirements for the models in cmUML.

- For each object o of type $T_{OP}$, o.Attr contains *self*, *state*, *port* referencing the corresponding objects
- For the object o of type $T_{PORT}$, o.Attr contains implicit attributes *self*, *state*, *acq*$_i$ *op*$_i$ as well as references to objects of type $T_{OP}$, corresponding to different invocations of triggered operations
- For each f $\in F_T$ , there exist a c $\in$C (with c.Tf = f) containing local attributes to hold the parameter and return values as well as the specification of corresponding method in terms of c-expressions and actions
- For each operation f$\in F_P$ the STATE object o contains local attributes to hold the parameter values, return value, and a transition (q, f, expr, q') $\in$o.Tr
- The STATE object does not make a call to its own triggered operation

### 5.5.3 DESCRIPTION OF FORMAL SEMANTICS

The behavioral semantics of an internal specification, say M, in cmUML is defined in terms of transition axioms of corresponding symbolic transition system (STS) $S_M \equiv (V_M, \theta_M, \rho_M , L_M)$. The type systems of the symbolic transition system i.e. T(S) is completely defined. The system variables V completely capture a dynamic execution of M. The semantics is described in terms of intuitive semantic modules in an imperative style. These modules define the execution semantics of configurations of semantic types $T_{PORT}$, $T_{STATE}$, $T_{OP}$ corresponding to the major abstractions of the cmUML framework. These semantic modules define the final transition relation $\rho_M$. These modules include liveness axioms. Thus the modules together fully define a transition system for corresponding concurrent system. The sets of initialization predicates, transition predicates, liveness axioms across all modules are collectively referred as $\theta$, $\rho$ $L$ respectively. A *snapshot* s of the transition system corresponding to the specified system is the evaluation of variables of V at a given instant.

All the system types, variables are informally defined and described below (the formal definitions of these semantic types and their corresponding domains can be skipped as these can be done on similar lines as in [Damm 2003]):

- Semantic configurations of types $T_{OP\text{-}conf}$, $T_{PORT\text{-}conf}$, $T_{STATE\text{-}conf}$ (i.e *OP-conf$_i$*, *PORT-conf*, *STATE-conf*,) respectively represent the semantic entities corresponding to behavior types $T_{ENV}$, $T_{PORT}$, $T_{STATE}$, $T_{OP}$
- *sconf*, a variable of type $T_{sconf}$, contains all instances of the semantic configurations

A semantic configuration fully captures the execution behavior of the corresponding semantic entity (subsuming the types of the corresponding entities). Necessary system variables and types are defined in simple form: variable_name(domain type or values). For simplicity, the null value of any type is represented by ε;

Msg-type $=_{def}$ (source($T_{PORT} \cup T_{ENV}$), dest($T_{PORT}$), type{*call,reply*}, mode{*synch, asynch}*,
p($F_t$), args($Type_{par}$(p)), ret($Type_{ret}$(p)$\cup$*void*))

Event_type $=_{def}$ (dest($T_{OP}$),p(Fp),args($Type_{par}$(p)), ret($Type_{ret}$(p) $\cup$ *void*) )

*Sys-err*(Boolean) = *false;* *--variable initialization*

Assuming the services are ordered 1,2,…..,n., we define the following types and variables to capture the allowed execution *scenarios* corresponding to multiple invocations and instantiations (depending on *callKind* of the invoked services).

PAR-type $=_{def}$ an integer vector of size n

**Semantic Configurations of type $T_{op\text{-}conf}$**: captures the execution behavior of invoked services. Prior to creation the corresponding execution instance is considered to be *dormant*. Creation of a new service instance, corresponding to its invocation, will pick a dormant index of *sconf*. During execution, these instances may become *suspended* when waiting for completion of invoked services, through corresponding PORT, on other components

**Semantic Module *OPconf* :**

```
OP-status:    dormant, executing, suspended;
OP-variables:             status(OP-status),loc(integer), msg(Msg_type);
                 promises(Set of <Msg-type>);
OPinit:       status:= dormant; loc := 0; msg:= ε; promises:= ϕ;
OPstart:      status:= executing; Param := msg.args;
                          if (!Pre) then sys-err= 'true' else OPexecute;
OPexecute:    loc:= loc+1;
                          if (loc<= |L|) then OPcall ∨ OPlocal; else  OPreturn;
OPlocal;      execute 'action_loc'  (as per UML semantics)
                          if (!assert_loc) then sys-err='true' ; OPexecute;
OPcall:                   status := suspended;
                          state.eve-queue.enqueue(Event_type<self,action_loc.opname,
……… >);
                          promises := promises ∪msg;
                          while(status != executing) wait;
                          promises := promises – msg;  OPexecute;
OPreturn;     if (msg.mode == asynch ∧ msg.ret != NULL)
                              port.outqueue.enque(<port,   msg.source,   reply,   asynch,
msg.p, -, ret>);
                          else
                    port.outqueue.enque( <port, msg.source,  reply, synch, msg.p, -, ret>);
                          status=dormant;
OPlive:       OPreturn ∧ (OPreturn => promises = ϕ)
```

**Semantic Configurations of type $T_{port\text{-}conf}$**: Captures the execution status of the PORT instances. It contains two queues: *in-queue* for incoming messages and *out-queue* for return messages. In addition the configuration keeps track of number of instantiations executing in parallel using instantiation counters *$in_k$, $out_k$* per each specified service. An *eligible* (w.r.t to the currently executing methods) service request is removed from in-queue and corresponding method object is created for execution. Also corresponding *$in_k$* is incremented.

**Semantic Module *PORTconf* :**

PORT-status: *idle, synch-wait, triggering;*

PORT-variables: Status(PORT-status), in-queue(*Queue*), out-queue(*Queue*), $in_k$(integer), $out_k$(integer), synch-msg(Msg-type), msg(Msg-type);

$PAR_{CO}$(PAR-type)$\wedge$ $PAR_{CO}[i]=in_i - out_i$

*-Dynamic vector reflecting current execution scenario*

$PAR_{par}$(PAR-type) $\wedge$ $PAR_{par}[i]$=1if the triggered operation is of type '*writePar',* ≥1 if 'readPar', 0 for other. -- *a static vector representing group of parallel operations*

$PAR_i$(PAR-type) $\wedge$ $PAR_i[i]$=1 if the callKind of i-th operation is *write*,≥ 1 if it is *read* else $PAR_i \equiv PAR_{PAR}[i]$

PORT-definitions: *Compatible*(x(PAR-type), y(PAR-type)) = *true* if x, y are compatible component wise else *false*.

PORTinit: status := *idle*; in-queue, out-queue := ε; synch-msg, msg := ε; $in_k$, $out_k$ := 0 ∀k;

```
PORTtrigger:      msg := choose(first(Msg-type ∈in-queue):
              msg.type=call ∧compatible($PAR_{CO}$, $PAR_{index(msg.p)}$ );.
              if(msg!= ε) then {status:= triggering; PORTinvoke;}
PORTinvoke:  j:=choose(first(integer): sconf[j]($T_{OP}$) ∧ sconf[j].status=dormant)
                  k := index(msg.p); $in_k$ := $in_k$+1; $OP_j$.msg := msg;
fork($OP_j$.OPstart);
PORTsend:    if(out-queue != ε) msg=first(out-queue);
```

```
                    if (msg.mode = *synch*)  msg.dest.synch-msg := msg;
                    else   msg.dest.in-queue.enqueue(msg);
PORTprocess:            while (*true*){ if(status=*synch-wait*)
                                   then PORTsend$\vee$PORTsynch;
                                        else PORTtrigger$\vee$PORTsend;}
PORTsynch: if(synch-msg != $\varepsilon$)
             then { *state*.synch- msg := synch-msg; *state*.status:= *rtc*; status=*idle*; }
PORTlive:  ($\forall$msg $\in$in-queue $\wedge$ msg.type=*call* $\exists$ PORTtrigger)$\wedge$
             ($\forall$msg$\in$out-queue$\exists$ PORTsend)
```

**Semantic Configurations of type $\mathbf{T_{state\text{-}conf}}$**: This represents the dynamic state-based synchronized execution behavior of a STATE behavior representing the *kernel* of the component. It contains an event queue. All the incoming messages i.e. from method objects (or the PORT object in future extension for trigger handling) are kept in the queue. The object continuously takes an *eligible* first event for processing in a *run-to-completion* (rtc) step. After processing the event is removed from the queue and the status of the corresponding service instance is changed to indicate the completion of its pending request.

**Semantic Module *STATEconf* :**

```
STATE-status: *idle, rtc, synch-wait;*
STATE-variables: status(STATE-status), eve-queue(Queue of Event-type);
                     synch-msg(Msg-type); eve(Event-type); state($\in$S.Q);
                     msg(Msg-type);
STATE-definitions:  *eligible*(eve)= *true* if $\exists$ a *firable* (tr $\in$Tr) and tr[2]=eve.p;
                                        *firable*(tr $\in$Tr ) = *true* if tr[1]=state and tr[3];
STATEinit: status:=*idle*; msg:= $\varepsilon$; eve-queue:= $\varepsilon$; synch-msg:= $\varepsilon$; eve:= $\varepsilon$; state:=$S_o$;

STATEtrigger: eve=*choose*(Event $\in$eve-queue: *eligible*(eve))
```

```
        if (eve != ε)  then {status := *rtc*; tr= *choose*(tr ∈Tr: firable(tr) ∧ tr[2]=eve.p);
            STATEaction; (state= tr[4]); STATErtc; eve.dest.status :=  *executing*;}

STATErtc:               STATEaction ∨ STATEprocess;
STATEaction:            STATEsend ∨ STATEprimitive or  STATElocal ∨ STATEnull;
STATEsend:                          STATEasynch-send ∨ STATEsynch-send;

STATEsynch-send:        status=*synch-wait*;
                        msg=*create*(Msg-type: msg.type=*call*  ∧ msg.mode=*synch*)
                         *port*.out-queue.enqueue(msg);
                        *port*.status=*synch-wait*; STATEreceive;

STATEreceive : while (status = *synch-wait*) *wait*; -- Assign synch-msg to local variables
                        STATErtc  ∨ STATEnull;
STATEasynch-send: msg=*create*(Msg-type:            msg.type=*call*  ∨ msg.mode=a*synch*)
                        *port*.out-queue.enqueue(msg);
STATElocal: tr= *choose*(tr ∈Tr: *firable*(tr) ∨ tr[2]= ε)
                if (tr != ε ) {STATEaction; state= tr[4]; STATErtc;}
STATEprimitive:  --do local actions/ primitive operations
STATEprocess: while(*true*){status=*idle*; *wait*; STATEtrigger; }
STATElive:   (∀ eve ∈in-queue ∃ STATEtrigger)
```

Thus the semantics of M is the STS(M) ≡ S (V, θ, ρ, *L*) where

System variables, V = {sconf, sys-err}

Initial condition, θ = ENVinit ∧ PORTinit ∧ STATEinit

Transition relation, ρ = ENVprocess ∧ PORTprocess ∧ STATEprocess

Liveness axioms, L=PORTlive ∧ STATElive  ∧ OPlive

## 5.6 SUMMARY

This chapter presented the semantic foundations for proposed cmUML framework. The semantic foundation is defined over a subset of elements of UML meta model and UML/ SPT Profile. As the cmUML framework separates the specifications in terms of interface and internal specifications, the semantic foundation is defined along two dimensions. The semantic foundation of interface specifications adopts LSC formalism and its semantics with necessary extensions to provide for exceptions and system responses as first class entities. The formal semantics of the internal specifications is defined using the formalism of Symbolic Transition Systems (STS). The formal semantics is close to the so called informal semantics of UML and UML/SPT profile elements used in cmUML profile. Further this includes the formal foundation of UML as defined in krtUML. While krtUML defines sequential execution in a component, cmUML adopts multi-threaded concurrent semantics in its ‘system’ components. Also formal semantics is defined in terms of executions of semantic modules corresponding to the main abstractions of cmUML profile. The executions of these modules are defined using a set of actions combined in imperative style. Wherever applicable the atomic actions of the semantic modules are chosen non-deterministically. Further each semantic module contains a basic liveness axiom.

# CHAPTER 6

# VERIFICATION APPROACHES

Testing has been the traditional approach of verification of implemented systems for correctness. But, with complexity of system behaviors increasing, an early phase of verification of system specifications saves cost and efforts towards system development. The lack of preciseness in early phases of development processes has been the main source of obstacle in integrating formal techniques towards early analysis of systems. With the Unified Modeling Language becoming the *de facto* industry standard language, many approaches and tools are proposed to integrate formal verification techniques in system development phases. But, the lack of formal semantics in UML has been the main source of obstacle in successful integration of formal techniques with UML towards analysis of critical system with concurrent and reactive behaviors.

This chapter presents the existing verification approaches that can be integrated with the proposed cmUML specification framework. The chapter presents an overview of two verification approaches: verification of interface specification by integrating LSC based verification techniques; verification by internal specification by translation into CSP formalism and related tools. LSC based verification technique can be used to verify consistency between various parts of the specification e.g. interface vs internal specifications. CSP based verification can be used to verify interface correctness, deadlock detection etc.

## 6.1 INTRODUCTION

One of the main purposes of software engineering is to enable developers to build systems that operate reliably despite their complexity. The formal methods community has developed many tools to help achieve this goal. With software rapidly growing in size and complexity, graphical specifications in languages like UML need to be formally verified, before the implementation phase in order to guarantee the development of more reliable systems. A few years ago the formal verification community began investigating mechanisms to integrate such graphical specifications with verification tools. While this approach achieved reasonable success on the translation of simple diagrams to model checkers' input notations, the results are not well integrated into general development environments. Further interpreting the results of verification is still highly human dependent particularly requiring developers to be expertise with the formal techniques.

The thesis has proposed a precise specification framework, namely cmUML, for concurrent and reactive systems. Investigations are made to find the suitable verification techniques, and tool environments that can be integrated with the cmUML framework. The separation of concerns approach in terms of interface specifications based on LSC formalism and internal specifications based on UML Activities and Statemachine facilitate integration of various verification approaches. For example, the LSC tool environments e.g. Rhapsody can be integrated for simulation and verification of interface specifications (e.g. consistency checks). In addition to simulation of cmUML specifications, various consistency checking scenarios are investigated like LSCs vs interface statemachines, LSCs vs Internal statemachines, and Interface vs internal statemachines.

## 6.2 LSC-BASED VERIFICATION OF cmUML SPECIFICATIONS

### 6.2.1 VERIFICATION OF INTERFACE SPECIFICATIONS

Interface specifications in cmUML mainly consist of a collection of *ScenarioContexts* (LSCs) and *AccessOrder* (a protocol statemachine). Various formal verification scenarios can be defined in cmUML framework. First, interface specifications can be verified with *PrimaryContexts* and the associated *AccessOrder*. To check that the specified behaviors of a *PrimaryContexts* are always 'satisfied', the model checking tool can translate the sequence diagram into an automaton (translation mechanisms exist), synchronized with the automaton corresponding to the *AccessOrder*, to inspect the evolution of the global states corresponding to the statemachine. The automaton corresponding to LSC evolves observing events in the statemachine, and when it reaches a final state (the last event in the sequence diagram), then the property as specified by the LSC is 'verified'.

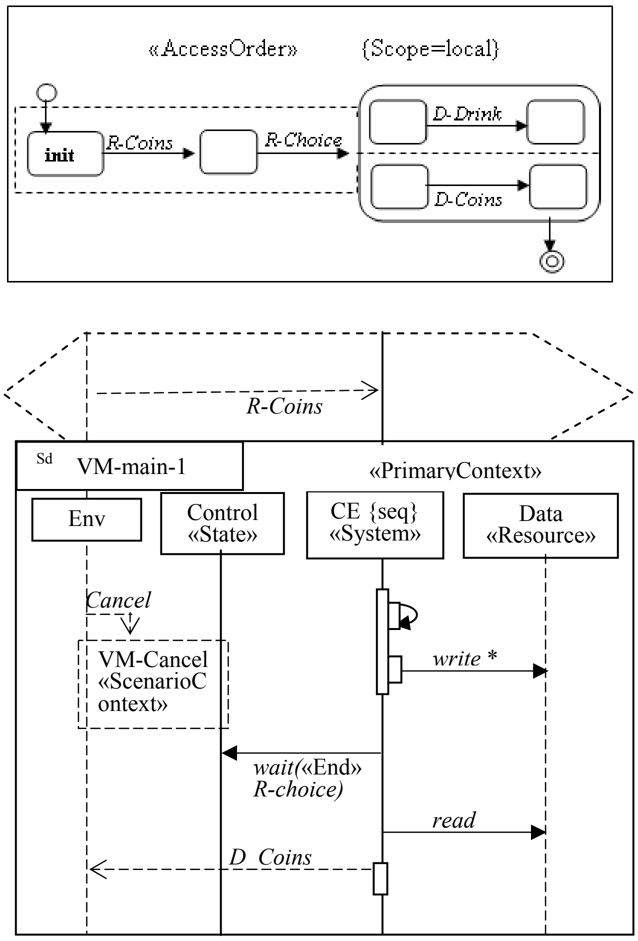


Figure 6.1 Verification of a 'PrimaryContext' against the Interface Statemachine

The verification process should carefully consider the situation when the system produces an unspecified event with respect to the LSC being verified. If the event is defined in the scope of LSC but is not specified, it can be ignored. If the event is associated with 'optional' semantics (i.e. 'cold' annotation) a choice may be presented to the user regarding processing of the event. If an event which is undefined in the scope of an LSC is observed a pre-defined exception can be raised or verification process terminated.

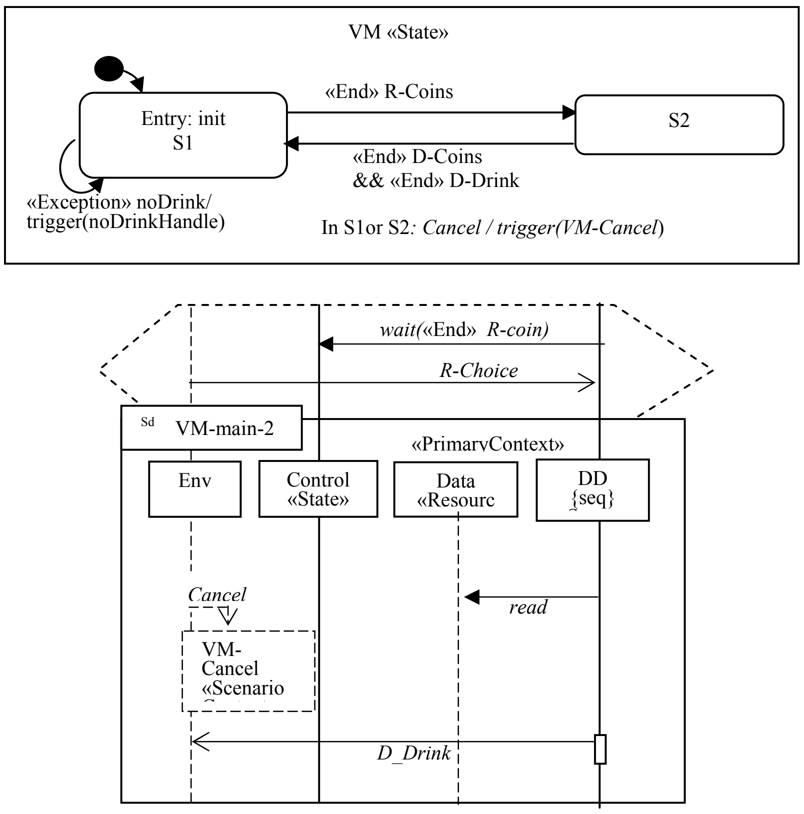


Figure 6.2 Verification of a *PrimaryContext* against a *State* Specification

A *PrimaryContext*, an LSC, describes the principle behavior corresponding to a use case (functional requirement). During initial phases of analysis the focus of verification are *PrimaryContexts*. The *SecondaryContexts*, again LSCs, represent the mandatory behavior that must be observed in response to ‘Exceptions’ raised during the execution of *PrimaryContexts*. As cmUML framework integrates the exception handling behaviors into the system specifications, the verification of *SecondaryContexts* is integrated into the verification of *PrimaryContexts*. For this, a simple extension mechanism, in the form of algorithm, is described in fig. 6.3.

**A Verification Algorithm**:

```
Verify (ScenarioContext X, FinisteStateMachine Y)

        Input: a ScenarioContext X, and a FiniteStateMachine Y
        Output: return true if X is 'terminated' else false if X is 'aborted'

        Construct an automaton A corresponding to X
                The states corresponds to 'cuts' of X as defined for LSCs
        Synchronize the automata A and Y

        Loop:  Trigger the execution of Y
                For each event ev generated
                Process the event in A in run-to-completion
                If status(A) = 'aborted' and error='exception'
                Trigger (SecondaryContext(exception))
                return false
                If status(A) = 'terminated' return true
                Go to loop
```

Figure 6.3. Verification algorithm integrating exception handling behaviors

The cmUML framework defines two statemachines for specification of a component i.e. interface statemachine ('AccessOrder'), and internal statemachine ('State'). The 'State' specification may be considered as more detailed version of 'AccessOrder' specification. Thus verification of 'AccessOrder' against 'State' specification constitute part of consistency checking or formal refinement between interface specification and corresponding implementation. Further the 'PrimaryContexts' defined in the interface specifications can also be verified against the 'State' specification of internal specifications. This further verifies the implementation of expected behaviors. Thus cmUML framework facilitates simple approaches to consistency checking.

## 6.3 VERIFICATION IN RHAPSODY TOOL ENVIRONMENT

The 'Rhapsody' UML verification environment supports verification of safety and liveness properties. The verification environment is integrated in the design tool "Rhapsody in C++", a commercial design tool offered by the company I-Logix, and is based on the VIS

(Verification Interacting with Synthesis) model checker [Harel 2002, VIS 1996]. Requirements or properties to be verified can be specified using Live Sequence Charts (LSC) [Damm 1999].

The interaction of the model with its environment is restricted to event communication. In order to specify the communication interface of the model the user has to define a set of events as being external. These external events are controlled by the model checker as inputs for the model. In order to restrict the possible environment behavior with respect to this event communication, the user of the verification environment can specify assumptions about possible event sequences provided by the environment using the specification techniques listed above. If the model checker detects a dynamic violation of a requirement specification, an error path is issued showing a concrete computation of the model violating the requirement.

To be able to use the VIS model checker both the model and the specification have to be transformed into the input formats of the model checker i.e., a finite state machine (FSM) description of the model and a computation tree logic (CTL) formula for the specification. An LSC specification can be translated into an adequate CTL formula. Both the FSM and the CTL formula are then fed into the VIS model checker, which either will state that the formula is true, or will produce a trace showing a violating run of the system. In order to become comprehensible, the trace is back translated into UML terminology so that model-constituents like objects, associations, and event queues become visible again. On the other hand, the event communication between the objects of the model which led to the contradiction of the specification is displayed as an LSC.

For verification purpose, the problem statement is refined with further events and features: drinks are sold at 50p (water), 1 re (soft drink), and 1.5 re (tea). The machine hold at most three drinks of each kind, but it can be refilled by the external event ‘FillUp’. This event then enables those drink lamps for which an adequate amount of money was already inserted into the machine. The detailed behavioral statemachine with all the detailed events is given below (assume the states C50(water), Re1orC100(drink), C150(tea) enable, as entry actions, the corresponding buttons only when there is the corresponding drink available; 'additional' coins are discarded during 'self' loops):

An important property that can be verified for vending machine specification is: "Whenever a customer wants to buy a water drink (thus, inserts at least one 50 ps coin followed by pushing the water button) and the VendingMachine is not out of water, then water is prepared and dispensed to the customer".

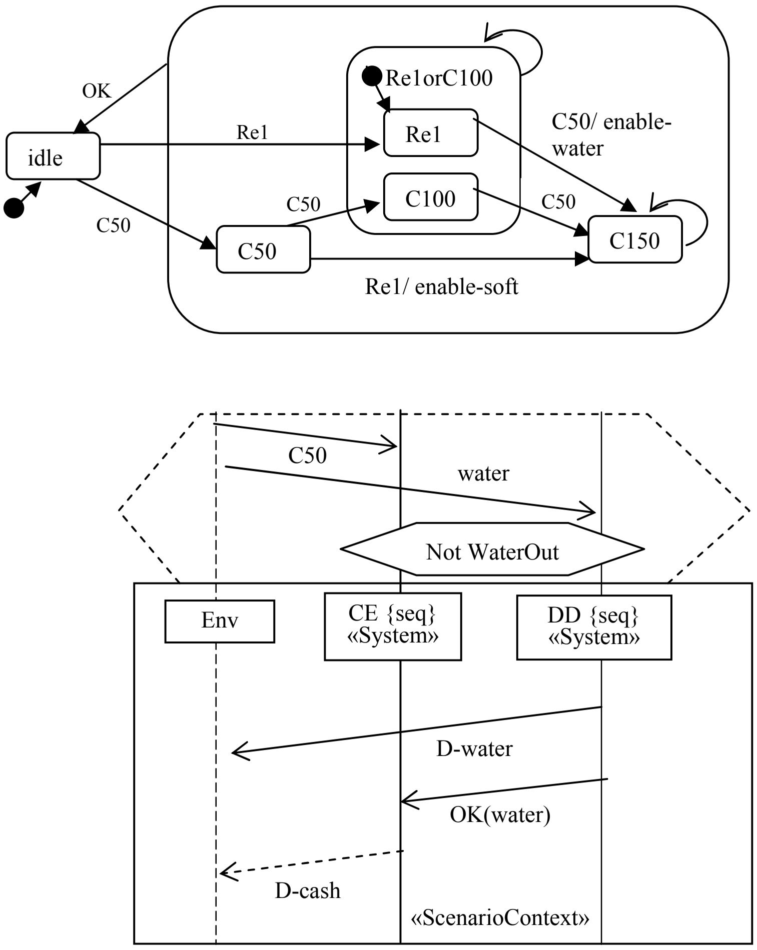


Figure. 6.5. Property Verification in Rhapsody with LSCs and Statemachine

Regarding the Vending Machine example, the LSC specification does not hold. The produced error path consists of several steps where initial steps might drive the model into a situation where no more water drinks are available (not shown in the figure). Starting at the 'Idle' state and inserting the first 'C50' leads to the state 'C50'. This event also "activates" the pre-chart. Note that the water lamp is not enabled since there is no water in stock. The next 'C50' leads to 'C100'. Now, the 'Re1' forces to take the self-loop of the or-state 'Re1orC100' which leads to state 'Re1' by taking the default transition. Crucial for recognizing the design error is the fact, that the internal state of the CoinExchanger has changed although the Re1 coin itself has been directly given back to the customer. The following 'FILLUP' enables the buttons of the ChoicePanel depending on the information about already inserted coins. But since the statechart of the CoinExchanger now encodes the fact that only a 1 Re coin was inserted the water lamp is not enabled. The first 'WATER' event then "concludes" the prechart (since in particular the machine is not out of water on the reception time of this event due to the previous refilling). The following 'WATER's mark the looping section of the infinite error path.

Thus cmUML framework approach supports the integration of industry standard tool environment 'Rhapsody' for verification purpose.

## 6.4 CSP BASED VERIFICATION

The previous sections describe the existing verification techniques that can be integrated with proposed cmUML framework for verification of interface specifications. This section describes an approach that can be integrated with cmUML framework for verification of internal specifications, in particular those of basic i.e. non-composite 'System' components.

The internal specifications in cmUML framework, which represents an abstract implementation, consists of data and control flow diagrams (represented by UML activity diagrams) and FSMs (represented by UML statemachines). The Activities represent the sequential executions and hence the computational aspect of the component. A generic, concurrent execution model is adopted towards the semantics of the internal specifications. The internal specification of cmUML components can have different semantics and execution models thus may represent different implementations corresponding to the same interface. Thus the associated verification approach for internal specifications should be generic and flexible to construct the required semantics and execution models.

The approach by Crichton et.al [Crichton 2002] describes a CSP based verification for concurrent models specified in UML. The approach also adopts a separation of concerns through separating executions of operations from statemachine behavior of an object. Thus operations are specified using Activities. The activity as well as the statemachine specifications can be translated into the notation of Communicating Sequential Processes (CSP) [Hoare 1985, Magee 1999] and verified using FDR model checking tool.

The interface specification approach in cmUML framework is closer to that of Cricton et.al. and hence can be easily mapped onto CSP specifications in similar manner. Thus cmUML internal specifications can also be formally verified for e.g. deadlock analysis.

### 6.4.1 CASE STUDY: A SIMPLE PRINTER SPECIFICATION

For the verification purpose a simple printer specification [Crichton 2002]. The problem can be specified in a similar manner in cmUML framework and the verification technique described is applicable as both the approaches are based on the separation of operations (specified as activity diagrams) from statemachine. The printer is described in terms of its five interface operations ('services' in cmUML); pause(), resume(), print(), service(), carelessService() described below

– pause() : pause any print job that might be in progress;
– resume() : resume any print job that is paused;
– print() : start a print job;
– service() : replenish the paper tray, pausing the printer first;

– carelessService() :replenish the paper tray, pausing the printer at the same time.

The first two operations can be treated as atomic (and hence can be specified on statemachine itself). The printer can be thought of having eight states i.e. the product of three conditions; printing or idle (1/0); paused, or not(1/0); and open or closed(1/0). In this the 'jam' state is undesirable (this state is reached from printing state (100) when tray is opend i.e. 'opentray' action occurs). Further a state is notified by appropriate events the effect of actions of operations e.g. 'change', 'call', events. These events can cause state transitions. It needs to be investigated whether this state is reachable in any situation of concurrent invocations of services.

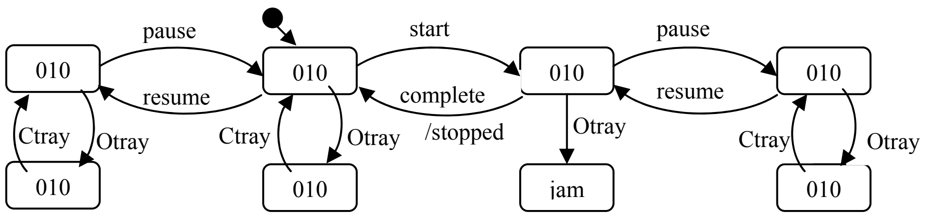


Figure 6.6. 'State' specification of «system» Printer [Source: Crichton 2002]

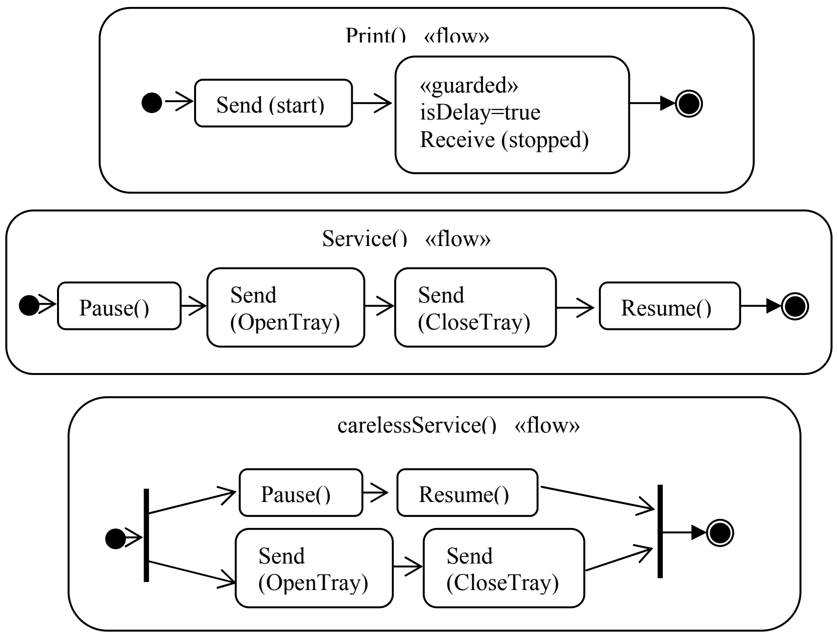


Figure 6.7. «service» Specifications of «system» Printer

Semantics of cmUML extensions e.g. 'guarded' actions are particularly intuitive and correspond to CSP semantics. These operations can be easily mapped onto CSP using latter's constructs e.g. 'process' and operators like □ (external choice), ||| (parallel execution but terminate same time), || (communication and synchronization on common events). The CSP translation of printer operations is given below.

$$Print = start \rightarrow stopped \rightarrow SKIP$$

$$Service = pause \rightarrow open \rightarrow close \rightarrow resume \rightarrow SKIP$$

$$\begin{aligned} CarelessService = {} & (pause \rightarrow resume \rightarrow SKIP) \\ & \;|||\; (open \rightarrow close \rightarrow SKIP) \end{aligned}$$

Similarly the «state» specifications can be translated into a set of CSP processes (as transitions represent individual processes). Translation of two states 'idle', and 'jam' is given below.

$$\begin{aligned} Idle = {} & start \rightarrow initialise \rightarrow Printing \\ & \Box\; open \rightarrow IdleOpen \\ & \Box\; pause \rightarrow IdlePaused \\ & \Box\; (\,\Box\, x : \{resume, close\} \bullet x \rightarrow Idle) \end{aligned}$$

$$\begin{aligned} Jammed = {} & (\Box\, x : \{pause, resume, open, close, start\} \bullet x \rightarrow Jammed \\ & \Box\; error \rightarrow Jammed) \end{aligned}$$

With the translated CSP models the refinement-checking tool FDR can be used to explore the consequences of the specified design and concurrency. To do this, a specification process is need to be defined, identifying a range of acceptable behaviors, and a variety of implementation processes, representing possible situations, or combinations of the model components. FDR checks whether these processes i.e. the 'Spec' and 'Implementations' are equivalent i.e. every trace, and every failure of the implementation is also a behavior of the 'Spec'. For the 'printer' example these are given below.

$$Interface = \{open, close, pause, resume, start, stopped, initialise, complete\}$$

$$Spec = \Box\, e : Interface \bullet e \rightarrow Spec$$

$$System1 = Print \parallel Printer$$
$$System2 = (Print \mathbin{|||} Service) \parallel Printer$$
$$System3 = (Print \mathbin{|||} CarelessService) \parallel Printer$$
$$System4 = (Print \mathbin{|||} Service \mathbin{|||} Service) \parallel Printer$$

Implementation processes *System*1 and *System*2 describe situations in which a single invocation of print(), and the simultaneous invocation of print() and service() act upon the printer state. In each case, the refinement check succeeds: *error* is impossible; the *Jammed* state is never reached. *System*3 describes a situation in which print() and carelessService() may be invoked simultaneously. In this case, the refinement check fails, and the tool returns as evidence the sequence <*start, initialise, open, error*> to show how the *Jammed* state could be reached. Similarly, when we check *System*4, which describes the effect of invoking print() concurrently with two invocations of service(), we are presented with the sequence <*pause, open, close, pause, resume, start, initialise, open, error*>: *Jammed* is reachable here, too.

## 6.5 SUMMARY

This chapter has presented a few existing verification methods, tool environments that can be integrated with the proposed specification framework, cmUML. For verification of interface specifications it is shown that LSC based environments can be adopted with minor extensions. Further an approach for integration of exception handling behavior during verification process is described. Various consistency checking scenarios are described. Also, an industry standard tool environment, Rhapsody, is found suitable for property verification of cmUML specifications. For verification of internal specifications, a CSP based translation approach is found suitable.

# CHAPTER 7
# CONCLUSIONS

The main objective of the thesis is to arrive at an abstract framework in UML that would allow concurrent abstractions to be used effectively from specification to implementation. The thesis has presented a formalism referred to as cmUML. The cmUML framework models separate external observable behaviors from internal implementation behaviors of the systems. These hierarchical specifications enable use of various kinds of verification approaches such as interface correctness, property verification, liveness, deadlock etc.

## 7.1 THESIS SUMMARY

The proposed cmUML framework retains the intuitive design notations of multi-view, graphical language UML for precise interface semantics and abstract implementation level models. The specification framework includes constructs for explicit specification of liveness, concurrency, synchronization and exception handling. Further the specification of these aspects is consistent in terms of both interface, and internal specifications. The operational semantics of cmUML has been defined using the formalism of Symbolic Transition Systems. The semantics of cmUML integrates low level UML action semantics with higher constructs of UML like Statecharts, activity and sequence diagrams and the underlying object model – thus providing a consistent view of system descriptions in spite of multi-view graphical specifications. The proposed specification language, namely cmUML defined for the specification phase is based on UML's light weight extension mechanisms (known *as stereotypes, tags,* and *constraints*) thus facilitating compatibility with existing UML tools. Thus, the framework can be used within existing UML based methodologies towards precise specification of complex systems with concurrent, reactive behaviors at an early phase of development processes.

The cmUML framework and corresponding specification language are based on UML metamodel and the conceptual model of UML/SPT profile (known as Profile for Schedulability, Performance, and Time). The framework addresses the limitations of both UML and UML/SPT profile for the precise specification of concurrent, reactive systems. The cmUML framework defines higher level abstractions with precise semantics based on the dynamic elements of the SPT profile. A conceptual model of the framework is presented together with a formal mapping between the elements of the conceptual model and those of SPT profile and UML meta model. Also the informal description of the semantics of the proposed specifications is described.

As a refinement of the proposed specification framework, the thesis has presented a step-wise specification process that can be applied to develop hierarchical specification of systems (or subsystem components) using proposed cmUML framework and its profile. The process is demonstrated using the case study of a Vending Machine specification. Further the advantage of the cmUML approach is demonstrated by comparing and validating it with other approaches, both formal as well as semi-formal. For this, classical problems of concurrency i.e., readers-writers problem and producer-consumer problem are specified and compared against their corresponding specifications in UML and formal approaches.

Though UML has become the *de-facto* industry standard language for specification of software systems, the UML models are not defined a formal semantics making the specifications ambiguous and inconsistent. To avoid this, a formal semantics is defined for the proposed framework. The formal semantics of cmUML framework is largely based on so called informal semantics descriptions of UML and UML/SPT Profile. Thus the semantics definition approach addresses the problem of ambiguity and inconsistency in UML specifications but retains the open framework philosophy of UML by not imposing the so-called concrete semantics. The formal semantics is described separately for both interface and internal specifications. The semantics of the interface specifications are based on LSC framework while the semantics of internal specifications are described using the formalism of Symbolic Transition Systems.

This thesis has also described the existing verification approaches that can be integrated with the proposed cmUML specification framework: verification of interface specification by integrating LSC based verification techniques; verification of internal specification by translation into CSP formalism and related tools. LSC based verification technique can be used to verify consistency between various parts of the specification e.g. interface vs internal specifications. CSP based verification can be used to verify interface correctness, deadlock detection etc.

In nutshell, cmUML combines the basic elements of UML, and the conceptual foundation of UML/SPT (causality and concurrency) under a unifying framework towards higher level abstractions for specification of complex systems. The framework integrates low level UML actions semantics and higher formalisms of UML like statechart, activity, and sequence diagrams with the underlying object model of UML retaining the latter's intuitive, multi-view design notations. Also, the framework doesn't constrain the system developers with fixed semantics, execution models, and implementation level constructs. Thus the cmUML framework provides a rigorous yet intuitive specification phase for precise specification of externally visible behaviors of concurrent and reactive systems i.e., concurrency, reactivity, exception handling, and synchronization. Further, the framework is closely integrated with requirements phase facilitating validation of the requirements against the system specification. The applicable verification strategies are presented.

## 7.2 CONTRIBUTIONS AND LIMITATIONS

Following contributions are made by this thesis work.

- Specification framework in Unified Modeling Language, namely cmUML, bridging requirements and early design for hierarchical specification of systems with concurrent and reactive behaviors.

- A specification language, namely cmUML Profile, using standard lightweight extension mechanisms of UML. Several stereotypes with precise semantics and specification context are defined. Extensions of UML Action and Activity as **GuardedAction** to specify precise semantics of action executions in concurrent environment. This gives implementation level expressiveness to internal specifications.
- Integration of UML/SPT profile and UML metamodel towards a unifying framework for lower level constructs e.g,,UML actions and higher level formalisms e.g., statecharts, activity, sequence diagrams. The integration further resolves ambiguities and inconsistencies of UML semantics. It further provides precise contexts for use of UML behavioral diagrams.
- Definition of semantic foundations for the proposed framework through adoption, and extensions of existing semantic frameworks like LSCs and krtUML. LSC foundation allows specification of liveness constraints. krtUML foundations define precise semantic foundations for UML action semantics.
- Integration of exception handling mechanism in specification and verification. The mechanism is comparable to Java's *try-catch* mechanism
- A specification process, in terms of specification tasks and heuristics, for application of the proposed framework
- Validation of the framework w.r.t both existing UML and formal approaches
- Investigation of existing verification approaches for integration with the proposed framework

Some of the limitations are:

- The specification framework and approach has been validated with simple examples and case study. The applicability of the approach for real-world systems need to be experimented
- The specification framework and process is short of tool support
- Formal semantics, and applicable verification approaches need to be further investigated.

## 7.3 FUTURE WORK

While the proposed cmUML framework has been demonstrated and validated using well known specification problems and a case study, further investigations with large scale industrial strength case studies shall help refine the various aspects of the framework. Though applicability of formal techniques has been investigated through appropriate extensions of the corresponding semantical frameworks, further investigations shall strengthen the semantic framework of cmUML. The cmUML framework can be extended to Real-Time embedded systems. As the framework is defined using light weight extension mechanisms of UML, it can be integrated with other UML-based methods and approaches.

# REFERENCES

[Allen 1997] R. Allen and D. Garlan, “A Formal Basis for Architectural Connection”, ACM Trans. Software Eng. and Methodology, vol. 6, no. 3, pp. 213-249, July 1997.

[Alpern 1985] B. Alpern, and F. Schneider, “Defining Safety and Liveness”, Information Processing Letters, 21(4):181-185, October, 1985

[Alvarez 2001] J.M. Alvarez, T. Clark, A. Evans, and P. Sammut., “An Action Semantics for MML”, In Proc. UML 2001, 2001.

[Bass 2003] L. Bass, P. Clements, R. Kazman, and K. Bass., “Software Architecture in Practice”, 2nd Edition, Addison-Wesley, 2003.

[Berry 1991] A. Benveniste and G. Berry, “The synchronous approach to reactive and real-time systems,” in *Proc IEEE*, vol. 79, pp. 1270–1282, Sept, 1991.

[Belina 1991] F. Belina, D. Hogrefe, and A. Sarma, “SDL with Applications from Protocol Specification”, Hemel Hempstead, U.K.: Prentice-Hall International, 1991.

[Borger 2000] E. Borger, A. Cavarra, and E. Riccobene., “Modeling the Dynamics of UML State Machines”, International Workshop, ASM 2000, Proceedings, volume 1912 of LNCS, pages 223-241. Springer-Verlag, 2000.

[Borger 2000] E. Borger, A. Cavarra, and E. Riccobene., “An ASM Semantics for UML Activity Diagrams”, In T. Rus, editor, Proc. AMAST 2000, volume 1816 of LNCS, pages 293-308. Springer-Verlag, 2000

[Bryan 1992] D. Bryan, “Rapide{0.2 language and tool-set overview", Technical Note CSL{TN{92{387, Computer Systems Lab, Stanford University, Feb. 1992.

[Cheng 2001] S. Cheng, and D. Garlan, “Mapping Architectural Concepts to UML-RT”, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, Las Vegas, USA, June, 2001.

[Cheng 2001] H.C. Cheng and W. E. Mcumber., “A General Framework for Formalizing UML with Formal Languages”, Proceed. *of the 23rd International Conference on Software Engineering (ICSE),* pp 433—442, Ontario, Canada, 2001

[Chinodo 1994] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A.Sangiovanni Vincentelli, “Hardware-software codesign of embedded systems,” *IEEE Micro*, pp. 26–36, Aug. 1994.

[Clark 2000] R.G. Clark, A.M.D. Moreira: Use of E-LOTOS in Adding Formality to UML. J. UCS 6(11): 1071-1087 (2000)

[Clarke 2003] E. Clarke, E. Mota, A. Groce, W. Oliveira, M. Falcao, J. Kanda., “VeriAgent: an Approach to Integrating UML and Formal Verification Tools”, *Sixth Brazilian Workshop on Formal Methods (WMF 2003)*, pages 111--129, Universidade Federal de Campina Grande, Brazil, October 2003.

[CoFI 1997] CoFI Task Group on Language Design, “CASL Summary Version 1.0”, Technical Report, 1998

[Crichton 2002] C. Crichton, J. Davies, and A. Cavarra, “A Pattern for Concurrency in UML”, Oxford Computing Lab, submitted in FASE 2002

[Damm 1999] W. Damm, D. Harel, “LSCs: Breathing life in to Message Sequence Charts”, In Porc. 3rd IFIP International Conference on Formal Methods for Open Object-based Distributed System, 1999

[Damm 2002] W. Damm, B. Josko, A. Pnueli, and A. Votintseva., “Understanding UML: A formal semantics of concurrency and communication in real-time UML”, In Proceedings of FMCO’02, LNCS. Springer Verlag, November 2002

[Damm 2003] W. Damm and B. Westphal., “Live and Let Die: LSC-based Verification of UML Models”, In Proceedings FMCO'02, First Int. Symp. on Formal Methods and Components and Objects, Netherlands, Vol. 2852, Lecture Notes in Computer Science, 2003.

[Dennis 1975] J. B. Dennis, “First version data flow procedure language,” Massachusetts Inst. Technol. Lab. Comput. Sci. Tech. Memo MAC TM61, May 1975.

[Douglass 2004] B.P. Douglass, “Real-Time UML: Advances in the UML for Real-Time Systems”, 3/e, Pearson Ed., New Delhi, 2004

[Engels 2000] G. Engels, J.H. Hausmann, R. Heckel, and S. Sauer, “Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML”, In Proceed. 3rd International Conference on the UML 2000, October 2000.

[Evans 1998] A.S. Evans and A.N. Clark., “Foundations of the Unified Modeling Language”, In 2nd Northern Formal Methods Workshop, Ilkley, electronic Workshops in Computing. Springer-Verlag, 1998.

[Evans 1999] A. Evans, R. France, K. Lano, and B. Rumpe., “The UML as a Formal Modeling Notation”, In The Unifed Modeling Language: the first international workshop, June 1998. Springer-Verlag, 1999

[Girault 1999] A. Girault, B. Lee, and E. A. Lee, “Hierachical Finite State Machines with Concurrency Models”, IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, Vol. 18(6), 1999

[Glasser 1997] U. Glässer and R. Karges. Abstract State Machine semantics of SDL. Journal of Universal Computer Science, 3(12):1382--1414, 1997

[Gomaa 1993] H. Gomaa, “Software Design Methods for Concurrent and Real-Time Systems”, Addision-Wesley Publishing Company, Reading Massachusetts, 1993

[Gomaa 2000] H. Gomaa, “Designing Concurrent, Distributed, and Real-Time Applications with UML”, Addison-Wesley Longman Publishing Co. 2000

[Goni 2004] A. Goni, Y.Eterovic, “Building Precise UML Constructs to Model Concurrency Using OCL”, UML 2004 Conference, LNCS Vol 3273, pp 212-225, 2004

[Gosling 1996] J. Gosling, The Java Language Specification, Addison-Wesley, 1996

[Guttag 1985] J.V.Guttag, J.J. Horning, and J.M. Wing, “The Larch Familyl of Specification Languages”, IEEE Software, Vol.2, No.5, Sept. 1985, pp. 24-36

[Hall 1990] A. Hall, “Seven myths of formal methods,” *IEEE Software*, Vol. 6, No. 9, pp. 11–19, 1990.

[Harel 1987] D. Harel, “Statecharts: A visual formalism for complex systems,” *Sci Comput. Program.*, vol. 8, pp. 231–274, 1987.

[Harel 2002] E. Gery, D. Harel, and E. Palachi, “Rhapsody: A complete life-cyclemodel-based development system”, In Proceedings of the Third International Conference on Integrated Formal Methods, pages 1–10, 2002.

[Hansen 1978] P. Brinch Hansen, “Distributed processes: A concurrent programming concept”, Communications of the ACM 21, 11, pp934-941, November 1978

[Hoare 1985] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall Int., 1985.

[Hussmann 2002] H. Hussmann, “Loose Semantics for UML, OCL”, In Proceedings 6th World Conference on Integrated Design and Process Technology (IDPT 2002). Society for Design and Process Science, June 2002.

[ILogix 2002] I-Logix Inc, Rhapsody, 2002.

[ITU 1994] ITU-T Recommendation Z.100. *Specification and Description Language (SDL)*. International Telecommunication Union (ITU), Geneva, 1994

[ITU 2000] ITU-T, “SDL combined with UML”, ITU-T recommendation Z.109, 2000

[Jagadish 2006(a)] S. Jagadish, R.K. Shyamasunder, “An UML-based approach to Specify Secured, Fine-grained, Concurrent Access to Shared Resources” Journal of Object Technology (JOT), vol.6 no.1, Jan-Feb, 2007, pp 107-119.

[Jagadish 2006(b)] S. Jagadish, R.K. Shayamasundar, “cmUML- A Precise UML for Abstract Specification of Concurrent Components”, Proceedings of 18th International Conference on Parallel and Distributed Computing and Systems (PDCS), Dallas, USA, Acta press, November 2006, pp 141-146.

[Jagadish 2007] S. Jagadish, L. Chung, R.K. Shyamasundar, “UML based Framework for Formal Specification of Concurrent, Reactive Systems”, Journal of Object Technology (JOT), to appear.

[Jones 1986] C.B. Jones, “Systematic Software Development Using VDM”, Prentice Hall International, 1986

[Kim 1999] S.-K. Kim and D. Carrington, "Formalizing the UML Class Diagrams Using Object-Z", In France and Rumpe, editors, Proc. UML'99, volume 1723 of LNCS, pages 83-98. Springer-Verlag, 1999.

[Kahn 1974] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceed. of the IFIP Congress 74*Amsterdam, The Netherlands: North-Holland, 1974.

[Kleppe 1999] J. Warmer, and A. Kleppe, "The Object Constraint language: Precise Modelling with UML", Addison-Wesley, 1999

[Kwon 2000] G. Kwon., "Rewrite Rules and Operational Semantics for Model Checking UML Statecharts", In Proceed. of the 3d International Conference on the UML 2000, University of York, October 2000.

[Lamport 1977] L. Lamport, "Proving the Correctness of Multiprocess Programs", IEEE Trans. on Software Engineering SE-3, 2(March 1977), 125-143

[Lamport 1983] L. Lamport, "Specifying Concurrent Program Modules", ACM Trans. Programming Languages and Systems, Vol.5, No.2, Apr. 1983, pp. 190-222.

[Lamport 1989] L. Lamport, "A Simple Approach to Specifying Concurrent Systems", Communications of ACM, vol.32 no.1, pp32-45, January 1989

[Lamport 2000] L. Lamport, "A Formal Basis for the Specification of Concurrent. Systems", Notes for the NATO Advanced Study Institute, Izmir, Turkey. June 26, 2000

[Larman 2001] C. Larman, "Applying UML and Patterns: An Introduction to Object-Oriented. Analysis and Design, and the Unified Process", 2/e. Prentice Hall, 2001

[Lilius 1999] J. Lilius and I.P. Paltor. vUML: a Tool for Verifying UML Models. Turku Centre for Computer Science, Abo Akademi University, Finland. Technical Report TUCS-TR-272

[Liu 2005] Z. Liu, and R. Venkatesh, “Tools for formal software engineering”, IFIP Working Conference on Verified Software: Theories, Tools and Experiments

[Lohr 1992] K. Lohr, “Concurrency Annotations”, Proc. on Object-oriented programming systems, languages, and applications, Canada, pp 327-340, 1992

[Luckham 1995] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann, “Specification and Analysis of System Architecture Using Rapide,º IEEE Trans. Software Eng., vol. 21, no. 4, pp. 336-355, Apr. 1995.

[Magee 1995] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, ªSpecifying Distributed Software Architectures,º Proc. Fifth European Software Eng. Conf. (ESEC '95), Sept. 1995.

[Magee 1999] J.Magee and J.Kramer., “Concurrency - State Models &. Java Programs”, Chichester: John Wiley & Sons, 1999.

[Manna 1991] Z. Manna and A. Pnueli., “The Temporal Logic of Reactive and Concurrent Systems Specification”, Springer-Verlag, New York, 1991.

[Mellor 2002] S.J. Mellor, and M.J. Valcer, “Executable UML: a foundation for model-driven architecture”, Addison-Wesley, 2002.

[Milner 1992] R. Milner, J. Parrow, and D. Walker, “A calculus of mobile processes, I,” *Inform. Computation*, vol. 100, no. 1, Sept. 1992.

[Mizuno 1999] M. Mizuno, "A structured approach for developing concurrent programs in Java", Information Processing Letters, Vol 69, No 5, pp232-238, 1999.

[Nenad 2000] N. Medvidovic and R.N. Taylor, “A Classification and Comparison Framework for Software Architecture Description Languages”, IEEE Trans. on Software Engg., Vol.26(1), January, 2000.

[Ober 1999] I. Ober, I. Stan, “On the Concurrent Object Model of UML”, Proceedings of EUROPAR’ 99, pp 1377-1384, Toulouse, France, 1999

[Ober 2001] S. Gerard, I. Ober, “Parallelism/ Concurrency Specification in UML“, white paper, UML Conference, Toronto, Canada, 2001

[Ober 2001] I. Ober., “Harmonizing Design Languages with Object-Oriented Extensions and an Executable Semantics”, PhD Thesis, Institut National Polytechnique de Toulouse,

[Ober 2004] I. Ober, S. Graf., “How useful is the UML real-time profile SPT without Semantics?”, In *SIVOES 2004, associated with RTAS 2004, Toronto Canada* 2004

[OMG 2001] OMG., “The Unified Modeling Language (UML) Specification - Version 1.4”, September 2001. Joint submission to the Object Management Group (OMG) http://www.omg.org/technology/uml/index.htm

[OMG 2002] Object Management Group, “UML Profile for Schedulability, Performance, and Time Specification”, OMG Adopted Specification ptc/02-03-02, July 1, 2002 (www.omg.org)

[Overgaard 1998] G. Overgaard and K. Palmkvist., “A Formal Approach to Use Cases and Their Relationships”, In UML 1998, 1998.

[Pankert 1994] M. Pankert, O. Mauss, S. Ritz, and H. Meyr, “Dynamic data flow and control flow in high level DSP code synthesis,” in *Proc. 1994 IEEE Int. Conf. Acoustics, Speech, and Signal Processing*, Australia, Apr. 19–22, 1994, vol. 2, pp. 449–452.

[Papathomas 1992] M. Papathomas, “Language Design Rationale and Semantic Framework for Concurrent Object Oriented Programming”, Ph.D Thesis, University of Geneva, 1992.

[Papathomas 1996] M. Papathomas, “ATOM – An Active Object Model for Enhancing Reuse in the Development of Concurrent Software”, RR 963-I-LSR-2, LSR-IMAG, Grenoble, 1996.

[Pnueli 1986] A. Pnueli, “Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends”, Lecture Notes in Computer Science 224, Springer-Verlag, NY, 1986, pp. 510-584

[Pnueli, 1997] A. Pnueli, and W. Damm, “Verifying out-of-order Executions”, Proce. Int. Conf. on Correct Hardware Design and Verification Methods (CHARME), Canada, pp 23-47, 1997

[Pressman 2004] R.S. Pressman, “Software Engineering: A Practitioner’s Approach”, Mc-Graw Hill Higher Eduction, 6/e, 2004

[Reggio 2000] G. Reggio, E. Astesiano, C. Choppy, and H. Humann., “Analyzing UML Active Classes and Associated State Machines - A Lightweight Formal Approach”, In FEAS 2000, 2000.

[Schniz 2004] I. Schinz, T. Toben, C. Mrugalla, B. Westphal, “The Rhapsody UML Verification Environment”, Proc. of 2nd Int. Conf. on Software Engineering and Formal Methods (SEFM’04), Beijing, China, pp 174-183, 2004

[Schmidt 1997] D. C. Schmidt, “Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software,” in *Handbook of Programming Languages* (P. Salus, ed.), MacMillan Computer Publishing, 1997

[Selic 1994] B. Selic, G. Gullekson, and P.Ward, “Real-Time Object-Oriented Modeling”, John Wiley, New York, 1994

[Selic 1998] B. Selic and J. Rumbaugh., “Using UML for modeling complex real-time systems”, available under http://www.objectime.com/uml, April 1998

[Selic 2004] Selic, B., “On the Semantic Foundations of Standard UML 2.0”, Lecture Notes in Computer Science vol. 3185, Springer-Verlag, 2004.

[Sendall 2001] S. Sendall, A. Strohmeier, “Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML”, in proceeding of UML conference, UML 2001, pp 391-405, Toronto, 2001.

[Shrotri 2003] U. Shrotri, P. Bhaduri, and R. Venkatesh., “Model checking visual specification of requirements”, *International Conference on Software Engineering and Formal Methods (SEFM 2003)*, page 202-209, Brisbane, Australia. IEEE Computer Society Press, Zurich, October 2005

[Shrotri 2003] U. Shrotri, P. Bhaduri, and R. Venkatesh., “Model checking visual specification of requirements”, *International Conference on Software Engineering and Formal Methods (SEFM 2003)*, page 202-209, Brisbane, Australia. IEEE Computer Society Press, Zurich, October 2005

[Spivey 1988] J. M. Spivey, “Introducing Z: A Specification Language and its Formal Semantics”, Cambridge Univ. Press, 1988.

[Vahid 1995] F. Vahid, S. Narayan, and D. D. Gajski, “Speccharts: A VHDL front end for embedded systems,” *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 694–706, June 1995.

[VIS 1996] The VIS Group., “VIS : A System for Verification and Synthesis”, In 8th international Conference on Computer Aided Verification, volume 1102 of LNCS, 1996.

[VonderBeeck 1994] M. von der Beeck, “A comparison of statecharts variants,” in *Proc. Formal Techniques in Real Time and Fault Tolerant Systems*, LNCS 863. Berlin, Germany: Springer-Verlag, 1994, pp. 128–148.

[Wing 1990] J.M. Wing, “A Specifier's Introduction to Formal Methods”, IEEE Computer, 23(9):8-24, September 1990

[Wing 1995] J.M. Wing., “Hints to Specifiers”, Technical Report, CMU-CS-95-118R, Carnegie Mellon University, USA, 1995

[Wing 1996] E.M. Clarke, J.M. Wing, “Formal Methods: State of the Art and Future Directions”, ACM Computing Surveys, 28(4):626{643, 1996.

# APPENDIX A
# SAFETY AND LIVENESS IN CONCURRENT SYSTEMS

Concurrent systems are analyzed in terms of their safety and liveness properties. Any property of these systems can be described in terms of their safety and liveness properties and analyzed using respective techniques. Safety and liveness were first described by Lamport [Lamport 1977]. Alpern and Schneider proposed formal characterization of safety and liveness properties of concurrent systems [Alpern 1985].

## A.1 CONCURRENT SYSTEMS AND PROPERTIES

A concurrent program execution can be viewed as a sequence $\sigma = s0s1s2...$ of states, each state $si$ (for $i > 0$) is the result of a single atomic action from $si$ -1. A set of such sequences is defined as a '*property'*. A property *P holds* for a program if the set of all sequences defined by the program is contained within the property. It is useful to distinguish two classes of properties, since they are proved using different techniques. A proof that a program satisfies a 'safety property' rests on an invariance argument, while a proof that a program satisfies a 'liveness property' depends on a well-foundness argument.

For formalization of safety, and liveness, let $S$ = set of program states, $S\omega$ = set of infinite sequences of program states, $S^*$ = set of finite sequences of program states. The execution of a program can be modeled as a member of $S\omega$. Thus the elements of $S\omega$ form 'executions'. Elements of $S^*$ are partial executions. Further $\sigma|= P$ if $\sigma$ is in property $P$. Let $\sigma i$ = partial execution consisting of the first $i$ states in $\sigma$.

## A.2 SAFETY PROPERTIES

Informally, a safety property stipulates that no “bad things” happen during program execution. Examples of safety properties (and their respective “bad things”) are mutual exclusion (two processes executing in the critical section at the same time), deadlock freedom (deadlock), partial correctness (starting state satisfied the precondition, but the termination state does not satisfy the postcondition), first-come-first-serve (servicing a request made after one that has not yet been serviced) etc.

For P to be a safety property, if P does not hold for an execution then at some point some ‘bad thing’ must happen. Such a ‘bad thing’ must be irremediable because a safety property states that the ‘bad thing’ never happens during execution. Thus following formal definition hold for a safety property P:

*P* is a *safety property* if and only if

$$(\forall\sigma\colon \in\sigma S\omega\colon \sigma|\neq P \Rightarrow (\exists i : 0\leq i\colon (\forall\beta\colon \in\beta S\omega\colon \sigma i\beta \,|\neq P)))$$

## A.3 LIVENESS PROPERTIES

Informally, a liveness property stipulates that  a “good thing” happens during program execution. Examples of liveness properties (and their respective “good things”) are starvation freedom (making progress), termination (completion of the final instruction), guaranteed service ( receiving service).

For P to be a ‘liveness property’, no partial execution is irremediable; a “good thing” can always occur in the future i.e. if a partial execution were irremediable, it would be a “bad thing” and liveness properties cannot reject “bad things”, only ensure “good things”.

Thus the following formal definition holds for liveness property P:

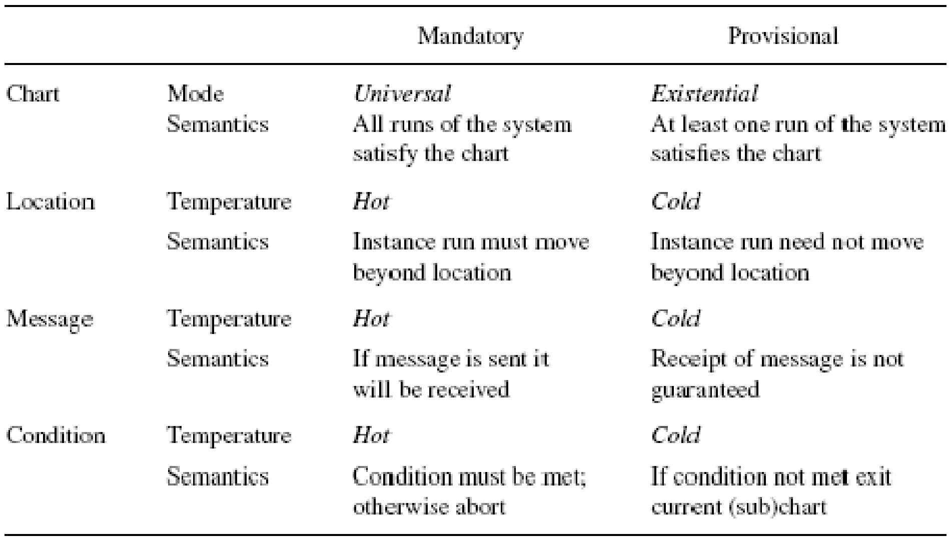
*P* is a liveness property if and only if $(\forall\alpha\colon \in\alpha S^*\colon (\beta\exists\colon \in\beta\, S\omega\colon \beta\alpha|{=}P)$

# APPENDIX B
# LIVE SEQUENCE CHARTS

While message sequence charts (MSCs) are widely used in industry to document the interworking of processes or objects, they are expressively weak, being based on the modest semantic notion of a partial ordering of events as defined, e.g., in the ITU standard. The language of LSC is a highly expressive and rigorously defined MSC language extension towards a serious, semantically meaningful behavioral specification of concurrent, reactive systems as well as providing tool support for formal analysis of system properties at early phases of development [Damm 1999]. Further LSCs address the central problems in behavioral specification of systems: relating scenario-based inter-object specification to state-machine intra-object specification.

## B.1 LIVE SEQUENCE CHART (LSC)

MSC specifications are typically used to capture sample scenarios corresponding to system functionalities (also known as use cases). At this stage the representational interpretation of MSC semantics is permissible. But as system becomes refined and conditions characterizing use cases evolve, the intended interpretation needs to distinguish *existential* (optional) behaviors from *universal* (mandatory) behaviors. Here the conventional semantics of MSCs is inadequate. LSCs allow the user to selectively designate parts of a chart, or even the whole chart itself, as universal (that is, live, or mandatory), thus specifying that messages have to be sent, conditions must become true, etc. The designer may incrementally add liveness annotations as knowledge about the system evolves. Hand in hand with this extension comes the need to support conditions as first-class citizens. 'Conditions' as first-class citizens, help in capturing assertions that characterize usecases. Conditions can thus qualify requirements as assertions over instance variables. Thus, LSCs provide the semantical basis for rigorous and complete consistency checks between the descriptive view of the system by sequence charts and the constructive one. Such checks could eventually be made using formal verification techniques like modelchecking



| | | Mandatory | Provisional |
|---|---|---|---|
| Chart | Mode | *Universal* | *Existential* |
| | Semantics | All runs of the system satisfy the chart | At least one run of the system satisfies the chart |
| Location | Temperature | *Hot* | *Cold* |
| | Semantics | Instance run must move beyond location | Instance run need not move beyond location |
| Message | Temperature | *Hot* | *Cold* |
| | Semantics | If message is sent it will be received | Receipt of message is not guaranteed |
| Condition | Temperature | *Hot* | *Cold* |
| | Semantics | Condition must be met; otherwise abort | If condition not met exit current (sub)chart |

Table. B.1. Summary of Liveness Notions in LSCs, with their Informal Meaning [Source: Damm 1999]

## B.2 FORMAL SEMANTICS OF LSCS

Formal semantics of a LSC ‘m’ is described by a symbolic transition system or a skeleton automaton A(m) [Damm 1999]. States of A(m) corresponds to ‘cuts’ in LSC induced by the partial order subsuming the constraints imposed by both the standard and the extensions. Each state or ‘snapshot’ of the system is described the instance variables, events, and the necessary system variables. The elements of abstract syntax of LSC are given below, where ‘m’ represents a LSC.

inst(m): set of all instances

dom(m, i): finite set of ‘abstract’ discrete locations of instance i, of m.

dom(m, i) ⊆ {0, …., l_max(m,i)}

dom(m): {< i, l > | i ∈ *inst(m)* ∧ *l* ∈ *dom(m, i )*}.

Messages = Message_Ids x {synch, asynch} x {!, ?}

Conditions = Condition_Ids x bexp(vis_var(m))

Labels = $\wp$(Messages) x Conditions

label(m) : dom(m) → Labels

temp(m) : (*(dom(m)* ∪ *Message Ids* ∪ *Condition Ids) → Temp)*

where Temp = {hot, cold}

order(m) : dom(m) → {true, false}  ---- a total order

coregion, L: a maximal set L of locations of i with order(m)(<i, l>) = false.

The state space of the STS associated with the basic LSC *m* is derived from the following (meta)variables, where *i* is any of the instances referred to in the LSC:

| | |
|---|---|
| ***i.location***: | the current location of instance *i* |
| ***i.events*** : | the events currently emitted by *i (*from *events(i )* ∪ *silent* |
| ***i.v*** : | the current local value of i's instance variable *v*; |
| ***status***: | {*active*, *aborted*, *terminated*} |
| ***promises***: | takes its value in the power-set of *dom(m)* ∪ {*m id*? \| *m id* ∈ *vis events(m)*}; |

**initialization predicate**: init(m) is the conjunction of following:

i.location=0; status=active; i.events ∈ vis_events(m)

**transition relation**: partitioned into following types of moves

active state: t-steps (local computations); i-steps (proceed instances)

terminated state: chaos-step (arbitrary changes)

aborted state: stutter steps (no changes to variables)

The semantics is a pure interleaving one: only a single instance is allowed to proceed at a time, and hence the transition predicate for the global transition relation is just the disjunction of the transition predicates of its partitions. Thus each action of the partions of the transition relation can be described in a self-explanatory imperative style.

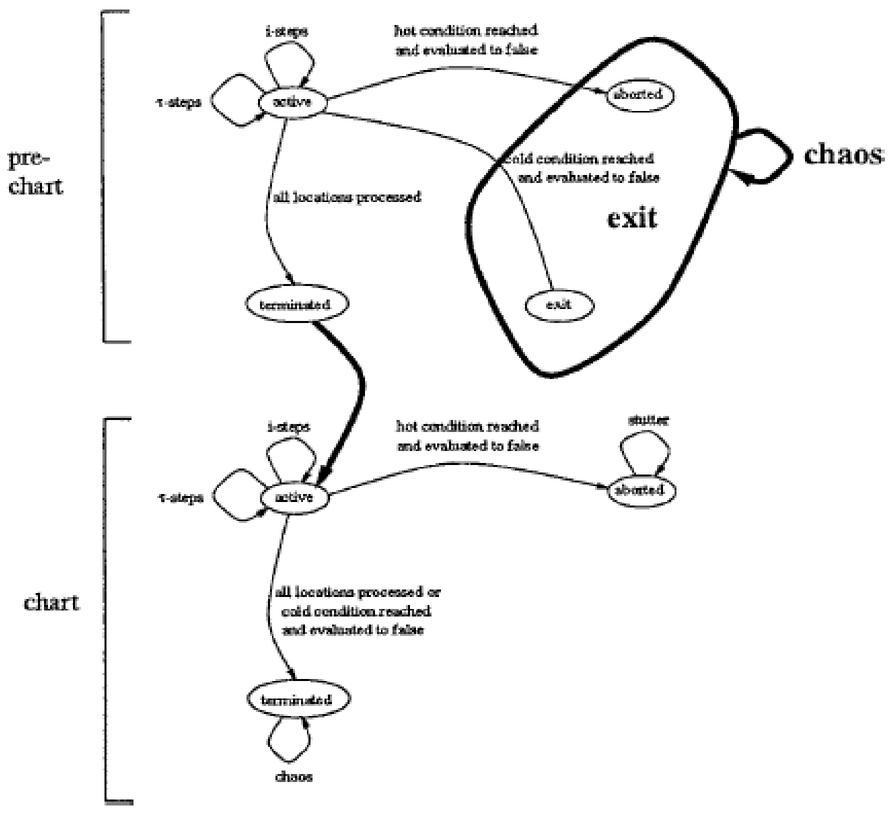
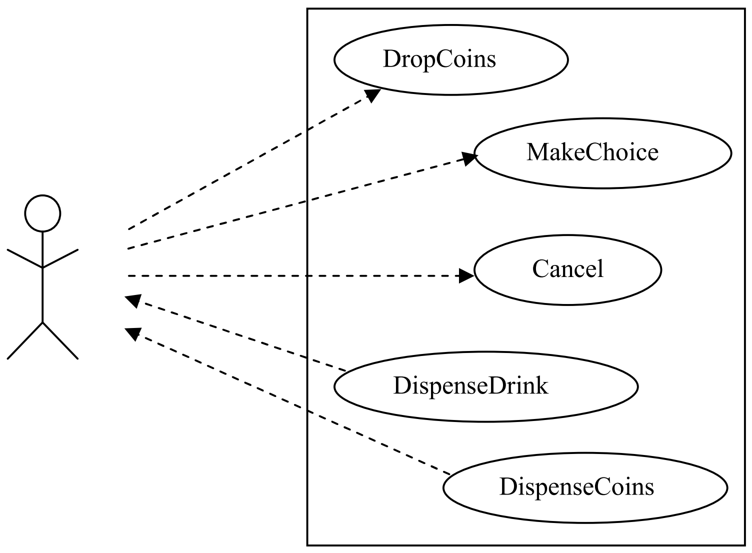


Figure. B.1. The Transition System of a LSC with Pre-chart
[Source: Damm 1999]

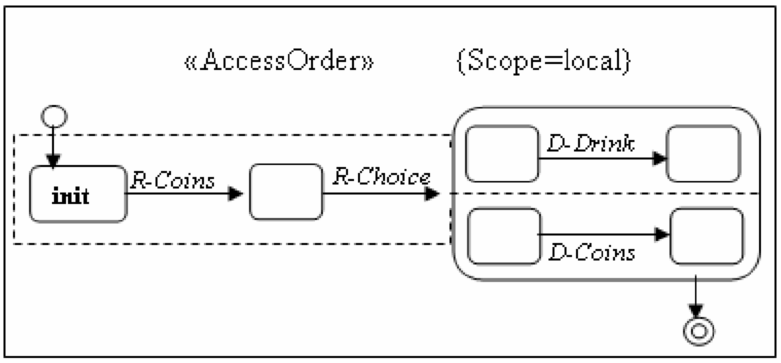
# APPENDIX C

# CMUML SPECIFICATION OF VENDING MACHINE

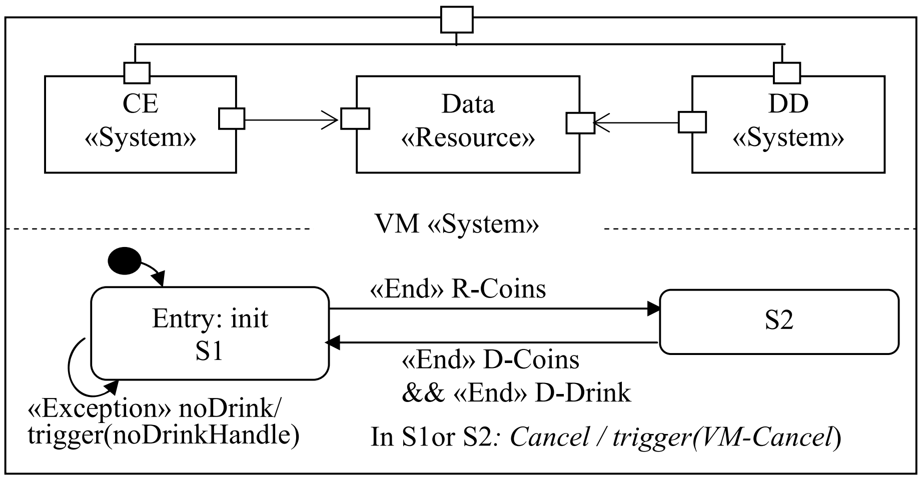
## 1. General Use Cases of a Vending Machine



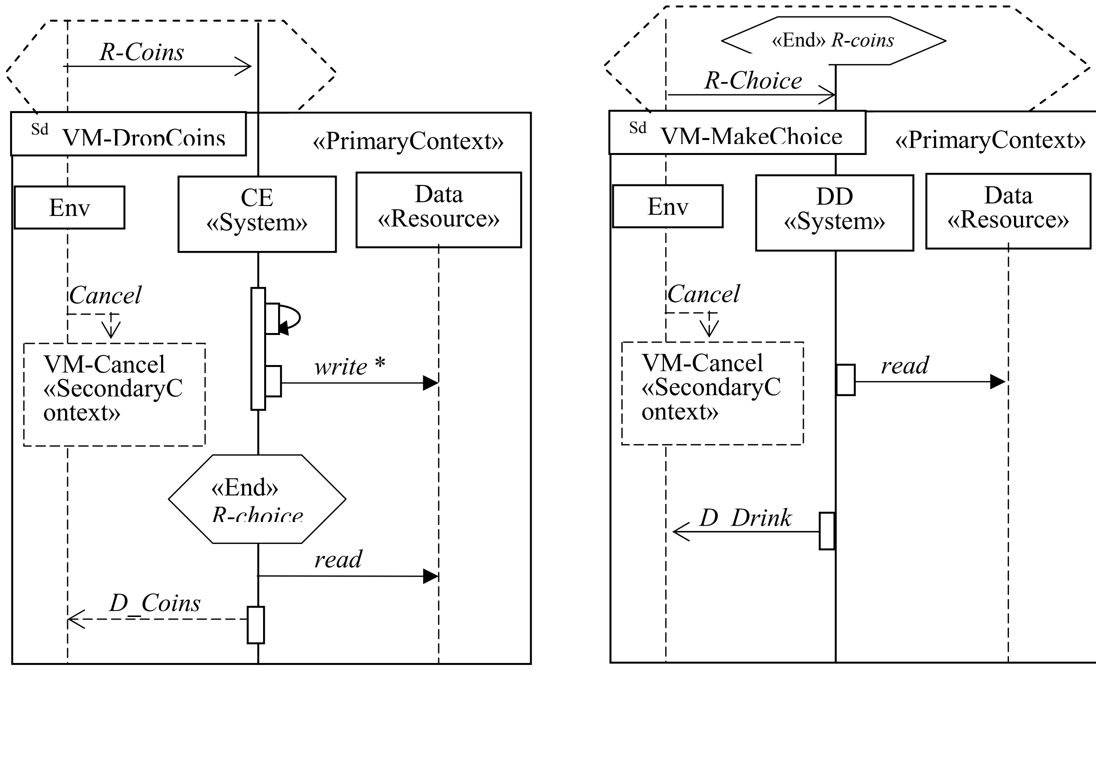
## 2. *AccessOrder* Specification of «system» VM



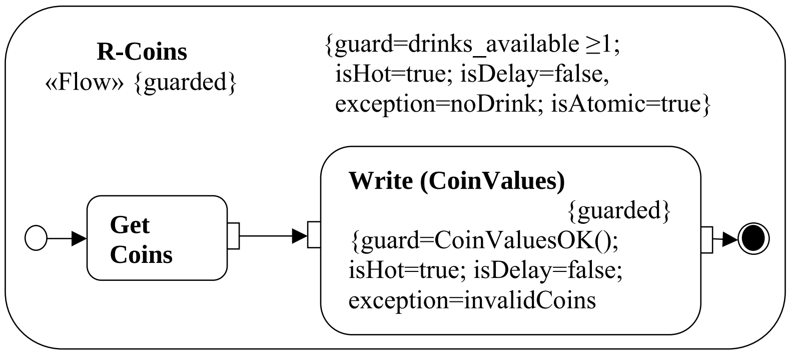
## 3. Specification of internal structure and State of «system» VM



## 4. «ScenarioContext» Specifications of Vending Machine Use Cases



## 5. *Service* Specification of R-Coins with *guarded* Semantics


R-Coins
«Flow» {guarded}
{guard=drinks_available ≥1;
isHot=true; isDelay=false,
exception=noDrink; isAtomic=true}
Get Coins
Write (CoinValues)
{guarded}
{guard=CoinValuesOK();
isHot=true; isDelay=false;
exception=invalidCoins

# BIOGRAPHY OF THE CANDIDATE

**NAME:** JAGADISH SURYADEVARA

Jagadish Suryadevara received B.Sc from Andhra University (1991), M.Sc (Applied Mathematics) from Andhra University (1994), and M.Tech (Computer Science) from Jawaharlal Nehru Technological University, Hyderabad (2000). From 1995 to 2002 he has was a lecturer in mathematics and computer science at Vignan Educational Institutions, Andhra Prasad. From 2002 to 2008 he was a lecturer in computer science in Birla Institute of Technology and Science (BITS), Pilani, Rajasthan. Currently he is a member of Real-time Modeling and Analysis Group, Mälardalen Real Time Research Centre, Västerås, Sweden.

Jagadish Suryadevara started his ph.d study in 2003 under the supervision of Prof. R.K.Shyamasundar, Sr. Professor in Tata Institute of Fundamental Research (TIFR), Mumbai. He presented a paper in 18th Int. Conf. on Parallel and Distributed Computing and Systems (PDCS), Dallas, USA. He also visited University of Texas, Dallas towards research collaboration. He participated in a work shop on semantic web held in South Korea in November, 2007. He also participated in Onassis Foundation lecture series for embedded systems at Creta, Greece in 2008. His research interests include concurrent, reactive systems, formal specifications and verifications, and software engineering. He can be reached at suryadevara.jagadish@gmail.com

## BIOGRAPHY OF THE SUPERVISOR

**NAME:** PROF. (DR). R.K. SHYAMASUNDAR

R. K. Shyamasundar received the B.E. degree from Mysore University, Mysore, India, in 1970 and the M.E. and Ph.D. degrees from Indian Institute of Science, Bangalore, India, in 1972 and 1975, respectively. Since then he has been with Tata Institute of Fundamental Research where he is currently a Senior Professor.

Prof. Shyamasundar has held various Visiting Scientist/Faculty positions at various places such as Technological University of Eindhoven Netherlands, State University of Utrecht. Netherlands, IBM T. J. Watson Research Center Yorktown Heights. NY, and Pennsylvania State University, University Park. He is a well known researcher in India and abroad. His main research interests are semantics of programming languages. real-time programming, and logic programming. Prof. Shyamasundar is a member of the Association for Computing Machinery, EATCS and the IEEE Computer Society. He is also a member of various scientific, and research committees of Government of India. He can be reached at shyam@tcs.tifr.res.in.