Generation of SDL Specifications from UML and MSC Use Cases

Stephan Bourduas

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements for the Degree of
Master of Applied Science

Concordia University

Montreal, Quebec, Canada

June 2001

0-612-64058-2

Abstract

Generation of SDL Specifications from UML and MSC Use Cases

Stephan Bourduas

As software systems are becoming more complex, errors in the implementation of user requirements have become common. Methodologies that incorporate formal and semi-formal techniques into the development process thus minimizing errors would be beneficial to the software engineering community as a whole. The inclusion of graphical notations such as UML (Unified Modeling Language), MSC (Message Sequence Charts) and SDL (Specification Description Language) in the development process will allow for a much higher level of automation resulting in fewer errors and ultimately better quality software.

In this thesis, we developed and implemented an approach for specifying distributed systems in UML and MSC, where MSC *use cases* are used to express the requirements and UML is used to describe the architecture of the distributed system under consideration. To specify the behavior, the MSC language is used to describe high-level use cases that are then refined in a stepwise manner into *design* MSCs. For each refinement, the newly obtained MSC is validated against the previous MSC using a conformance relation that we defined. UML, more precisely Class Diagrams and Object Diagrams, are used to specify the architecture of the system. A distributed system specification style in UML was developed and mapping rules were defined to enable the translation from a UML to SDL architecture. The approach then makes use of an existing methodology that combines the design MSCs and the SDL architecture into a full SDL specification, which is correct with respect to the design MSCs and is free of deadlocks.

Our approach takes advantage of the formality of SDL while using the two very popular and intuitive notations, UML and MSC, as a front end. The formal semantics of SDL allow for simulation and validation. In addition, commercial tools support automatic code generation.

Dedication

To my parents, Maria and René.

Acknowledgements

I would like to express my heartfelt thanks to my supervisor, Dr. Ferhat Khendek, for his patience, guidance and dedication. He motivates students to learn and his door is always open. Dr. Khendek is a credit to the teaching profession, and I could not have asked for a better supervisor.

Words are not sufficient to express my gratitude for the love and support of my parents, Maria and René, without which I would have never made it this far. Thank you for everything.

My cousins, Maria, Venetia and Kosta, deserve thanks for all their support and understanding. I know that whatever lies ahead, I can always count on them to be there, I am forever grateful.

My aunts and uncles, for all your support and love, I cannot thank you enough. I would like to thank in particular Leah and Nick Priftis for all the time they spent helping me and for their friendship.

My dearest Maria, for always being there for me, for all your support, for just listening, for just being you, I thank you endlessly.

I would also like to take this opportunity to thank France Telecom for their financial support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Software Lifecycle

Traditional engineering disciplines deal with the invariant constraints of the real world. For example, electrical engineers can use mathematics to verify the correctness of their designs. Software engineering is more abstract and informal and currently lacks predictive models and techniques [1]. The design of software is an intellectual endeavor that is not limited by physical laws [1], and hence is more difficult to verify for correctness. As such, a problem can be solved in many ways, but not all of the solutions will be acceptable.

Early software systems were simple due to the fact that computers were relatively primitive compared to today's technology. They were text-based and running single-threaded operating systems that were mainly used for batch processing and number crunching. As the complexity of software systems increased, errors in the implementation of user requirements became more frequent. A major reason is that the techniques used to develop the small early systems did not scale up for large systems [2]. In addition, many errors can be attributed to the fact that the methods employed by software practitioners are mostly informal and require a great amount of intuition and experience in order to create quality software systems. Hence, methodologies that add more structure and minimize errors would be beneficial to the software engineering community as a whole.

The objective of software engineering is to produce *quality* software while incurring *reasonable* costs within a *predictable* amount of time [2]. The lifetime of a software system can be divided into stages, which if managed properly, would result in high quality software. A methodology that defines how the development process progresses from stage to stage is referred to as a software lifecycle model. One of the most widely used models is the Waterfall model as shown in Figure 1.1.

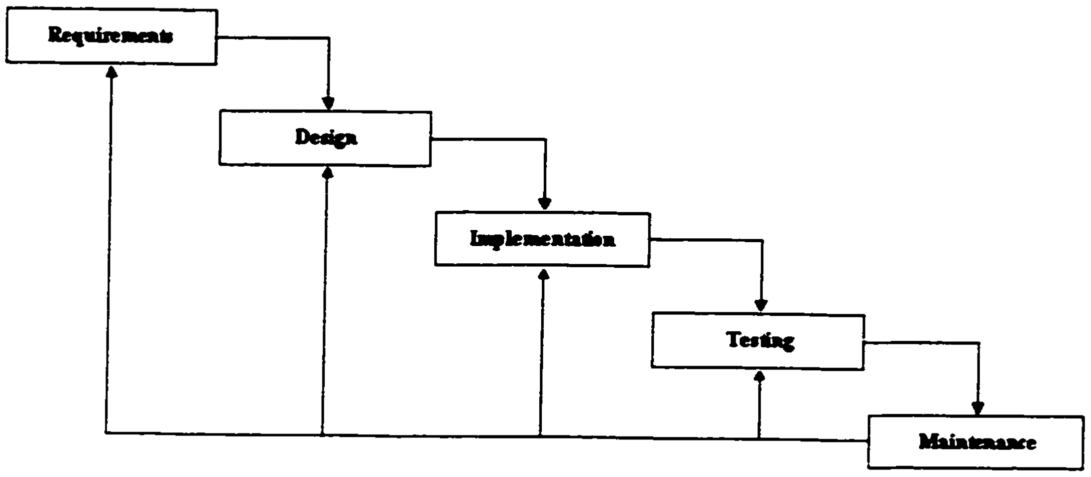


Figure 1.1: Waterfall Lifecycle Model.

The waterfall model is document driven, where the output of a stage is the input to the next. If an error is detected at any stage, a feedback mechanism allows for the corrections to be made at the appropriate stage, after which, the changes must be propagated throughout the rest of the stages. The different stages of any software lifecycle are:

i. *Requirements:* The client defines the desired features offered by the system at this stage.

ii. *Design:* The system is partitioned into subsystems that will implement the desired functionality. The design documents serve as a blueprint for the implementation team. It is also possible to have multiple design documents that show different levels of abstraction (high-level view or a low-level view for instance).

iii. *Implementation:* The design documents are realized into a working system by the implementation team.

iv. *Testing:* The implementation is tested with regards to the user requirements. If an inconsistency is detected at this stage, the feedback mechanism will be used to correct the error.

v. *Maintenance:* Once the system has been completed and delivered to the client, any changes such as adding features or correcting errors fall under the maintenance category.

1.2 Formal and Semi-Formal Notations

The methodologies and notations that are used for designing software systems can be classified into two groups:

i. *Formal:* A formal language has a formal syntax and semantics that is based on mathematics. Algorithms can therefore be applied to a formal specification in order to simulate, verify and validate a system. The semantics of a formal language is fully defined and thus there is no ambiguity in a formal specification.

ii. *Semi-Formal:* A semi-formal language may have a standardized syntax, but the semantics is not completely defined (or not formally defined). They are often based on natural languages and as a result, mathematical algorithms cannot be applied to them.

Both formal and semi-formal languages have their strengths and weaknesses. An advantage of semi-formal languages is that they offer great flexibility when specifying systems and are generally intuitive and relatively easy to learn. Formal languages, even though they are very rigidly defined, allow for the application of mathematical modeling that increases the confidence in the finished specification, but they are more difficult to use and learn, and are not as intuitive as semi-formal notations.

MSC (Message Sequence Charts) and SDL (Specification and Description Language) [3, 4] are widely used for telecommunication software engineering, and they fall into the category of formal languages. SDL can be used to create high or low-level specifications that can be simulated and validated. In addition, commercial tools such as ObjectGeode [5] and SDT Tau [6], can automatically generate source code from a low level SDL specification. The MSC language can be used to express the behavior of a system at varying granularities (generally used at the requirements stage) and is very intuitive.

The Unified Modeling Language (UML) is a semi-formal language that has gained in popularity in recent years in the object-oriented (OO) community. It is a very expressive notation where a software practioner with only a basic knowledge of the language can design complex systems.

1.3 Using MSC and UML to Generate SDL Design specifications

The purpose of this thesis is to develop an approach for designing software that encompasses all of the stages of the software lifecycle, and that lends itself to automation, allowing for conformance relationships and mapping rules that can be rigidly enforced so that the introduction of human errors can be minimized.

A use case describes some functionality offered by a system as perceived by the user or an external actor of the system [7, 8, 9]. The user sees the system as a black box that responds to an input with a specified output. Use cases are not formally defined and are very often specified using a combination of text and diagrams that must be interpreted by the system designers and translated into a more concrete representation. It would be beneficial to represent use cases in an unambiguous way from the start, thereby reducing the probability of misunderstanding, and enabling the use of CASE tools. The MSC language [10, 11, 12] is an excellent candidate, as discussed in [13]. A use-case model can be developed through interactions with the system designers and the customers as described in [7, 8, 18] and expressed formally using MSC. The process starts with the user requirements being described as MSC Use Cases that represent an abstract view of the system. The first methodology (shown in Figure 1.2) deals with the incremental refinement of MSC use cases into design MSCs. The MSCs are then refined in a stepwise manner, guided by the system architecture, until a detailed design is obtained. To guarantee that the resulting design MSCs are correct, a conformance relation between two MSCs was defined. The conformance relation checks if a modified MSC preserves the event orderings of the original. The conformance relation is used after every refinement thereby allowing a user to have confidence in the correctness of each refinement, and ultimately in the final design specification.

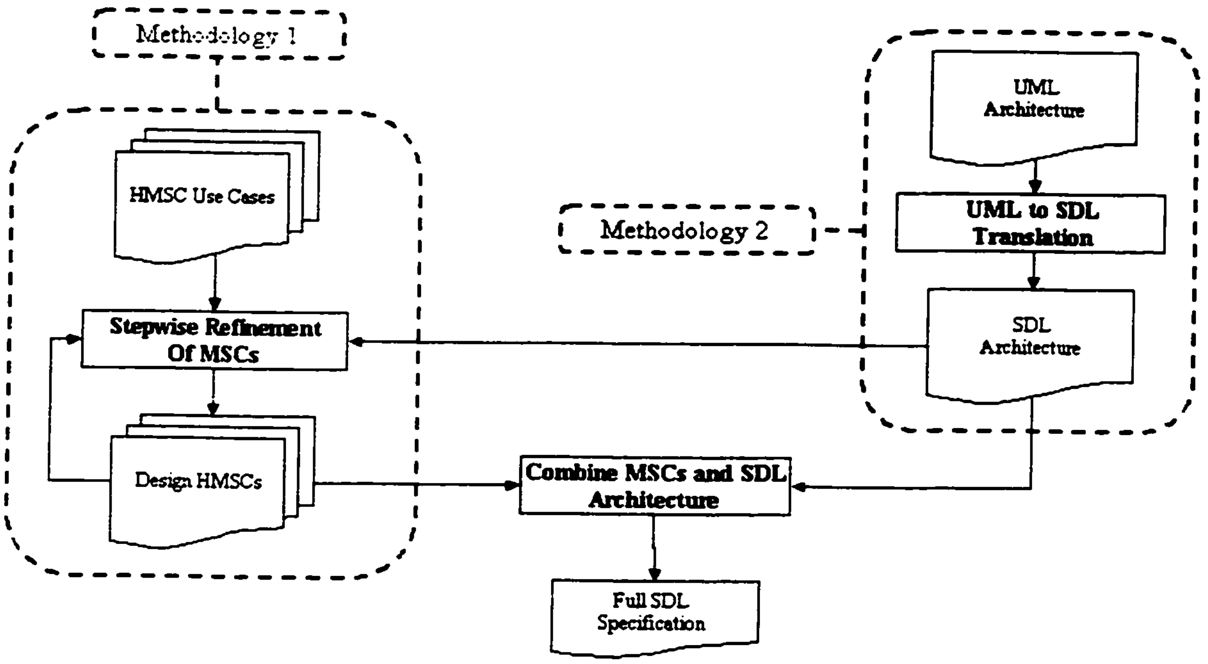


Figure 1.2: The overall process.

The popularity of UML, and most importantly its expressiveness, makes it an ideal notation to use at the early stages of development. On the other hand, its lack of formality becomes a liability at the later stages of the development process. SDL, with its formal semantics, is a perfectly suited language for specifying detailed designs. By analyzing the strengths and weaknesses of both languages, it becomes apparent that the two can be used together at different stages of the software lifecycle to create an approach that is superior to using them separately. For the purposes of this thesis, the architecture of a system is defined to be the specification of the components of the system and their interconnections minus any behavioral specification. The second methodology depicted in Figure 1.2 uses UML to describe a distributed system architecture that can be translated into SDL. The architecture of the target system is described using a stylized form of UML that is then translated into SDL via mapping rules.

The work presented in this thesis is intended to leverage an existing methodology by building on top of it. The methodology in question generates a full (behavior and architecture) SDL design specification from behavioral requirements given in design MSCs and the architecture of the target system given in SDL [14, 15]. Typically, design

MSCs are created in an ad-hoc, stepwise manner where a designer adds more and more detail until they are satisfied with the result. This sort of approach is error prone because there is no guarantee that the detailed design conforms to the requirements. To remedy this problem, the aforementioned different, but complimentary methodologies were developed. The design MSCs obtained from the first methodology are combined with the SDL architecture obtained from the second to automatically generate a full SDL specification (architecture plus behavior) using the MSC-to-SDL tool [14, 15,], as shown in the bottom portion of Figure 1.2.

The automation of formal methodologies like the ones we have developed encourage the widespread use of these techniques to develop robust software systems by experts and novices alike, thus significantly reducing the costs normally associated with complex systems.

1.4 Organization of Thesis

The thesis is organized in the following way:

- *Chapter 2:* Introduces the reader to the UML, SDL and MSC notations. In addition, the MSC to SDL methodology that generates a full SDL specifications from MSCs for a given target architecture specified in SDL is briefly described.

- *Chapter 3:* Describes our MSC use-case refinement approach. The conformance relation that we developed is first explained. Next, our stepwise refinement approach is described for bMSC, and then expanded to include HMSC. The approach is then illustrated with an example.

- *Chapter 4:* Describes our stylized use of UML for describing distributed system architectures. The UML to SDL architectural mapping rules that we defined are then presented and the approach is illustrated via two examples.

- *Chapter 5:* Discusses the implementations of the two methodologies, and the limitations of the tools. An example that illustrates how the two methodologies from chapters 3 and 4 are combined is then presented.

- *Chapter 6:* Outlines the contributions of this thesis and presents some ideas for expanding upon them.

Chapter 2

Background

This chapter introduces concepts that are needed to understand the material presented in Chapters 3, 4 and 5. The first three sections present UML, MSC and SDL respectively. The last section introduces the MSC to SDL methodology that is used to generate full SDL specifications.

2.1 Unified Modeling Language

The Unified Modeling Language, or UML, evolved from the combined work of Grady Booch, James Rumbaugh and Ivar Jacobson. They each had their own notations from which they took the strengths of each and combined them into UML.

The UML has become the *defacto* standard for the documentation and design of software systems. The software engineering community has embraced UML with enthusiasm because it enables designers to communicate their ideas in a simple yet powerful graphical notation. As UML is an object-oriented modeling language, using UML extensively in the design phase will in general result in well designed software that saves money by reducing implementation costs, testing and maintenance (fixing problems after delivery). An added advantage of using UML extensively during the design phase is that most of the documentation will be finished before the system is even implemented. It is well known that for various reasons, most organizations do not document their software properly. Having good documentation is crucial when bugs need to be fixed, or when features are added or modified. Without documentation, it is very difficult for a programmer who is unfamiliar with the system to identify and change the correct parts of a program without introducing more bugs.

The following sections will introduce the aspects of UML that are needed to understand the material presented in this thesis. Readers interested in learning more about UML can visit [16] or [17].

2.1.1 Static Modeling

A UML class diagram shows the static relationships between the classes that make up a system [7, 8]. The class diagram is used as a blueprint for later on in the development process, where the static relationships between classes must be respected when the dynamic aspects of the system are modeled.

2.1.1.1 Class

Figure 2.1 depicts the graphical UML notation for a class. The class shown in Figure 2.1(a) is comprised of three compartments:

i. *Name:* The name of the class, which should make sense with respect to the system being designed and be as unambiguous as possible.

ii. *Attribute:* Class attributes are values that describe the characteristics of a class/class instance. Attributes can have different visibilities (i.e. public, private, protected) and be of different types.

iii. *Operation:* The operations of a class can be used to manipulate the attributes (variables) or state of a object. Operations are defined by visibility, return type and parameter list, which together make up the *signature of the operation* [7].

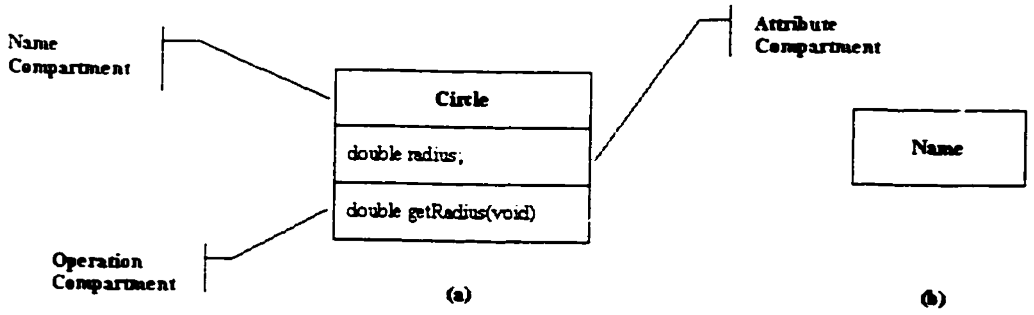


Figure 2.1: Representing a class in UML.

The graphical representation of a class is not restricted to the form in Figure 2.1(a), which shows all of three compartments. Figure 2.1(b) shows a shorthand version of a class with only the name compartment shown. The name compartment of a class must always be shown, but the other compartments are optional. In addition, UML allows for the creation of custom compartments by a user, although it is not usually needed or recommended for compatibility reasons.

2.1.1.2 Extending UML

The elements provided by UML are generic in nature and can be used to define systems in any domain. It is sometimes desirable to extend or modify existing elements for custom applications such as creating domain specific elements. UML itself provides the means of customizing model elements. This feature makes UML very powerful because customizations can be made while still using the standard UML notations.

In order to understand some of the topics discussed in Chapter 4, it is necessary to understand two extension mechanisms provided by UML, which are explained in the next two sections.

2.1.1.2.1 Stereotypes

The stereotype [7] extension mechanism can be applied to any UML model element for the purpose of defining a new element that is based on an existing one. UML has predefined stereotypes that can be used in certain situations.

Stereotypes are often used to add meaning to an additional model element [8]. When a stereotype is created, the characteristics belonging to the new stereotype are also defined. Henceforth, whenever the stereotype is applied to a model element, it will be understood that the model element has the characteristics of the stereotype. Figure 2.2 shows an example of a class with the UML Actor stereotype applied to it.

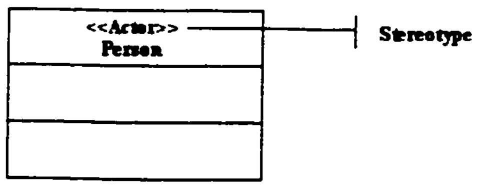


Figure 2.2: UML stereotypes.

The name of the stereotype is placed in brackets (guillemets) in the name compartment (above the name) [8]. The Actor stereotype is a pre-defined stereotype that indicates that the Person class is an external agent that interacts with the system.

2.1.1.2.2 Tagged Values

Tagged values can be used to attach any kind of additional information to elements of a model [7, 8]. As with stereotypes, UML has some predefined tagged values. One of the more commonly used tagged values is the abstract value as shown in Figure 2.3. A tagged value is always shown in braces.

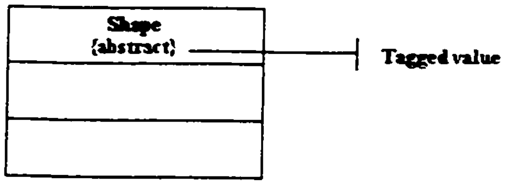


Figure 2.3: UML tagged values.

2.1.1.3 Relationships Between Classes

2.1.1.3.1 Association

A system will usually consist of more than one class. The classes that comprise a system will have relationships with each other that need to be expressed in UML. The most common and simple type of relationship is called an *association* [7]. A line connecting two classes together denotes an association in UML. An association is given a name to make the relationship between two classes more clear (the name is often a verb). A bi-directional association, where each participant is aware of the other (programmatically speaking), is most commonly used. Since association names are commonly verbs, using an arrowhead after the name can show the direction of the action. A bi-directional association can have two names (optional) as shown in Figure 2.4(a).

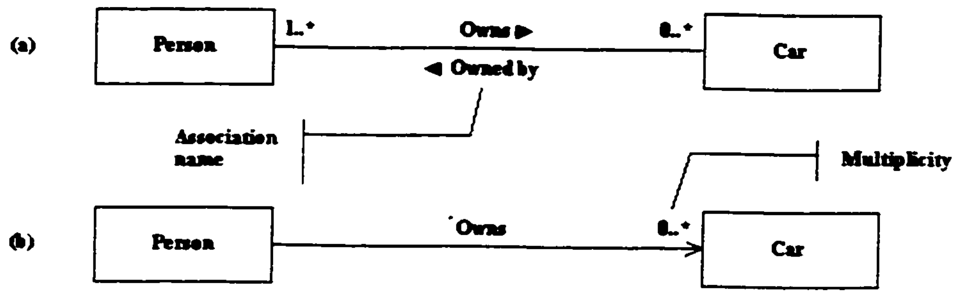


Figure 2.4: Representing associations in UML.

An association can be shown to be unidirectional (or navigable) by adding an arrowhead at one end of the association [8], which means that the association can be used only in the direction of the arrow, as shown in Figure 2.4(b).

The *multiplicity* of each class is shown at their respective association ends as shown in Figure 2.4 (by default, the multiplicity is one). The multiplicity defines how many instances of each class involved in the association will be present when the system is running. The example of Figure 2.4(b) states that a person (1 by default) can own zero or more (0..*) cars.

2.1.1.3.2 Aggregation and Composition

Aggregation and composition are two specialized types of association. Aggregation means that the relationship between the two classes is some sort of part-whole relationship and can be described as "consists-of" or "is part of" [7]. Aggregation is shown by using a hollow diamond at one end of the association as shown in Figure 2.5(a). The composition relationship is stronger where the parts live only as long as the whole does. Composition is shown using a blackened diamond as seen in Figure 2.5(b).

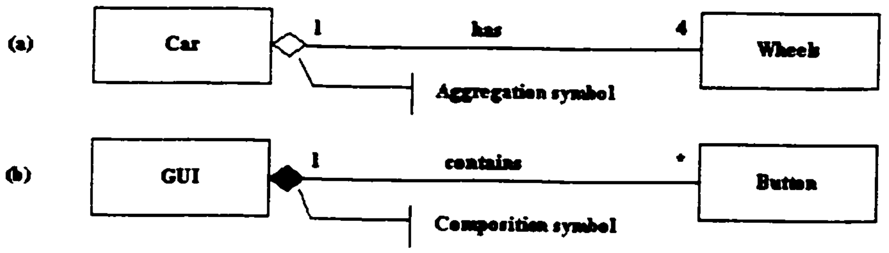


Figure 2.5: Aggregation and composition in UML.

The difference between aggregation and composition is subtle and it is usually not incorrect to use the aggregation exclusively since composition denotes a stronger form of aggregation [7, 8]. To illustrate the difference, the wheels of a car can be removed and used on another car whereas the buttons contained in a graphical user interface (GUI) will die when the GUI is closed. The relationship between the buttons and the GUI is stronger than that of the wheels and the car.

In order to enable the use of the concept of inheritance from object-oriented programming, UML provide designers with the generalization relationship [7] shown in Figure 2.6. Generalization is used to show a relationship between a general superclass and a more specific subclass. The superclass is often times an abstract[1] class as shown by the Shape class in Figure 2.6. The shape class can have an operation called "getArea" where the area of the shape is returned. The subclasses of the shape class must each implement this operation to return their respective areas. For instance, the area of a square is computed differently than the area of a circle.

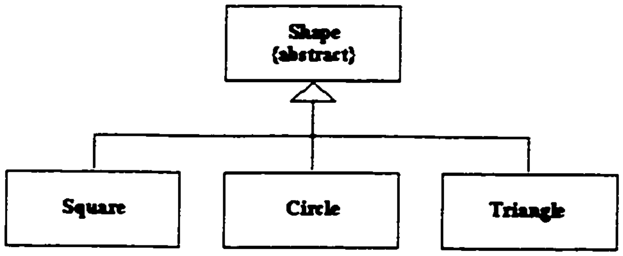


Figure 2.6: Generalization in UML.

2.1.1.4 Type and Implementation Classes

There are two standard stereotypes that are of interest. These are the *type* and *Implementation Class* stereotypes.

[1] An abstract class is not instantiable. It is used to group common characteristics together that can be refined by subclasses.

A *type* stereotype applied to a class is used to specify operations applicable to a set of objects without defining the physical implementation of that class [18]. These types of classes cannot be instantiated (much like an abstract class).

An *Implementation Class* may realize a number of different Types. An Implementation Class realizes a Type if it provides all of the operations defined for that Type [7]. An object may have at most one Implementation Class. However, an object may conform to multiple different Types [7, 18]. All of the standard UML associations can be used to show relationships between Type classes. Implementation and Type classes, on the other hand, can only be related by the realization relationship [7, 18]. The implementation of a type is shown by the *realization* relationship between the type and implementation classes (a dashed line with a solid triangular arrowhead). This is demonstrated in Figure 2.7.

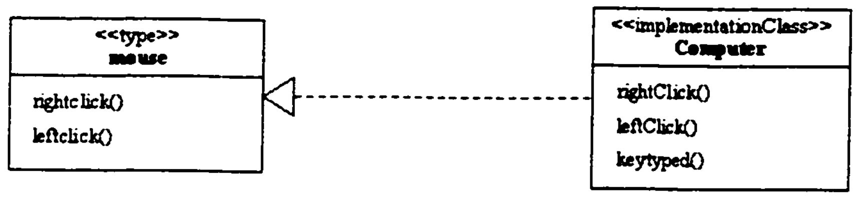


Figure 2.7: Realization relationship in UML.

2.1.1.5 Interface

Another type of class that is commonly used in UML is called an Interface. An interface consists of operation signatures that have no implementation (abstract operations) [8]. Figure 2.8 shows and example of an interface class called "Shape".

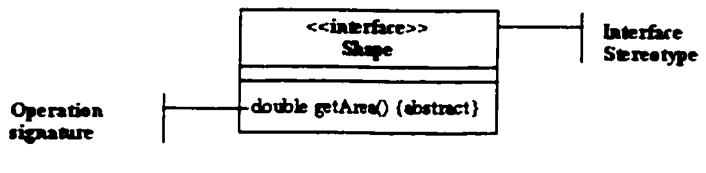
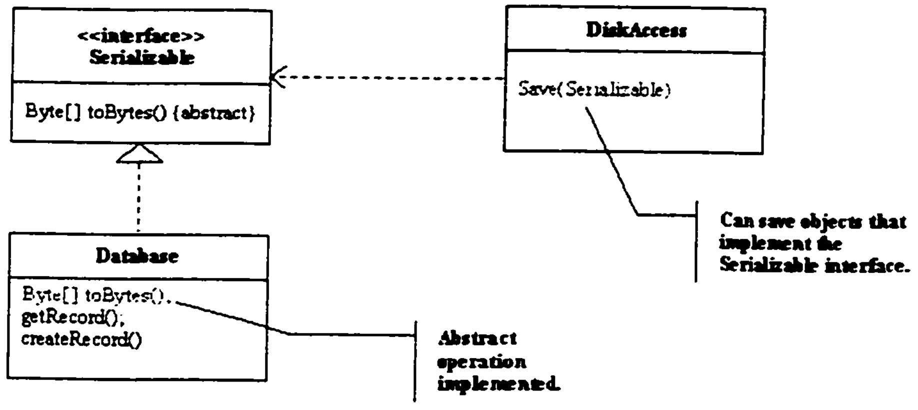


Figure 2.8: Interfaces in UML.

Interfaces are used to create uniformity in systems. Different classes that use the same interface will be able to "understand" each other. Classes can be dependent on, or realize, an interface. Figure 2.9 shows how two classes relate to the Serializable[1] interface. The DiskAccess class has a save operation that takes objects of type Serializable. As long as a class implements the Serializable interface, it can be passed to the DiskAccess class for saving to disk.

Interfaces are very useful because a class can implement multiple interfaces, which will allow a single class to be compatible with many other classes that may understand only one interface.



<<interface>>
Serializable
Byte[] toBytes() {abstract}
DiskAccess
Save(Serializable)
Can save objects that implement the Serializable interface.
Database
Byte[] toBytes(),
getRecord();
createRecord()
Abstract operation implemented.

Figure 2.9: Using an interface in UML.

2.1.1.6 Package

A package in UML is used to group related classes together to make a model more manageable. Packages can have relationships between each other and use the extension mechanisms just like classes [8]. A package can contain classes or other packages. Figure 2.10 shows a package called Application that is dependent on the API package.

[1] The Serializable interface in Java allows an object in memory to be converted into a byte array that can be stored on disk or sent over the Internet, the original object can then be reconstructed using the byte array. This allows the state of an object to be preserved.

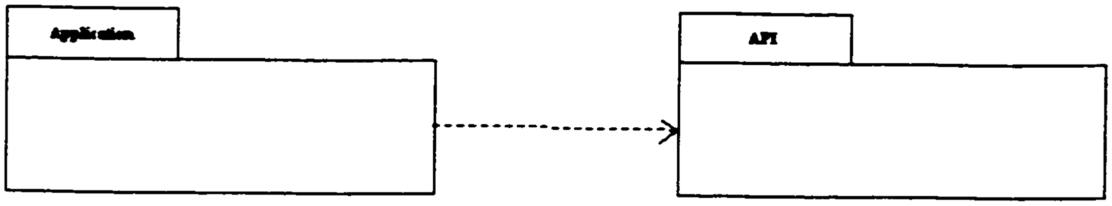


Figure 2.10: Packages in UML.

2.1.2 Dynamic Modeling

Dynamic modeling shows how the system will behave during the execution of a system using State, Sequence, Collaboration/Object and Activity diagrams [7]. For the purposes of this thesis, only Object diagrams are considered.

An object diagram shows how instances of classes (objects) are related during runtime. An object is represented as a box containing the name of the object and the name of the class separated by a colon. The two names are underlined to denote an object instance. Figure 2.11 shows a simple object diagram. The lines that connect the objects are called *links* and are instances associations that are defined in class diagrams.

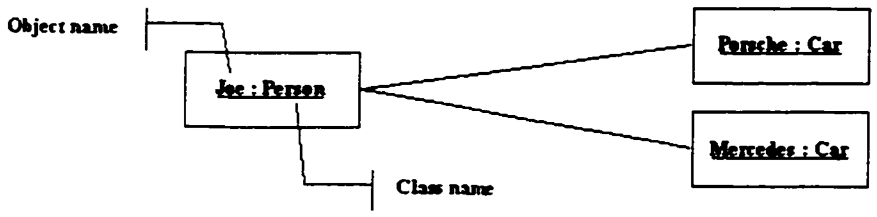


Figure 2.11: UML Object Diagram.

2.2 Message Sequence Charts

Message Sequence Charts (MSC) is a specification language that is standardized by the International Telecommunications Union (ITU) [10]. It is a formally defined language and has both graphical and textual representations, but people work mostly with the graphical notation. Due to its intuitive nature, the MSC language has been widely adopted across the telecommunications industry because it is especially useful in describing the behavior of distributed systems.

As the MSC language can be used at any level of abstraction, it may be used for the following purposes during the software lifecycle:

i. To describe use cases and user requirements.

ii. To design the system under consideration.

iii. To develop test cases.

iv. To capture simulation/execution traces.

Using the MSC language to describe use cases requires a high level of abstraction, whereas the design of a system will require a low level of abstraction.

Although the MSC language is very rich and allows for the description of complex behaviors, only the relevant features will be mentioned in this thesis. Basic and High-Level MSC represent two levels of abstraction that can be used together to describe complex systems. They will be introduced in the next two sections.

2.2.1 Basic Message Sequence Charts

Basic Message Sequence Charts, or bMSC [12, 18], are used to show how parallel processes communicate with each other. A bMSC consists of process instances that have vertical lines (lifelines) and arrows representing messages connecting these lines. An example of a bMSC is shown in Figure 2.12. The element name of a process represents the class name (in object-oriented terminology) and the instance name represents the instantiated object name. Another important feature of a bMSC is the fact that messages can be sent to and from the environment, which allows open system to be modeled.

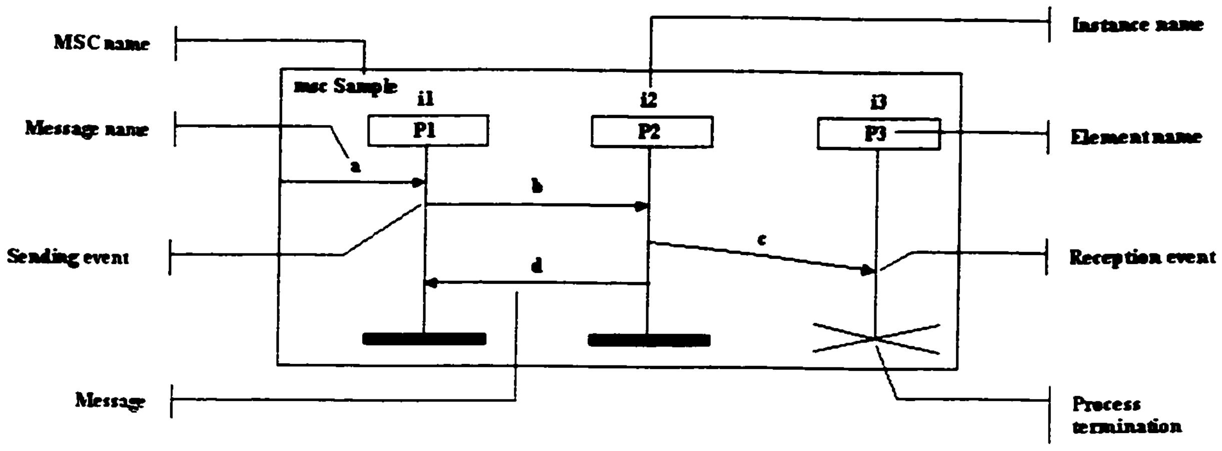


Figure 2.12: A basic MSC.

Messages in an MSC are asynchronous [18], which means that a process instance can send a message and continue its execution without waiting for some sort of reply. The events on the axis of a process are ordered in time, which increases in the downward direction. A message consists of two events, sending and reception. The sending event must always occur before the reception, unless a coregion is used. A coregion [10, 12] can be used to relax this restriction for selected parts of a process axis. This feature can be useful because the order may not have been decided on yet, but will be finalized later. A coregion is indicated by drawing a portion of the process axis as a dotted line, as shown in Figure 2.13.

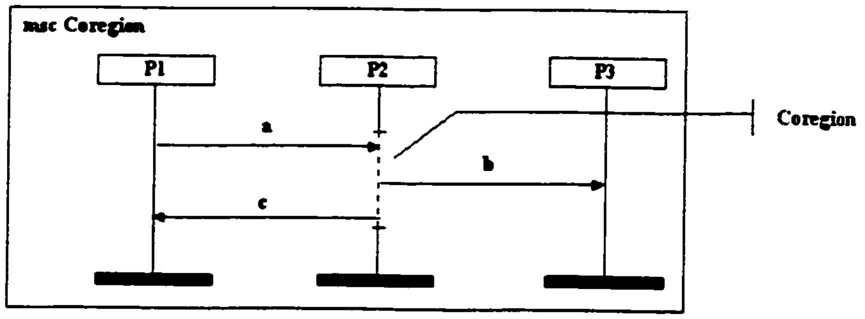


Figure 2.13: A coregion example.

A bMSC does not have to consist of only messages being passed back and forth. Designers can use conditions, actions, timers and inline expressions to add more complexity to a MSC. Figure 2.14 shows the inline expressions that are available in MSC [10, 12]:

- *Alternative:* Only one section will be executed.
- *Parallel:* The events all sections are executed in parallel.
- *Loop:* The events enclosed by the loop expression are executed multiple times.
- *Optional:* The events enclosed may or may not be executed.
- *Exception:* Represents an exceptional case in the execution.

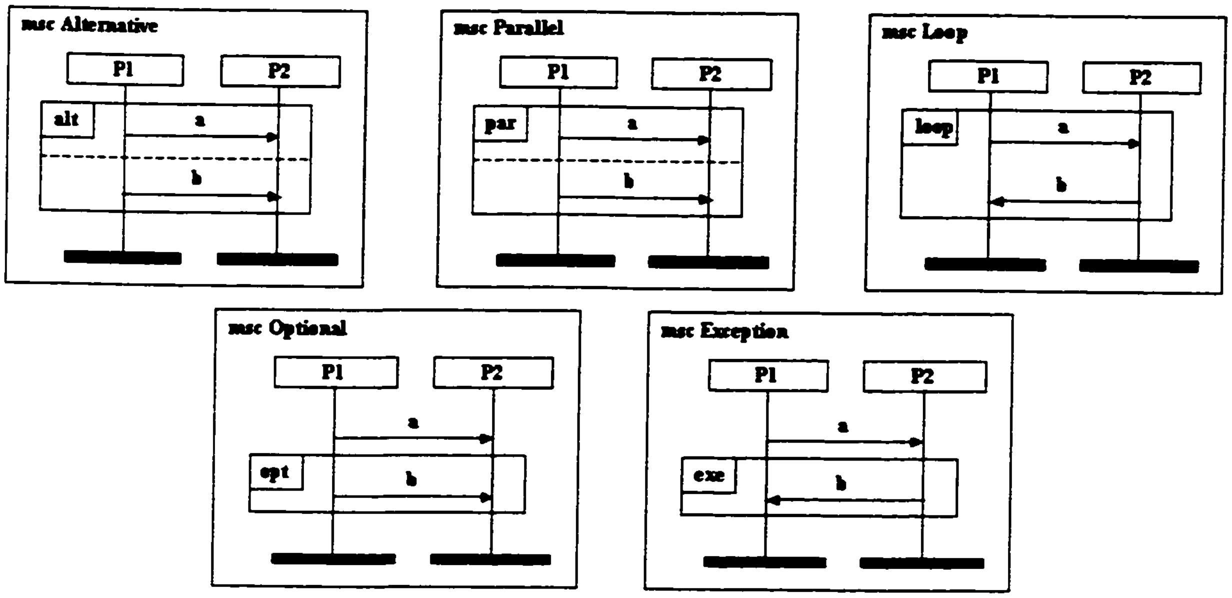


Figure 2.14: MSC inline expressions.

Conditions [10] are used to specify states within the system. Figure 2.15 shows the three types of conditions is MSC:

i. *Global:* A state that applies to all instances within an MSC.

ii. *Non-Global:* A state that applies to a subset of the instances of an MSC.

iii. *Local:* A state that applies to only one instance of an MSC.

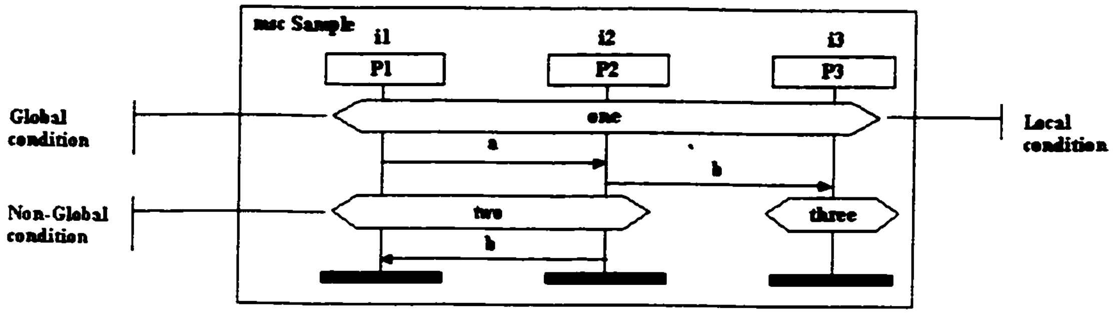


Figure 2.15: MSC conditions.

Timers [10] can be used express timing constraints. Timers can be set, reset, and can timeout. When they expire, timers generate a message that is put into the input queue of the process. Figure 2.16 the graphical notations for timer events in MSC.

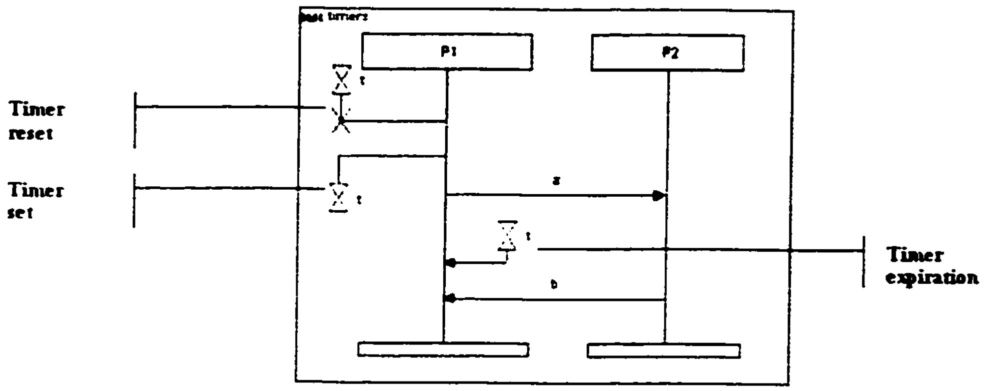


Figure 2.16: MSC timers.

2.2.2 High-Level Message Sequence Charts

Describing a complex system with only one MSC is impractical because it would be too large to be of any use. High-Level Message Sequence Charts (HMSC) allow many small bMSCs to be related to each other in an easy to understand manner [12]. An HMSC shows the system at a higher level of abstraction where the execution trace of the system can be followed in terms of bMSC, and not individual messages like in HMSCs. An example of an HMSC is shown in Figure 2.17.

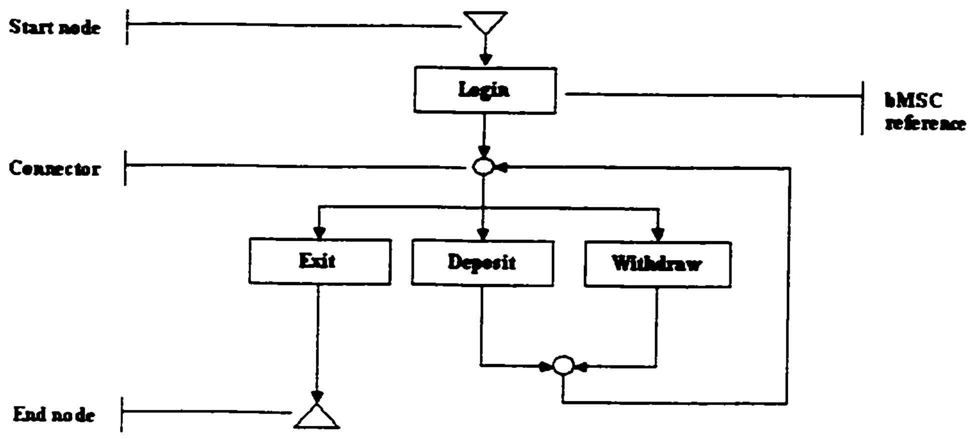


Figure 2.17: An HMSC example.

In Figure 2.17, after the Login bMSC is executed, any of the following bMSCs can be executed, which offers conditional execution of whole bMSC to be defined. After either the Deposit or Withdraw bMSCs, the flow of execution loops back to a connector where a choice of bMSCs is once again available. HMSCs are simple but powerful and enable complex systems to be designed using the MSC language.

2.3 Specification Description Language

The Specification and Description Language (SDL) is a standard defined by the International Telecommunications Union (ITU-T), and is referred to as recommendation Z.100 [3]. Like the MSC language, SDL has a graphical and textual notation. SDL can be used to simulate, validate and verify a system due to the fact that it is formally defined. This has led to its widespread use in the telecommunications industry for modeling distributed systems.

A complete SDL specification consists of two parts, the structure of the system and the behavior. Each will be discussed in the next two sections.

2.3.1 Structural SDL

An SDL system is made up of a hierarchy of components. In UML, The static and dynamic portions of a system are presented separately. In SDL, the static and dynamic portions are presented on the same diagram. This may seem confusing at first, but SDL is quite intuitive and easy to learn. SDL is a large standard, and so only a subset will be presented in this section.

There are two possible ways to describe the architecture of a system in SDL. The first is local specification (to specify elements directly), which is the original (old) style. The second is to use remote specifications using SDL *types*. The old style has three major components that are used to design a system [3, 4]:

i. *Process:* A process is the basic behavioral unit in SDL and has it's own thread of execution. A distributed system is made up of multiple processes executing in parallel. The behavior of a process is defined using extend finite state machines and will be discussed in the next section.

ii. *Block:* A block can contain either other blocks or processes. This allows a system to be specified in a hierarchical manner.

iii. *Channel:* Channels connect blocks and processes together and allow for the exchange of asynchronous messages (or signals) back and forth.

The combination of blocks, processes and channels allows the system architecture to be defined. Figure 2.18 shows an example of a system specified using the old style of SDL (using Blocks and Processes exclusively). A disadvantage of the old style is that it is not object-oriented and so reuse is not easily accomplished.

Figure 2.18 also shows examples of two other important SDL features, the text symbol and signal list. The text symbol is used for variable declarations and the signal list is used to specify which signals are valid for a particular direction of a channel. A bi-directional channel has two signal lists, one for each direction.

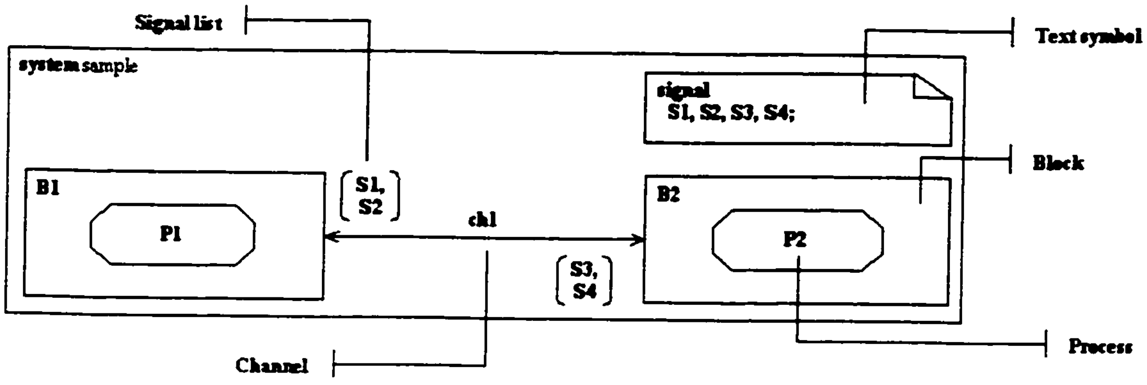


Figure 2.18: Old style SDL specification.

Using the old style of SDL can be compared to writing a program using a sequential programming style. Of course, these days it is desirable to use object-oriented languages for programming because such systems tend to be of better quality. The equivalent in SDL is to use types [3, 4], which allows referenced systems to be modeled. A type is similar to a Class in UML. A type can be defined once, and instantiated many times. Figure 2.19 shows a system similar to Figure 2.18, but using a Block Type instead of blocks and processes.

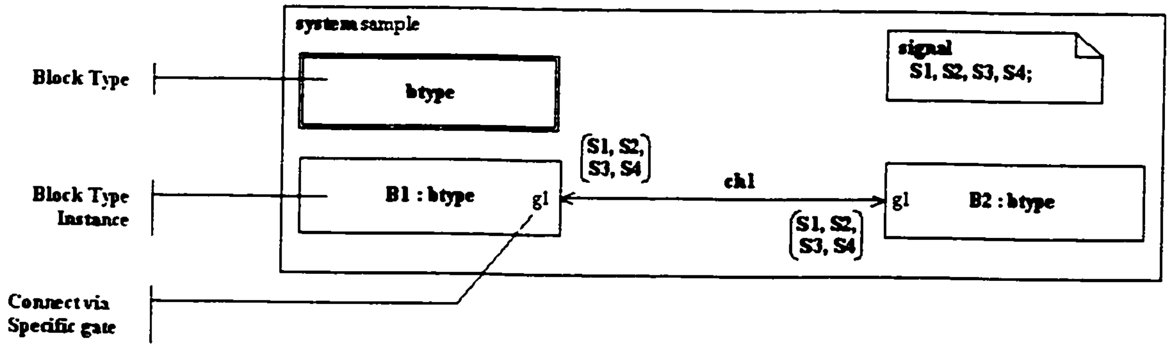


Figure 2.19: SDL specification using types.

A designer must be familiar with the following SDL components in order to use types:

- *Process Type:* The behavior of a process type is defined using extended finite state machines just like a process. The only distinction between the two is that a process type is instantiable.

- *Block Type:* A block type is conceptually the same as a block except that it represents an instantiable element. A block type can contain other block types (and block type instances) and process types (and process type instances).

- *Gate:* A gate is used to specify the interfaces that are used to connect type instances together.

In Figure 2.19, each block type instance has a label near the channel end (arrowhead). The label in this case is "g1", and this represents an SDL gate. The gate actual gate definition is contained in the block type specification as shown in Figure 2.20. A feature of SDL present in the diagram is a *signallist*, which is simply a variable name used to represent a set of signals. Inside the block type specification, there is a process type defined, and a process instance. The architecture of the system shows that gate g1 connects the external environment (with regards to the block type) to the process type instance.

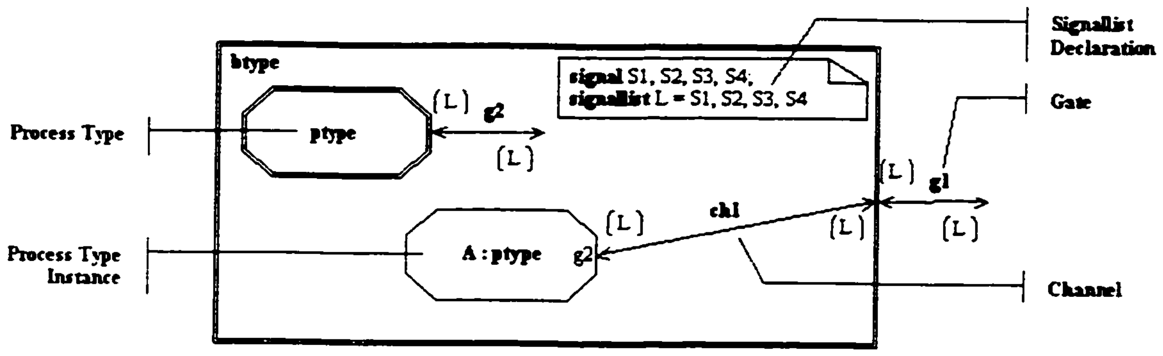


Figure 2.20: Block Type specification.

2.3.2 Behavioral SDL

Once the structure of the system has been determined, the behavior must be defined. As was previously mentioned, the behavior is defined using extended finite state machines inside of process types (and processes). A subset of the graphical notation is presented in this section.

An EFSM is either in a stable state or in a transition between states. In SDL, a process has an input queue where incoming messages are stored until they are consumed. A transition can only be triggered by the reception of a signal. A transition always has a starting state and a terminating state. Figure 2.21 shows an example of an EFSM with some elements that are used to define the process behavior, they are [4]:

- *Variable Declaration:* The declaration of any local variables can be made using a text symbol as previously seen.
- *Start Node:* The start node defines where the execution of the EFSM begins when the process is first initialized.
- *State:* A stable point in the behavior of a process.
- *Input:* A transition can be triggered upon reception of a signal. As long as the input queue of a process is empty, the transition will not start.
- *Output:* During a transition, a process can send signals through channels that connect it to other processes, the environment, or itself.
- *Save:* The input queue of a process may have many messages waiting to be consumed. If the current state does not allow for the consumption of the message at the head of the queue, SDL will discard the message. The save construct allows the process to bypass that message and consume another one while saving the bypassed message for later consumption.
- *Decision:* A transition may be split into multiple branches depending on a condition. It is not necessarily a Boolean operation. A decision consists of two parts, the question and the possible answers.
- *Task:* A task is used to assign a new value to a variable.
- *Stop:* At this node, the process terminated execution.

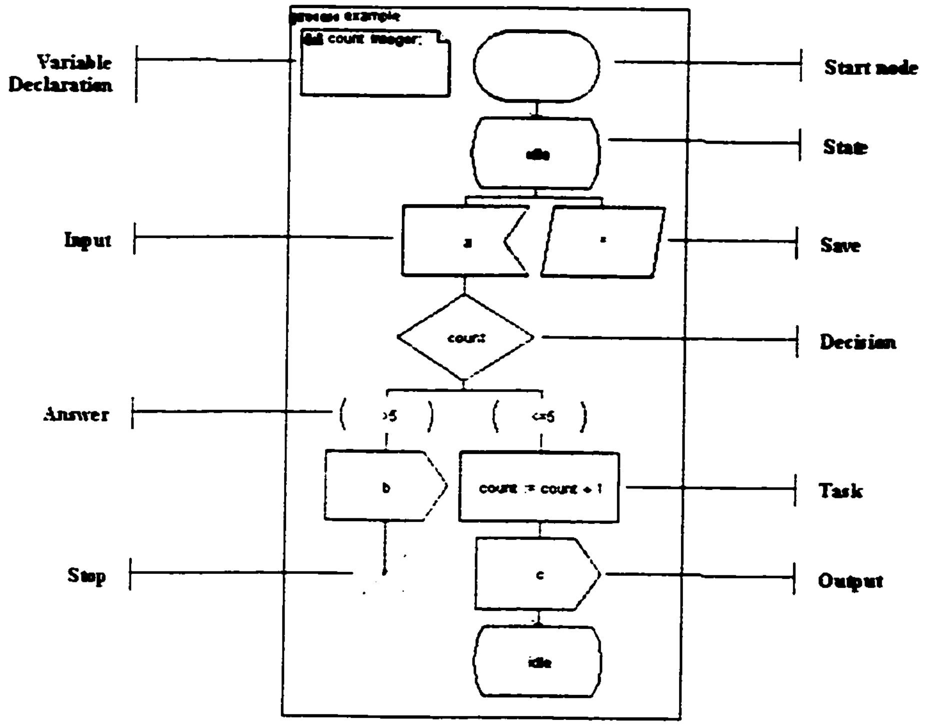


Figure 2.21: An example of an SDL behavioral specification.

2.4 MSC to SDL Methodology

As previously mentioned, the approach presented in this thesis builds on an existing methodology for the automatic generation of SDL specifications from design MSCs and a given target architecture specified in SDL [14, 15, 19]. The generated SDL behavior, expressed as extended finite state machines, will be consistent with the original MSC specification. Moreover, the generated SDL specification is free of deadlocks and common design errors that can be encountered in distributed system specifications.

The order of sending and consuming events in MSCs is explicitly defined, but the actual arrival order of messages into the input queue of a destination process is not. The arrival order depends on the system architecture and the process interleaving. An SDL process discards messages that are not expected at the current state, but which may be needed at a later state. When the process reaches that later state, it will block because the signal has been discarded, which can lead a system to eventually deadlock. To avoid this possibility, the MSC-to-SDL methodology detects that for a given state, message interleaving is possible, and the SDL save construct is used in order to save an out-of-order message for later consumption.

The methodology is best explained through the use of an example. The bMSC in Figure 2.22 shows the desired behavior of the system under consideration. To illustrate the approach, two different architectures are used as shown in Figure 2.24.

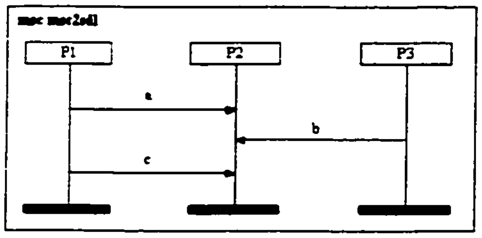


Figure 2.22: bMSC describing desired behavior.

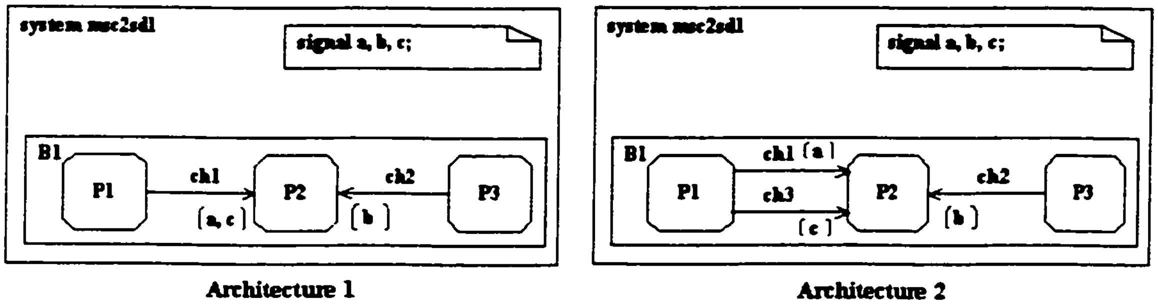


Figure 2.23: SDL architectutures.

The architecture of the system affects the generated behavior, as shown in Figure 2.24. Processes P1 and P3 are the same for both architectures, but the behaviors for process P2 are different. For the first architecture, in state "s1", message "b" can arrive before message "a", and therefore must be saved at that state. Examination of the second architecture reveals that "a" and "b" are sent to P2 via two different channels. If "a" is delayed and "c" arrives first, it must be saved at "s1" or else the system will deadlock at state "s3".

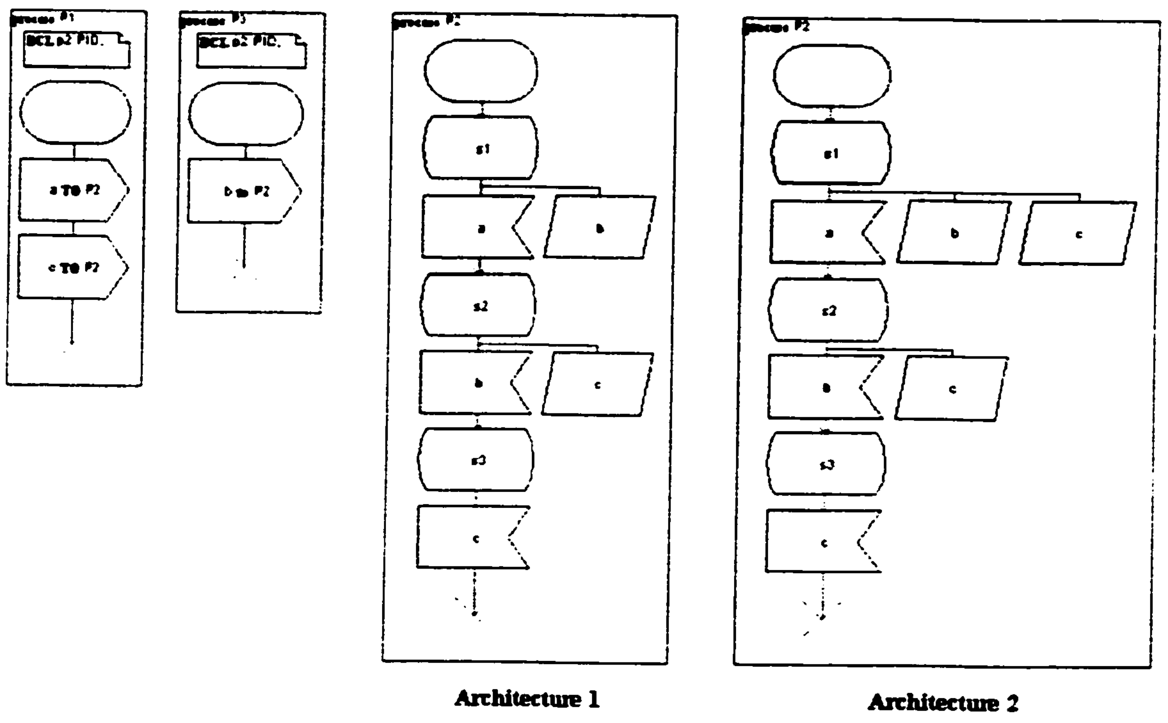


Figure 2.24: Resulting SDL behavior.

The addition of the SDL saves to the generated process behaviors guarantees that deadlock will be avoided, and the messages will be consumed according to the specified orders in the bMSC specification. The MSC-to-SDL approach detects errors such as deadlock, process divergence, non-local choice and unspecified reception. The interested reader can refer to [14, 15, 19] for more detailed information.

Chapter 3

Stepwise Refinement of Message Sequence Charts

This chapter describes our MSC use case refinement approach as introduced in [20]. The conformance relation that we developed is first explained. Next, our stepwise refinement approach is described for bMSC, and then expanded to include HMSC. The approach is then illustrated with an example.

3.1 Introduction

Distributed software systems, like any software system, go through requirement, design, implementation and testing phases. Ensuring consistency throughout the software process is a difficult task. At the start of the development process, the target system is defined in high-level interactions with the user called Use Cases. The Use Cases represent an abstract view of the system that will be refined as the project progresses until the final product is delivered. During the many phases, mistakes due to ambiguities or omissions in the specs, or miscommunication between participants can occur. These types of errors may not be detected until late in the development process, or even after the product is delivered, and can be expensive to correct.

Use Cases are generally expressed in a textual format that states that when a user performs some action X, the system responds with Y. Since the English language can be used to express the same ideas in many ways, errors due to simple misinterpretations frequently happen. If a more formalized approach for documenting use cases were used, these types of errors could be virtually eliminated. Furthermore, having use cases in a formalized notation enables the introduction of higher levels of automation than are currently available.

The MSC language is currently used to express desired program executions at low levels of abstraction, which will be referred to as design MSCs. When a designer sits down in front of a blank page to translate a use case into a complex scenario using MSCs or UML sequence diagrams, they first start with a few messages being passed between processes. Over time, the designer will incrementally add more details until the specification is complete. This process is unstructured and cannot guarantee consistency with the use case being modeled.

Since MSCs can be used to express different levels of abstraction, it *should* be possible to use them for expressing use cases as well as design MSCs. Since the design MSC will be a representation of how the use case is actually implemented, it is evident that there exists a relation between the two. Furthermore, if a relationship can be defined between the use case and design MSCs, there should be a definable relationship between intermediary MSCs.

The approach develops a use-case model through interactions with the system designers and the customers as described in [7, 8, 9]. The result of this represents the functional requirements of the system under construction. Once the use cases have been agreed on, the designers must specify the architecture of the system. Guided by this architecture, the use cases are then refined in a stepwise manner. The method defines two types of refinement, horizontal and vertical. Horizontal refinement consists of adding new messages or actions to existing MSC axes. Vertical refinement consists of decomposing an axis into at least two other axes following the architecture of the system. In both cases, the enriched MSC must conform to the previous (or parent) MSC. It is therefore necessary to define a conformance relation between MSCs that can be used to validate refinement steps made by the designer. These relationships will be more fully discussed in the rest of this chapter.

3.2 Describing Use Cases with MSC

Jacobson introduced the concept of use cases in the Object-Oriented Software Engineering (OOSE) method [9]. Use cases are now part of the UML standard. The fact that they are not formalized allows for flexibility in the early stages of development. This flexibility can be a detriment if inexperienced designers are expected to interpret the use cases and then produce a design specification. In [13], the authors propose a method for formalizing use cases using MSCs. We follow this approach for the modeling of use cases with the MSC language, and introduce a stepwise refinement process of these use cases into design MSCs.

3.2.1 Basic MSC Use Case

Basic MSCs, or bMSCs, are simple diagrams that capture the interactions between system components, and between system components and the environment. For use case modeling, bMSCs are used to show the interactions between the system and the environment as illustrated in Figure 3.1. The user sees a "black box" represented by a single process instance in the MSC. The axis represents the system boundary that interacts with the user. In this example, the user sends message "a" to the system, and the system reacts by sending message "b" to the user.

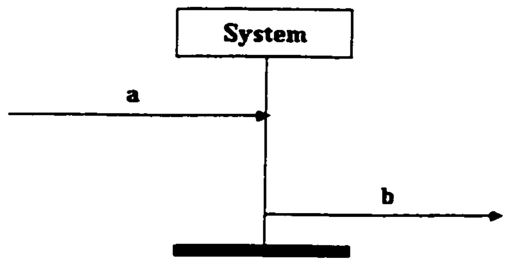


Figure 3.1: A bMSC use case.

For simple use cases, a single bMSC is sufficient to show the behavior, but complex behaviors such as conditional executions or alternatives can lead to large and unreadable bMSCs. To avoid this sort of problem, high level MSCs are considered.

3.2.2 Using High-Level MSC

High-level MSC, or HMSC, allows for a more abstract view of the system [12, 13]. HMSCs improve the readability of the system by hiding low-level details and showing graphically how a set of MSCs can be executed to produce a program trace. They are viewed as roadmaps composed of MSCs using sequential, alternative and parallel operators. HMSCs are seen as directed graphs where the nodes are: start symbol, end symbol, MSC reference, a condition, a connection point, or a parallel frame.

For more complex use cases, it is easier to use several bMSCs and then combine them using a single HMSC. Figure 3.2 shows how a bMSC can be split up into multiple bMSCs and the relations between the new bMSCs are shown in a HMSC. This minimizes the use of operators within bMSCs and results in more simple and straightforward MSCs.

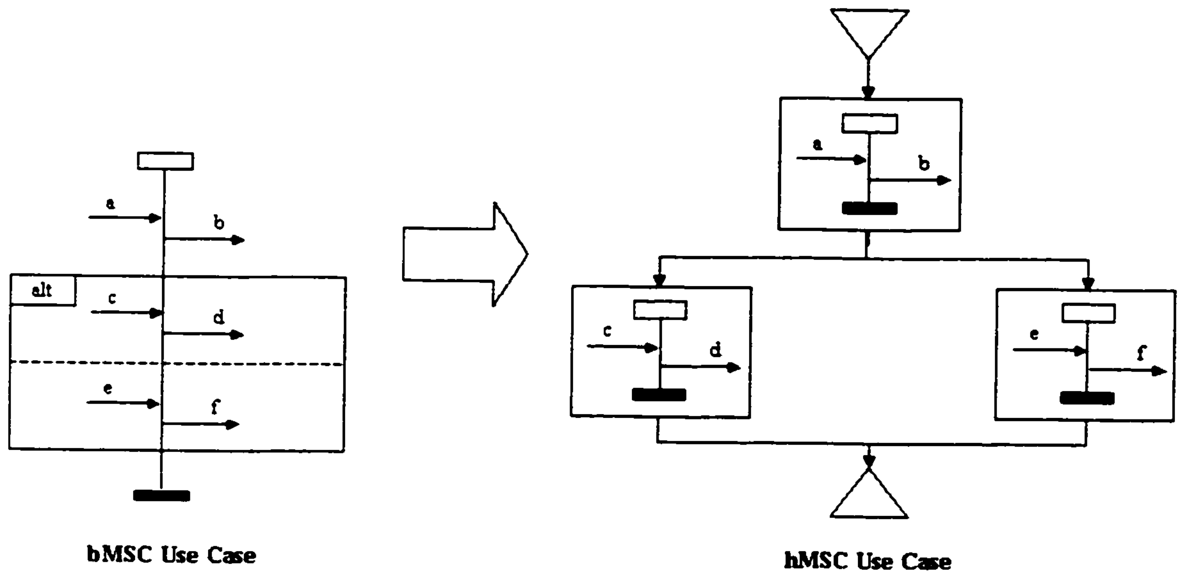


Figure 3.2: A HMSC use case.

A use case describes one possible usage of a system. Most systems will have many functions that will result in many use cases. In order to be able to organize many use cases and to allow a designer to view and navigate them with ease, a HMSC which references the HMSC use cases could be used to show an even higher level of abstraction. Figure 3.3 shows a HMSC that references two use case HMSCs.

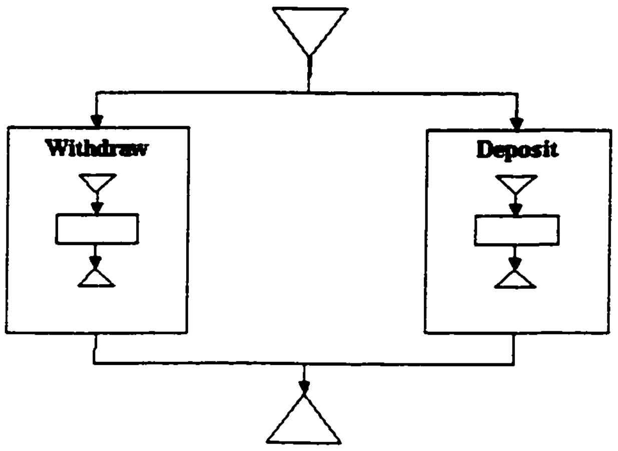


Figure 3.3: Multiple use cases in one HMSC.

Recall that a use case specifies an interaction with a user from the start of a scenario until its completion. If there were multiple use cases that start the same way, and then diverge at some later point to perform different functions, there would be repetition of the initial behavior across multiple use cases. In order to avoid this repetition, the shared behavior can be defined in one MSC that can be executed before the different scenarios diverge. Figure 3.3 shows two use cases that are common functions of an ATM machine. Before one can withdraw or deposit money from/to an account, a user must login first. This login step is an example of a shared behavior between the two use cases that can be extracted to create a pseudo use case. This is shown in Figure 3.4 where a new HMSC reference "Login" has been added. There are two different execution paths in the HMSC. Each complete path represents a use case.

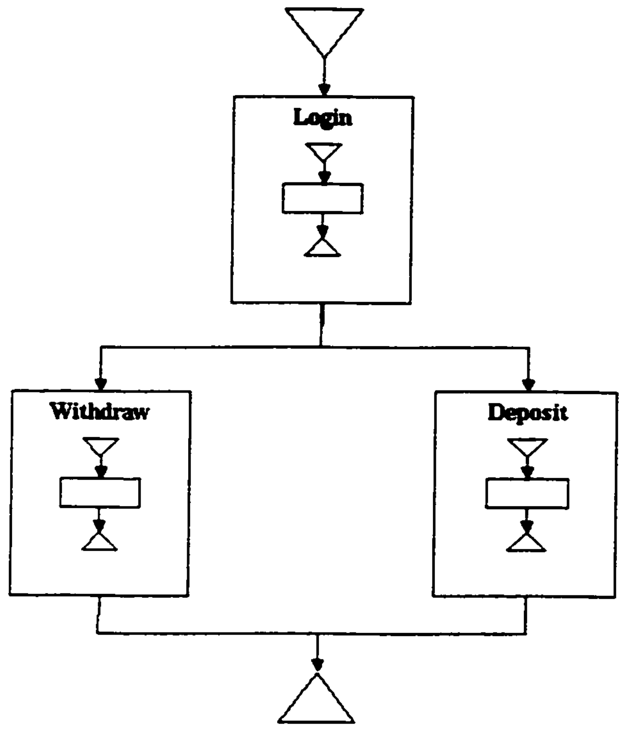


Figure 3.4: Extracting redundant behavior from use cases.

Using three levels of abstraction allows us to specify the use cases for an entire system. Figure 3.5 shows how the various levels relate to each other.

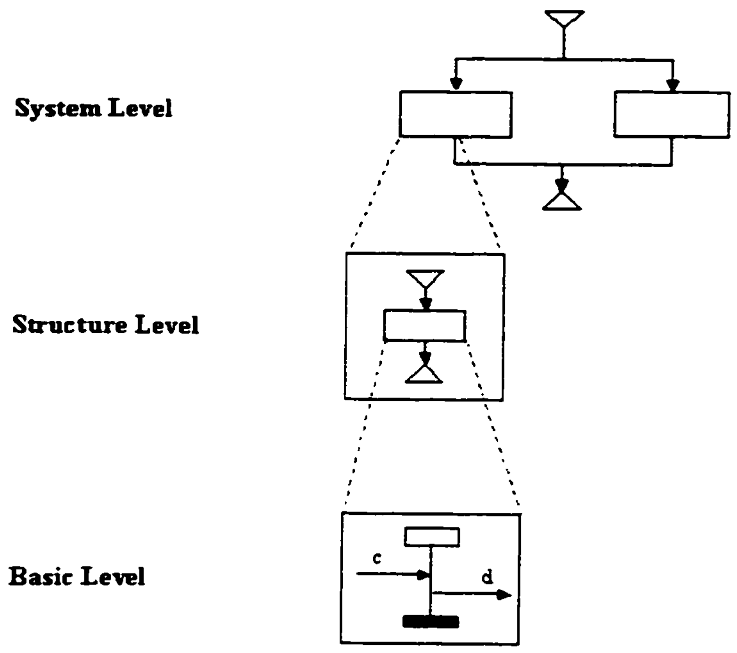


Figure 3.5: Three levels of abstraction.

The system level expresses the functional view of the system by using a top-level HMSC. An abstract view of the system as a set of use cases is given at that level. The structure level describes each use case (using HMSC) without going into details. The basic level shows the interactions between the system and the environment using bMSC. A bMSC at that level cannot refer to a HMSC as allowed in the standard [10]. A bMSC at the basic level does not contain alternatives or optional behaviors. Any alternative or optional behavior can be represented using HMSC at the structure level. In this thesis, the HMSC language is restrained to the weak sequencing operator, alternative operator and iteration. The parallel operator is not taken into consideration.

3.3 Conformance Relation for MSCs

In [20], conformance relations between bMSCs and HMSCs have been introduced and are explained in this section.

3.3.1 bMSC

The bMSC refinement process consists of several steps. M_k is the bMSC obtained after refinement k. In order to preserve, throughout the refinement process, the semantics of the original use case defined by the user, each newly obtained bMSC should preserve the semantics of its parent bMSC. In other words, bMSC M_n must preserve the behavior described by bMSC M_{n-1}. M_n must have the events of M_{n-1} and for these events M_n preserves the orders defined in M_{n-1}. The new messages and events introduced in M_n are not constrained. Informally, we say that a bMSC M_2 preserves the behavior of (or *conforms to*) a bMSC M_1, if and only if for each axis A_{1n} in M_1 there is a corresponding set of axes $\{A_{21}, A_{22}, \ldots, A_{2n}\}$ in M_2 with all events of A_1 included in $\{A_{21}, \ldots, A_{2n}\}$, and all the orders defined between events in M_1 are preserved in M_2.

For a formal definition of the conformance relation, a formal definition of a bMSC is needed. The definitions introduced in [21, 22] are used for this purpose.

Definition 1 (bMSC): A bMSC is a tuple $<V, <<, P, M, L, T, N, m>$ where

- V is a finite set of events,
- $<< \subseteq V \times V$: is a transitive and acyclic relation,
- P is a set of processes,
- M is a set of message names,
- L: $V \rightarrow P$ is a mapping that associates each event with a process,
- T: $V \rightarrow$ {send, receive, local} defines the types of each event as a send, receive or local,
- N: $V \rightarrow M$ maps every event to a name
- m: a partial function that pairs up send and receive events.

The relation << is defined between the sending event (s_m) and the receiving event (r_m) of message "m" as $s_m << r_m$, and between events e_1 and e_2 in the same axis, $e_1 << e_2$, if e_1 appears before e_2.

Definition 2 (Conformance for bMSCs): A *bMSC* $MSC_2 = <V_2, <<_2, P_2, M_2, L_2, T_2, N_2, m_2>$ *conforms to a bMSC* $MSC_1 = <V_1, <<_1, P_1, M_1, L_1, T_1, N_1, m_1>$, if and only if there exist an injective mapping $\Gamma: V_1 \rightarrow V_2$ and a surjective function $\phi: P_2 \rightarrow P_1$ such that:

- $L_1(e) = \phi(L_2(\Gamma(e)))$
- $T_1(e) = T_2(\Gamma(e))$
- $N_1(e) = N_2(\Gamma(e))$
- if $m_1(e) = f$ then $m_2(\Gamma(e)) = \Gamma(f)$
- if $e <<_1 f$ then $\Gamma(e) <<_2 \Gamma(f)$

The conformance relation introduced in [20] is similar to the matching relation defined in [21, 22]. However, there are two major differences. The first one is related to the MSC semantics. Our conformance relation does not distinguish between the visual order and the enforced order as is done in [21, 22]. The visual order being the transitive and reflexive closure $<<^*$ of $<<$, while the enforced order depends on the communication architecture where some visual orders may not hold, because of race conditions [23]. In the formal semantics of MSC (for bMSC [24]), the visual order has to be enforced. The issue of whether a specific architecture allows for order or not is another issue. The second difference, as mentioned earlier in our conformance relation, is an axis in bMSC M_n may correspond to a set of axes in bMSC M_{n+1}. In other words, there may exist a one-to-many relationship between axes, contrary of the one-to-one relation in [21, 22].

Mauw and Reniers have proposed a refinement relation for interworkings [25]. This refinement consists of decomposing an instance into its constituents and adding internal messages between these constituents. Beside the semantic issues, our conformance relation can be seen as a combination of the refinement relation in [25] and the matching relation in [21, 22]. In this thesis, the term "refinement" and "conformance" are not synonymous. The term "refinement" to means that a MSC has been modified, but does not ensure conformance.

For an illustration of the conformance relation, consider the bMSCs in Figure 3.6. A mapping Γ between M1 and M2 that associates each event to itself and a function ϕ that associates each process to itself satisfy the conditions in Definition 2. bMSC M2 conforms to bMSC M1. A message was added to M2 and as a result more messages and events are in M2, but these events preserve the orders defined in M1. The bMSC M3 also conforms to M1. However in the case of M3, the function ϕ associates P21 and P22 to P2.

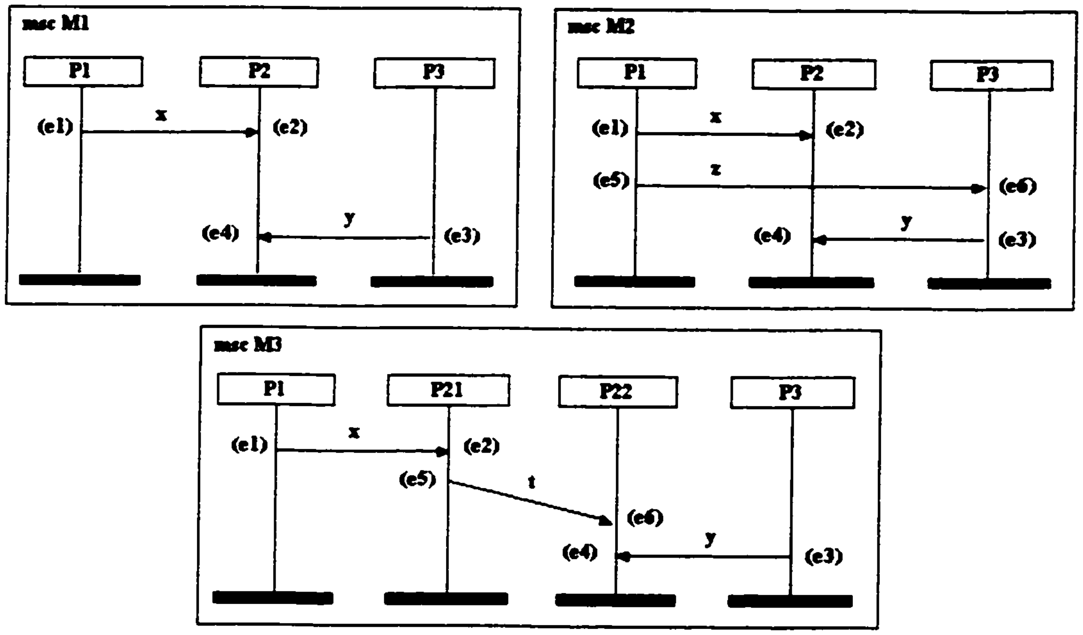


Figure 3.6: Examples of conformance between bMSCs.

Now consider the bMSC M4 in Figure 3.7. In this case, no mapping of events that preserve the orders defined in M1 exists. In fact, reception of message "x" by P21 and reception of "y" by P22 are not ordered as specified in M1.

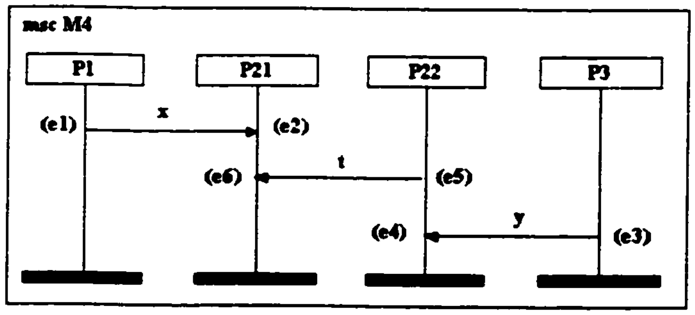


Figure 3.7: Example of non-conformance between bMSCs.

3.3.2 HMSC

In order to build complex behaviors from simple behaviors, the MSC standard [10] has defined weak sequential, iteration, alternative and parallel compositions of MSCs. Informally, weak composition of two bMSCs can be seen as a concatenation of the axes of the common processes in the bMSCs.

Definition 3 (Weak composition of bMSCs). Given two bMSCs, *M1* = $<V_1, <<_1, P_1, M_1, L_1, T_1, N_1, m_1>$ and *M2* = $<V_2, <<_2, P_2, M_2, L_2, T_2, N_2, m_2>$, with $V_1 \cap V_2 = \varnothing$, M1 o M2 = $<V_1 \cup V_2, <<, P_1 \cup P_2, M_1 \cup M_2, L_1 \cup L_2, T_1 \cup T_2, N_1 \cup N_2, m_1 \cup m_2>$, where $<< = <<_1 \cup <<_2 \cup \{(e_1, e_2)$, such that $L_1(e_1) = L_2(e_2)$, $e_1 \in V_1$ and $e_2 \in V_2\}$.

From the syntactical point of view, a HMSC defines a roadmap where MSCs are combined using weak sequential, iteration, alternative and parallel compositions. As previously mentioned, the parallel operator is not taken into account. From a semantic point of view, a HMSC is defined as a potentially infinite set of alternatives of (infinite) weak sequential composition of bMSCs [20].

Definition 4 (HMSC). A HMSC H1 is a set of sequences seq_i (or bMSCs), with seq_i = Mi1 o Mi2 o Mi3 o ... o Miq, where Mij is a bMSC, for j = 1, ..., q and i = 1, ..., n.

The set and the sequences could be infinite. The iteration is defined as the repetition of a given bMSC using weak sequential composition.

As defined in [20], a HMSC H1 conforms to HMSC H2, if and only if for each alternative bMSC M1 in H1 there is a bMSC M2 in H2, such that M2 conforms to M1. M2 preserves the behavior of M1. H2 may contain more alternatives than H1, but each alternative of H1 is preserved in at least one alternative of H2.

Definition 5 (Conformance for HMSCs). Given two HMSCs, H1 = $\{seq1_i, i = 1, ..., n\}$ and H2 = $\{seq2_j, j = 1, ..., m\}$, *H2 conforms to H1*, if and only if for each $seq1_i$ in H1 there exist $seq2_j$ in H2 such that *$seq2_j$ conforms to $seq1_i$*.

The conformance between $seq1_i$ and $seq2_j$ is given in Definition 2. Both conformance relations (for bMSCs and HMSCs) are transitive and reflexive.

3.4 From Use Cases to Design Specification

The ultimate goal of the approach is to enable a designer to refine the initial use case MSC incrementally into a MSC design specification while preserving the interactions specified by the use case. The refinement is guided by the system architecture. At every stage of the refinement process, the resulting MSC is checked for conformance against the parent MSC.

3.4.1 Stepwise Refinement of bMSCs

A use case is an abstract description of some desired functionality provided to the user of a system. The use case description sees the system as a black box that generates some output for a given input. Typically, designers will use their experience to generate design specifications from a textually represented use case. This can produce errors in implementation if the use cases are unclear. How these types of errors can be avoided is the goal of the approach proposed by this chapter.

Using MSCs to describe use cases would result in clearly defined graphical representations of desired behavior that are easy to understand by designers. This would reduce the risk of misunderstanding, but the risk of error introduced during the design phase still exists. The reason for this is that in general, designers translate high-level use cases into complex behavioral specifications in an ad-hoc manner. It would be greatly beneficial if clearly defined methods were to enable software engineers to go from high to low-level specifications in an incremental fashion with some guidance.

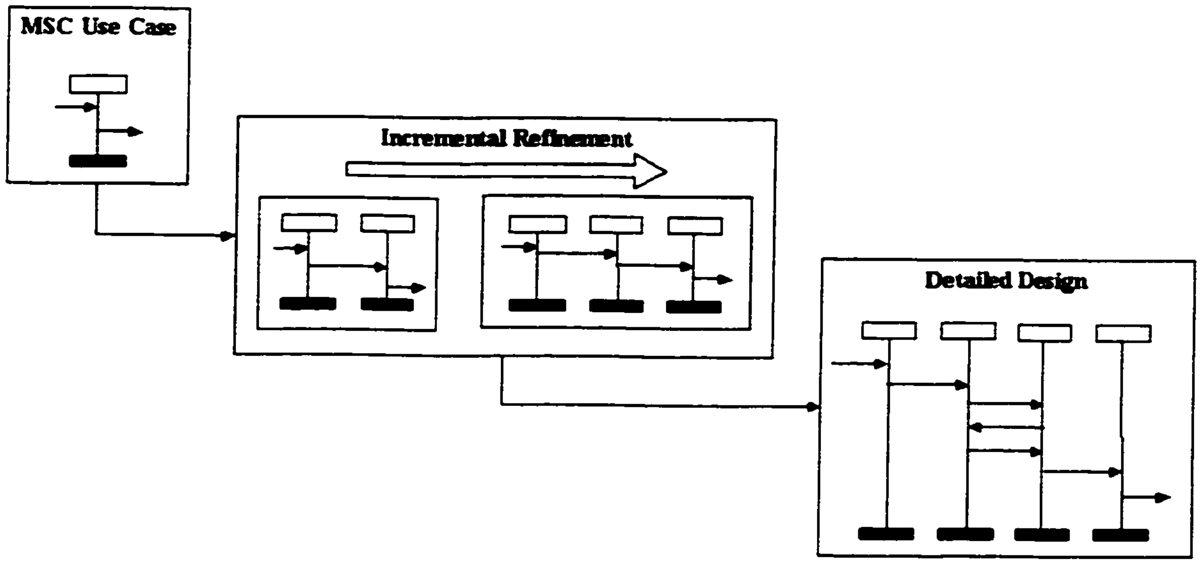


Figure 3.8: Overall refinement process.

Figure 3.8 shows how a use case given in the form of a MSC can be incrementally refined into a design specification. By using this sort of refinement coupled with the conformance relation discussed previously, a method of going from use cases to design specifications can be developed. This will reduce the risk of inadvertent errors being introduced by designers.

The system architecture plays an important role in the stepwise refinement of MSC use cases. A refined MSC must conform to its parent MSC and follow the underlying system architecture. This system architecture can be represented using either UML or SDL. The stepwise refinement methodology of use cases is to be used in conjunction with existing tools for translation of a UML architecture into a SDL architecture [26], and the generation of a SDL specification from a given target architecture and MSCs [14].

3.4.1.1 Vertical Refinement

A system can be viewed as a black box that can respond to user input. This box is composed of other components that in turn can contain components. The entire system can be viewed as a hierarchy. Figure 3.9 shows an example of such a system hierarchy. The topmost level represents the system level and each sub-level represents a more detailed view of the system. Vertical refinement is closely related to the architecture of the system.

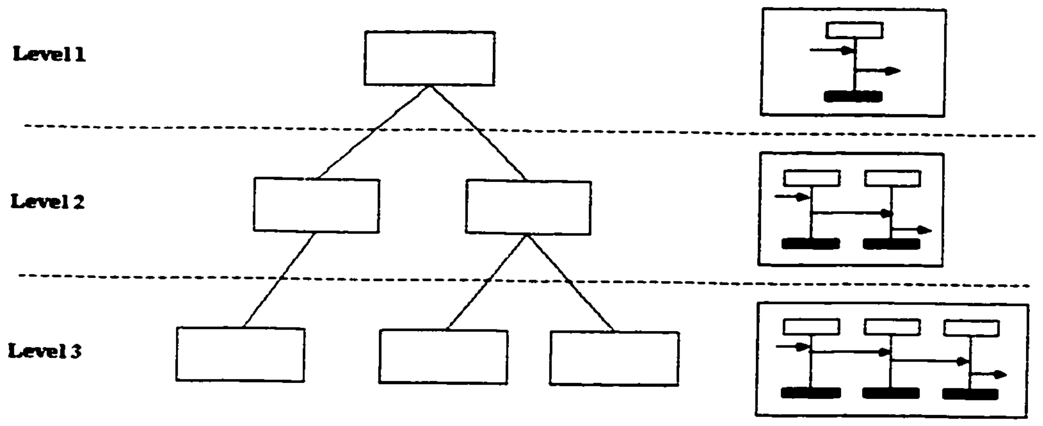


Figure 3.9: System hierarchy.

The system in Figure 3.9 has three levels. Each level has a corresponding MSC that describes its behavior. The Level 1 MSC represents a use case for the system. The second level shows the single component of the top level decomposed into two components. This decomposition is reflected in the MSC for level 2, where there are two process instances. In fact, the number of process instances is always equal to the number of components present in the system architecture for the level in question. As the refinement process goes deeper into the architecture, the generated MSCs will have more process instances to reflect this.

3.4.1.2 Horizontal Refinement

By inspecting the MSCs in Figure 3.9, it is apparent that more detail about the system behavior is added at every level. This cannot be done automatically, but must be done by the designer. New messages can be introduced in the lower levels, but the previously defined event orders must be preserved. The MSCs of each lower level must conform to the MSCs in the higher levels. Vertical refinement deals with decomposing instances into their lower level constituents. A method for adding more details within a level is needed; this is called horizontal refinement. The goal is to refine an MSC in an incremental manner by adding behavioral information. Suppose a designer wanted to refine the Level 1 MSC from Figure 3.9 into the Level 2 MSC. Vertical refinement of Level 1 to Level 2 would yield the MSC from Figure 3.10a. Horizontal refinement of Figure 3.10a would result in the MSC represented by Figure 3.10b, which is the same as the MSC for Level 2 of the architecture shown in Figure 3.9.

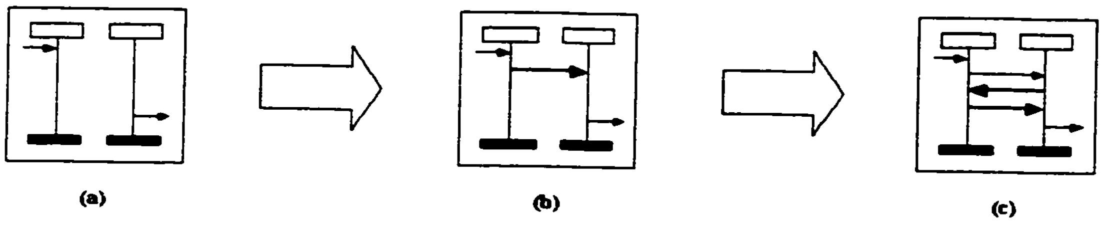


Figure 3.10: Horizontal refinement.

Now suppose the designer wanted to enrich the behavior of Figure 3.10b. Another iteration can result in Figure 3.10c for instance. It should be noted that the MSC in Figure 3.10a is a vertical refinement of the top level MSC and that it represents the system architecture, but it does not conform to its parent MSC. However, after one horizontal iteration, Figure 3.10b does conform to the top-most MSC. In general, a combination of vertical and at least one horizontal refinement is needed in order to generate a MSC that may conform to the previous level.

3.4.1.3 Basic MSC Refinement Methodology

Vertical refinement reflects the architectural decisions, while horizontal refinement allows the designer to make additions to a bMSC. Both types of refinements, vertical and horizontal, are used together to incrementally refine a use case bMSC into a design specification. Figure 3.11 illustrates how both types of refinements are combined to reach a design specification.

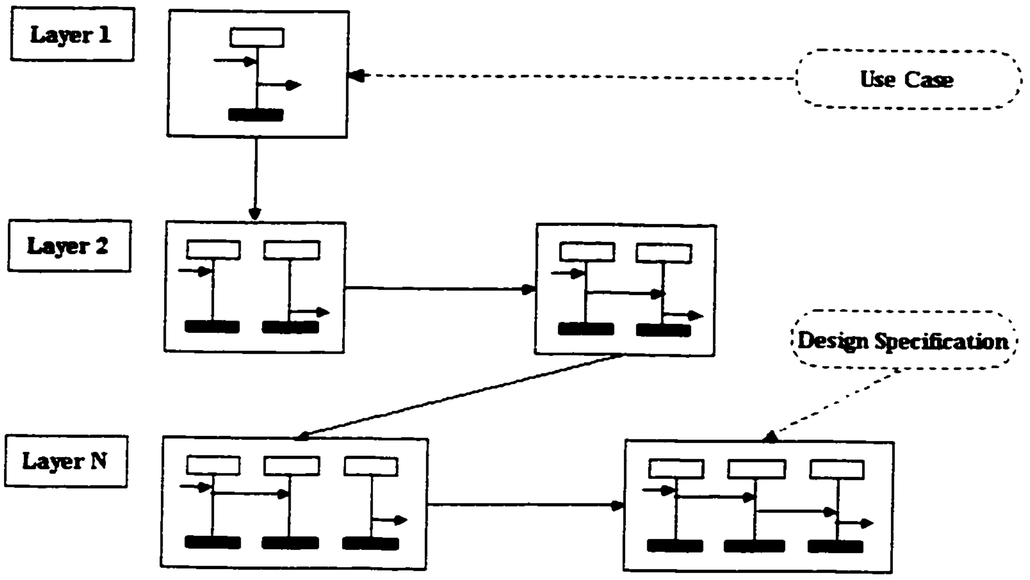


Figure 3.11: Overall refinement methodology.

The refinement process is dependent on the system architecture. The number of layers used in the architecture limit the number of vertical refinements. There can only be one vertical refinement per layer. Each vertical refinement step splits a bMSC instance into several instances according to the architecture. The instances that are not decomposed keep the same names. When an instance is decomposed the designer has to distribute its events among the new instances. For each message "m", the names of the associated events (sending and receiving) are kept unchanged. The messages and events associated with instances that are not refined are kept unchanged and re-generated automatically.

Horizontal refinement is concerned with adding messages, events and local actions to the bMSC. The designer can add new messages, sending and receiving events as well as local actions. The messages that can be used in a horizontal refinement are specified in the architecture of the system at that level and the designer cannot introduce messages that are not specified in the architecture at the current level of refinement. Unlike vertical refinement, there is no limit to the number of horizontal refinements that can be performed at any given layer. After each horizontal refinement, the refined bMSC can be automatically checked via a CASE tool for conformance.

The verification of the conformance relation between bMSCs is implemented with the Event Order Tables (EOT) introduced in [14, 15]. As mentioned earlier, the bMSCs used do not contain alternative or optional behaviors.

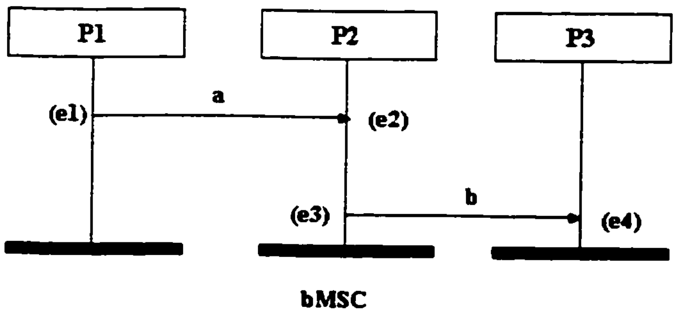
3.4.2 Event Order Tables (EOT)

An EOT for a bMSC is a matrix that shows precedence relationships between events of the MSC. Figure 3.12 shows a bMSC and its corresponding EOT. Each arrow represents the sending of a message from one instance to another. Consequently, each message sent between instances corresponds to two events: sending and receiving. Once the events on a MSC have been labeled appropriately, the EOT can be constructed. Following the MSC semantics and the relation <<, two rules are followed to generate the EOT [14]:

i. Each instance is totally ordered (except for co-regions).

ii. A reception event happens after its matching sending event.

Inspection of each process axis in Figure 3.12 reveals that the first rule must be applied to P2, yielding the relation e2 << e3. Applying the second rule to the MSC of Figure 3.12 yields {e1 << e2, e3 << e4}. The transitive closure <<* of << allows for the construction of the EOT. For instance, e1 << e2 and e2 << e3 implies e1 << e3. A cell in the EOT is set to true if the row event occurs before the column event.



	e1	e2	e3	e4
e1		T	T	T
e2			T	T
e3				T
e4				

Event Order Table

Figure 3.12: A bMSC and its EOT.

Definition 6 (Inclusion for EOTs) [20]. An EOT T_1 is included in EOT T_2, if and only if

i. All the events of T1 are in T2, and

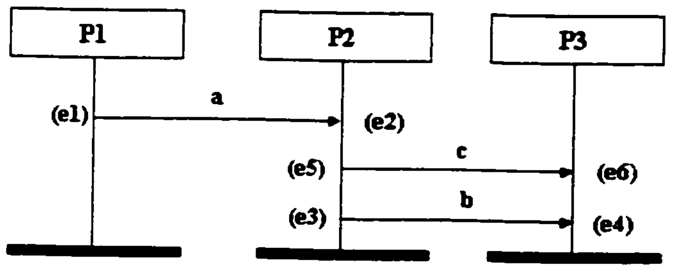
ii. For each pair of events (ei, ej) if ei << ej in T1 then ei << ej in T2.

It can be said that that M_{k+1} conforms to M_k if and only if the EOT of M_k is included in the EOT of M_{k+1}.

Proposition 1 (Conformance of bMSCs using EOTs) [20]. Given two bMSCs, M_k and M_{k-1}, provided that events in M_k are mapped into themselves in M_{k-1}, M_{k-1} conforms to M_k if and only if EOT of M_k is included in the EOT of M_{k-1}.

Proof: The proof is straightforward. The refinement approach (and untimately a tool) maps events in M_k into themselves in M_{k-1}. The orders defined in M_k are preserved in M_{k-1} if and only if the EOT of M_k is included into the EOT of M_{k-1}. **[end proof]**

Figure 3.13 shows a bMSC that conforms to the bMSC in Figure 3.12, the message "c" sent from P2 to P3 has been added. In order to preserve the original event labels, the event labels e5 and e6 have been used for the new events.



Refined bMSC

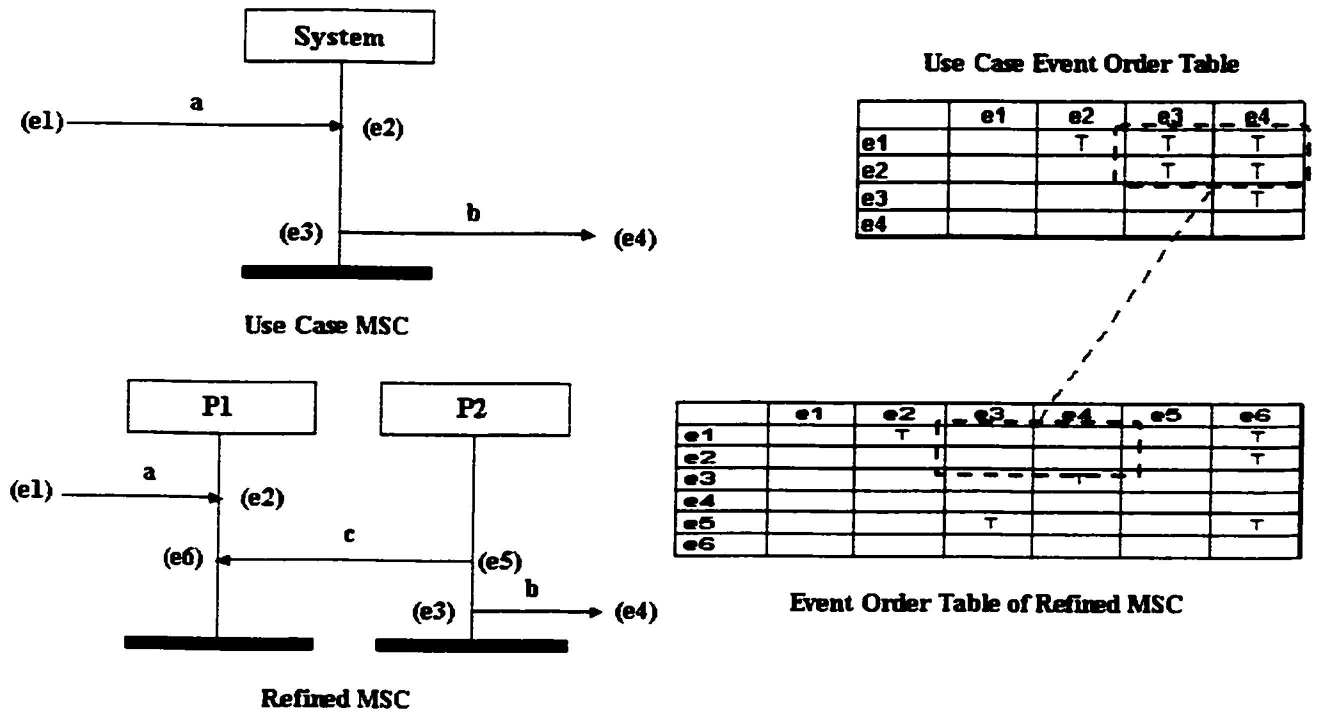
	e1	e2	e3	e4	e5	e6
e1		**T**	**T**	**T**	T	T
e2			**T**	**T**	T	T
e3				**T**		
e4						
e5			T			T
e6				T		

Event Order Table

Figure 3.13: Example of conformance using EOTs.

The EOT in Figure 3.12 specifies an ordering of events that is maintained by the new bMSC. In fact, the EOT in Figure 3.12 is included in the EOT in Figure 3.13. The old relations are bolded in Figure 3.13; as long as these are not violated, the new bMSC conforms to its parent bMSC.

For illustration of a non-conforming example, consider the bMSCs in Figure 3.14. The corresponding EOTs are shown in the same figure. The second bMSC is a refinement of the use case bMSC. The single instance in the use case bMSC is split up into two instances. A new message "c" from P2 to P1 has been added. The EOT of the new bMSC shows how the events are related to each other. The refinement has violated the order e2 << e3, for instance. The circled part of the EOT in Figure 3.14 reflects this violation. Therefore the refinement does not yield a bMSC that conforms to its parent bMSC. This refinement cannot be accepted.



Use Case Event Order Table

	e1	e2	e3	e4
e1		T	T	T
e2			T	T
e3				T
e4				

	e1	e2	e3	e4	e5	e6
e1		T				T
e2						T
e3				T		
e4						
e5			T			T
e6						

Event Order Table of Refined MSC

Figure 3.14: Example of non-conformance using EOTs.

3.4.3 Stepwise Refinement of HMSCs

A HMSC is composed of a number of bMSCs following a certain roadmap. To enrich a use case HMSC, the bMSCs are refined separately. A HMSC is refined according to the following rules:

i. The roadmap remains unchanged.

ii. For vertical refinement, all bMSCs are vertically refined at the same time so that each bMSC has the same number of instances and reflects the same level of abstraction.

iii. Each bMSC can be refined horizontally independently of the others.

Vertical refinements are done automatically according to the system architecture and the current level of abstraction, after which the user can perform horizontal refinements of the resulting bMSCs. However, it is important to note that because a refined bMSC conforms to its original, it is not necessarily true that the corresponding HMSCs conform. The reason for this is that when an axis is decomposed and the messages are distributed across the refined axes, the new message orderings may not respect the ones specified in the original. In Figure 3.15, when M1 is executed before M2, the sending of "x" always precedes the sending of "y". This is not the case for when M1' and M2' are concatenated; P12 *may* send "y" before P11 sends "x", which violates the message ordering in M1 and M2.

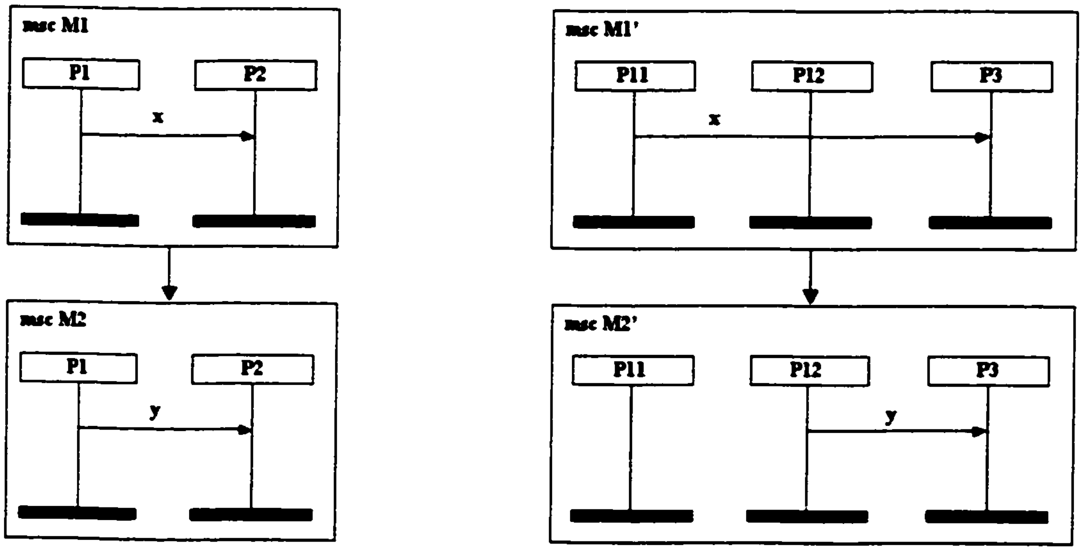


Figure 3.15. Non-conformance between HMSCs.

Theorem 1 (Conformance between HMSCs) [20]. Given a HMSC H1 and its refinement HMSC H2, (with the same roadmap), if:

i. Each bMSC in H2 conforms to its corresponding bMSC in H1.

ii. In the HMSC H2, for each pair of bMSCs, M1' and M2', such as M1' o M2' in H2, for each set of axes {A11, ..., A1n} in M1' and M2' resulting from the decomposition of A1 (in M1 and M2 in H1), the order between the last event of A1 in M1 and the first event of A1 in M2 is preserved,

Then H2 conforms to H1.

Proof. Consider two HMSCs, H1 and its refinement H2 obtained following the rules mentioned above. From a semantic point of view, each HMSC is a set of alternatives of concatenation of bMSCs. Roadmaps of H1 and H2 are identical, each bMSC in H1 has a corresponding bMSC in H2. Any sequence of concatenation seq1 in H1 has its corresponding sequence of concatenation seq2 in H2, using corresponding bMSCs in H2. In order to show that H1 conforms to H2, we have to show that seq2 conforms to seq1. For that we only have to prove that if M1' conforms to M1 and M2' conforms to M2, M1' and M2' are obtained with our refinement approach, M1' o M2' conforms to M1 o M2.

M1' conforms to M1 means that orders in M1 are preserved in M1'. M2' conforms to M2 also means that orders of M2 are preserved in M2'. M1 o M2 (before decomposition of axes) imposes orders on events in the same axes and these orders have to be preserved in M1' o M2'. The second condition of the theorem ensures the preservation of these orders in M1'o M2'. Therefore, M1' o M2' conforms to M1 o M2. **[End of Proof]**

During the refinement of HMSC use cases, the conformance between the new HMSC and its parent HMSC is verified by checking the conformance between each pair of bMSCs. When a bMSC in the new HMSC does not conform to its corresponding bMSC in the parent HMSC, the refinement is not accepted. New refinement and modifications have to be applied to the new bMSC and the conformance checked again. Notice in the refinement approach, the conformance of bMSCs is a sufficient condition and not a necessary one. However, in order to ensure the conformance between HMSCs, the conformance between pairs of bMSCs is enforced. The study of all the possible necessary conditions is left for future studies.

3.5 Application

The advantage of using MSCs is that they are formally defined and comprise of both graphical and textual notations. This means that an algorithm can be applied to bMSCs to automatically generate and compare EOTs. Automation reduces the likelihood of human error and increases the confidence in the generated specifications.

This section illustrates the refinement approach through an example. Figure 3.16 shows a system architecture representing an ATM machine using SDL. There are two processes, namely "ATM" and "BANK". There is a channel that allows the user of the system to interact with the ATM process instance, and another channel allows for the two process instances to interact with each other. The signals these processes exchange with each other and their environment are specified in the architecture.

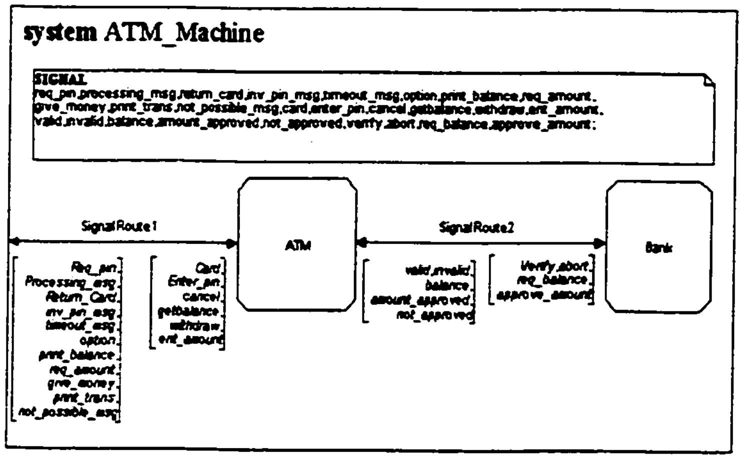

system ATM_Machine
SIGNAL
req_pin,processing_msg,return_card,inv_pin_msg,timeout_msg,option,print_balance,req_amount,
give_money,print_trans,not_possible_msg,card,enter_pin,cancel,getbalance,withdraw,ent_amount,
valid,invalid,balance,amount_approved,not_approved,verify,abort,req_balance,approve_amount;
SignalRoute1
SignalRoute2
ATM
Bank
Req_pin, Processing_msg, Return_Card, inv_pin_msg, timeout_msg, option, print_balance, req_amount, give_money, print_trans, not_possible_msg
Card, Enter_pin, cancel, getbalance, withdraw, ent_amount
valid, invalid, balance, amount_approved, not_approved
Verify, abort, req_balance, approve_amount

Figure 3.16: *ATM_Machine* system architecture in SDL.

Two functions provided by an ATM machine are the ability to withdraw and deposit money. Both of these functions have a common starting point. The user must first insert a card and enter the correct PIN before a transaction can occur. Figure 3.17 shows the system and structure level HMSCs.

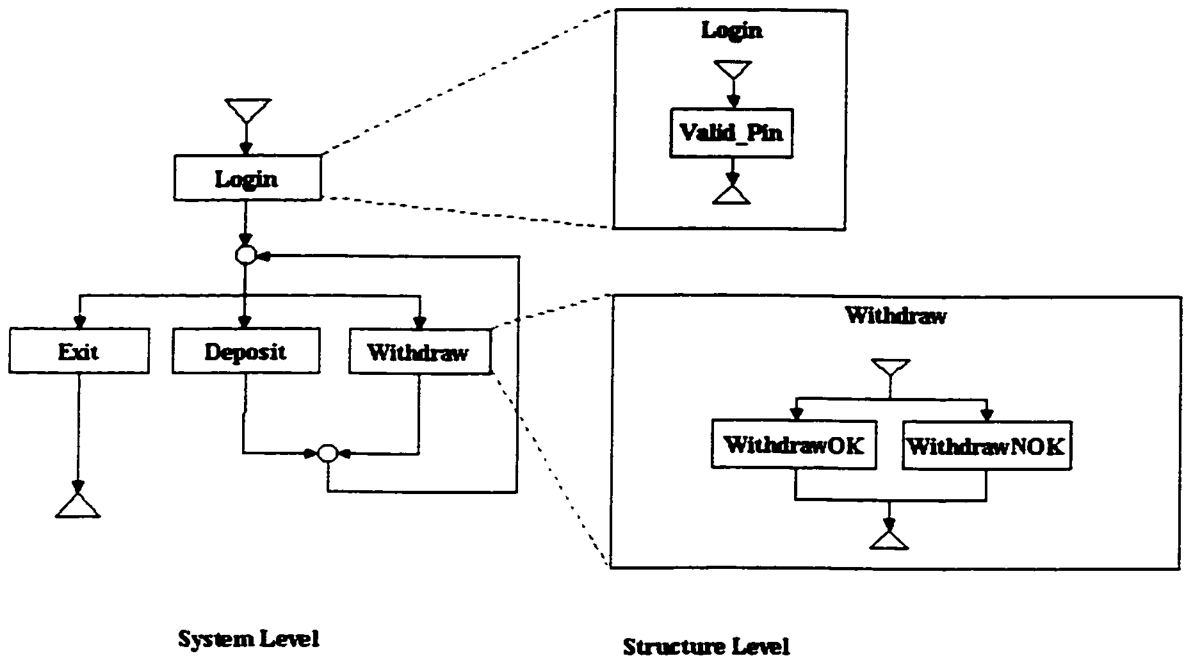


Figure 3.17: System and structure level HMSCs for the ATM example.

The system level shows three different use cases, "Exit", "Deposit" and "Withdraw". The system level HMSC also shows that each use case has the common behavior denoted by the "Login" block. The structure level shows more detail. The "Login" HMSC has only one bMSC, "Valid_Pin". Later on, the system designer can decide to add an "Invalid_Pin" scenario to the "Login" HMSC that would not affect the rest of the system. The "Withdraw" HMSC shows that the withdraw use case can have two results, "WithdrawOK" and "WithdrawNOK".

For the purpose of illustrating the method, refinements on only "Login" and WithDraw" are performed. The same process can be applied to the other use cases. Figure 3.18 shows the two bMSCs "Valid_Pin" and "WithdrawOK" being refined vertically. Note that there is no new behavior added, only a splitting of the instances to reflect the architecture of Figure 3.16.

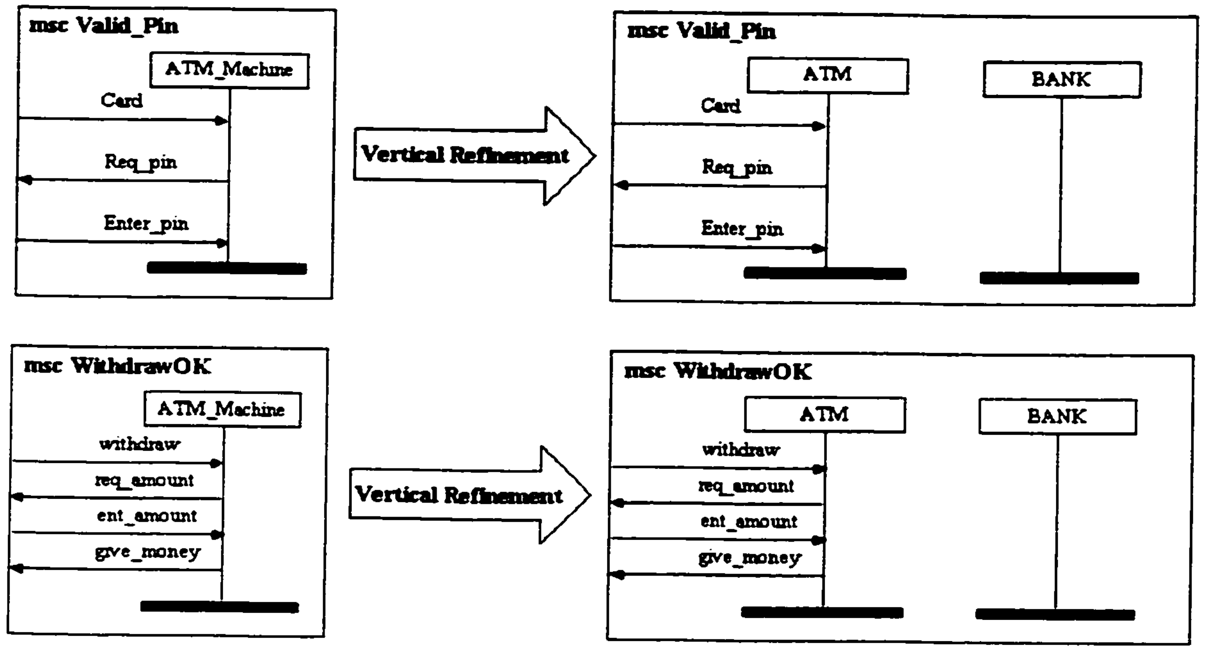


Figure 3.18: Vertical refinement of ATM use cases.

Once vertical refinement has been performed, the designer can now enrich the bMSCs. Horizontal refinements are shown in Figure 3.19.

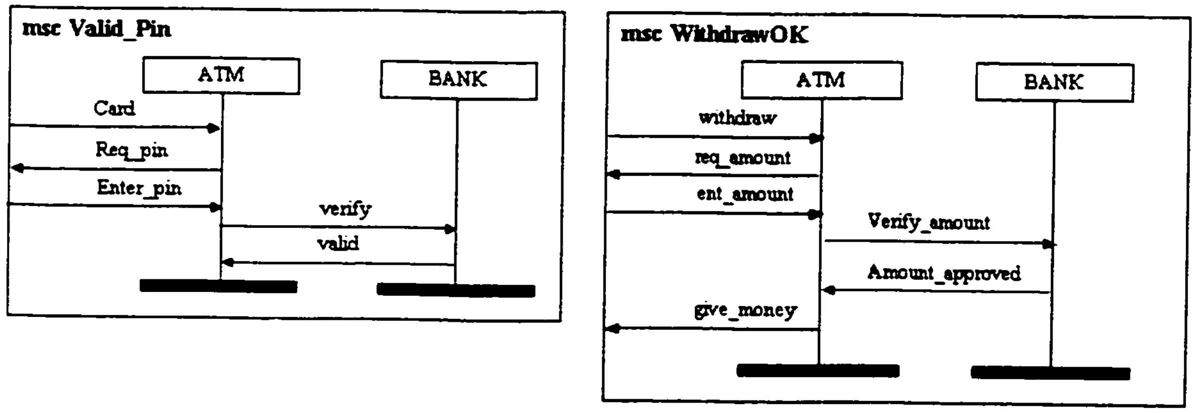


Figure 3.19: Horizontal refinements.

Once the user has finished performing a horizontal refinement, the new MSC is verified for conformance against the previous MSC. If the new MSC conforms, the user can perform another horizontal refinement. Figure 3.20 shows a second refinement performed on the WithdrawOK MSC. Since the new MSC conforms to the MSC in Figure 3.19, by extension, it also conforms to the MSC in Figure 3.18.

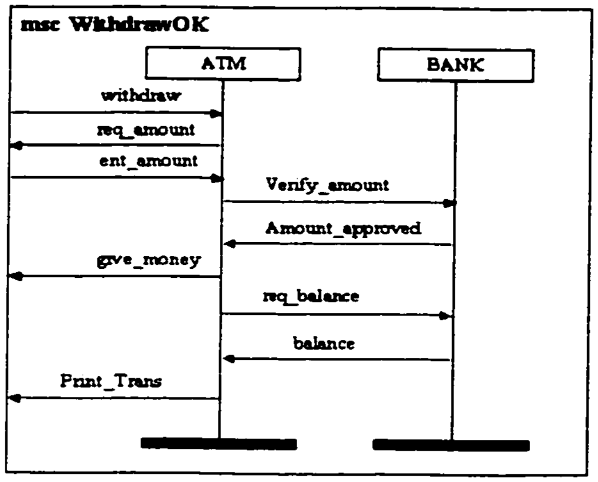


Figure 3.20: Another horizontal refinement of *WithdrawOK.*

Now suppose that the ATM block of Figure 3.16 was further decomposed into three separate processes as shown in Figure 3.21. The ATM block now contains three processes, Dialogue, MoneyHandler and Printer.

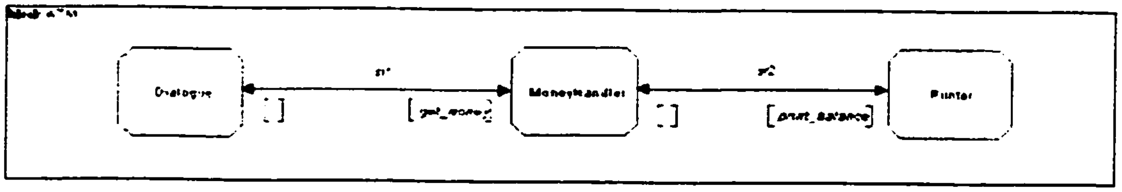


Figure 3.21: Inside the *ATM* Block.

Following the refinement process, the MSC of Figure 3.20 must first be refined vertically to show the decomposition of the ATM block. This refinement is followed by a horizontal refinement with the addition of messages. The resulting MSC, that conforms to the MSC in Figure 3.20, is shown in Figure 3.22.

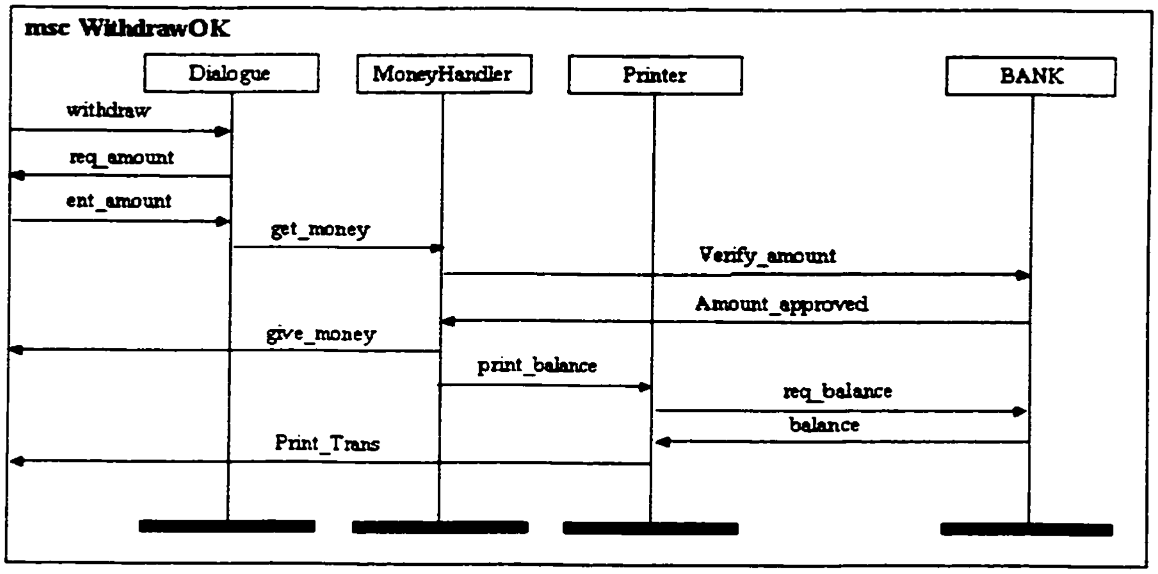


Figure 3.22: Vertical Refinement of *WithdrawOK*.

Chapter 4

UML to SDL

This chapter introduces our stylized use of UML for describing distributed system architectures. The UML to SDL architectural mapping rules that we defined are then presented and the approach is illustrated via two examples.

4.1 Introduction

The Unified Modeling Language (UML) [7] is a set of standard notations for modeling software systems at different stages of their development. Although the lack of formality in UML is beneficial at the early stages of development, this very property changes into a liability when the time comes for simulation, validation and implementation phases, where SDL is better suited. It should also be noted that UML is still relatively new, whereas SDL has been around for many years. Instead of throwing away all the experience accumulated with SDL over the years and starting from scratch with UML, the two can be combined so that the advantages of both languages could be used.

The translation from UML to SDL necessitates a proper mapping between UML and SDL. For that purpose, two different approaches can be taken:

i. Use UML stereotypes to map SDL directly into UML so that UML to SDL translation becomes one-to-one, which is the approach taken in the (Z.109) [27, 28] standard.

ii. Use the standard elements of UML so that a designer need only have basic knowledge of SDL, which is the approach taken in this thesis.

The first approach requires a designer to learn SDL before being able to design a system. Since this method maps SDL into UML, then why not use SDL directly? A drawback is that by importing UML into SDL, the advantages that UML has over SDL in the early phases of development are lost (since we would in a way be formalizing UML by importing a formal language).

The second approach is the one that we have adopted. We use UML in such a way that all the advantages of UML are preserved. Anyone familiar with UML should be able to use the methodology with a minimum amount of learning. Our approach uses standard UML concepts described in the OMG UML Specification 1.3 [7], but in a stylized way similar to a profile.

4.2 Distributed System Architecture Description Using UML

Currently, UML is the most popular language for describing software systems, but it still has deficiencies when dealing with real time distributed systems [29, 30, 31]. A number of researchers and practitioners feel that in order to overcome the limitations of the current version of UML, the language should be extended to support concepts from other notations. There is much debate on the subject, but the prevailing opinion is that extending UML to accommodate everyone (people from different domains) would result in a bloated and unwieldy language. It is also generally agreed that the UML extension mechanisms such as stereotypes and tagged values are sufficient for any domain specific customization [7, 8] The UML extension mechanisms has allowed for the development of UML *profiles* for domain specific applications

A problem faced by designers of real-time distributed systems is that UML, being a semi-formal language, is unsatisfactory for simulation, verification and validation. Real-time distributed systems are much more complex than regular systems and the specification and verification of complex behavior in time is an essential feature not currently offered by UML [29, 30, 31].

The technique introduced in this chapter uses UML to model distributed systems architectures. By architecture, it is meant that different components of the system and their connections for communication minus behavior are modeled. The behavior of such systems will be specified with MSC, which in its latest version [12] includes UML Sequence Diagrams.

Standard UML elements with as little customization as possible are used so that the specifications developed by the technique would be easy to understand. For the description of the architecture of a distributed system, UML Class and Object Diagrams are used. The following subsections explain the way in which these diagrams are used for the description of distributed systems components and their interconnections.

4.2.1 Class Diagram

UML Class Diagrams are used to show static relationships between elements in the system [7, 18, 30]. The way different model elements are employed to accomplish this is described in the following subsections.

4.2.1.1 Subsystems

A package is a method of organizing model elements. A subsystem on the other hand represents a behavioral unit in the model. The subsystem stereotype applied to a package shows this distinction. A subsystem may or may not be instantiable [7, 8, 18]. For the purposes of this thesis, all subsystems are assumed to be instantiable. The graphical representation of a subsystem is shown in Figure 4.1.

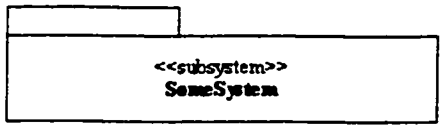


Figure 4.1: Subsystem representation.

Instantiable subsystems are of interest because they have their own threads of execution and can be treated as stand alone processes. As a result, they can be instantiated and deployed across many locations.

Instantiable subsystems are the major building blocks that are used to describe a distributed system. These components are interconnected through channels. If an appropriate method is used for describing these channels, the subsystems can be made to be as weakly coupled as possible, which is a prime concern in object oriented design. Using an appropriate method for channel descriptions will also enable a designer to have tight control over the valid signals that can be passed between components. The following subsections describe our approach for specifying inter-component communication.

4.2.1.2 Communication Between Classes

When describing a distributed system, a distinction between two types of channels must be made:

i. Subsystems to subsystem channels that allow classes inside a subsystem to communicate with a component inside another subsystem.

ii. Component to component channels that allow components to communicate with each other inside a subsystem.

In a distributed system, messages are passed between model elements via channels. The obvious approach to show communication between classes would be to simply connect them with associations as shown in Figure 4.2.

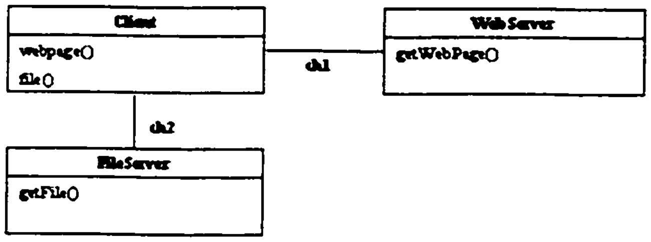


Figure 4.2: Communication through simple associations.

The drawback with this approach is that it is difficult to specifically indicate which signals travel over which channels. How would we specify that "file" could only be sent to class Client via "ch2" but not via "ch1"? This could be accomplished using Type and Implementation Classes.

To show the existence of a channel between two classes, associations between Type classes are used. The channel will then exist between the classes that implement the Types. The messages or signals that can be passed along these channels are the operations that are defined within the associated Type classes. An example of two classes communicating via two channels is shown in Figure 4.3.

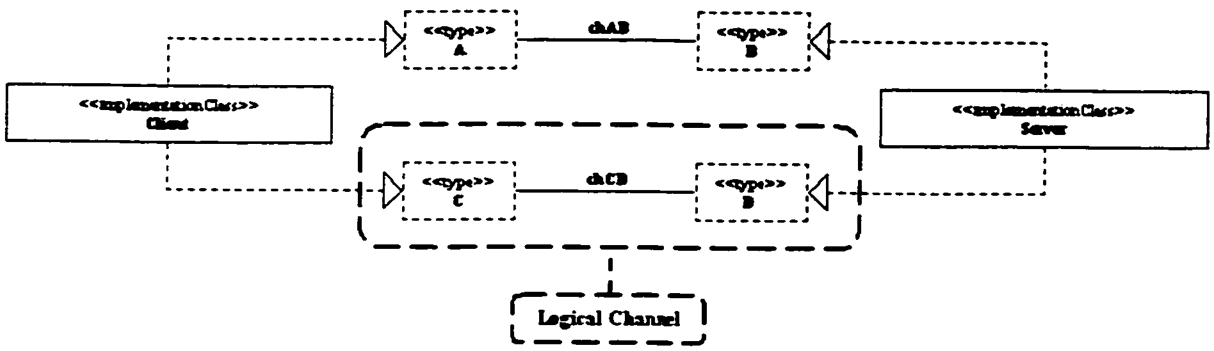


Figure 4.3: Communication using two channels.

The logical channel indicated in Figure 4.3 consists of two type classes. In this particular example, two channels are used to connect two implementation classes. The operations defined over the associated types denote the allowed signals that can be passed over the channel. Unlike the previous example shown in Figure 4.2, defining channels in this manner allows designers to explicitly denote which signals can be sent over specific channels.

This use of type and implementation classes is *similar* to the Proxy pattern [32] where the type classes are analogous to proxies and control access to the implementation classes.

4.2.1.3 Communication Between Subsystems

Another reason why straightforward associations between components are impractical is in the case of inter-subsystem communication. When designing a subsystem that can be deployed, the interface that a subsystem offers the "outside" world must be clearly defined. Using associations in the standard UML manner leads to confusion.

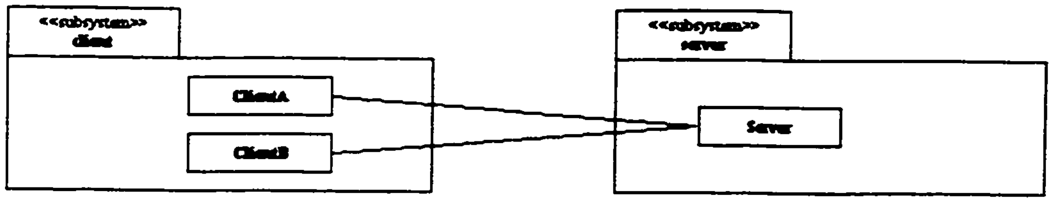


Figure 4.4: Inter-subsystem communication using normal associations.

In the Class Diagram shown in Figure 4.4, there are two associations between the client and server subsystem. Suppose that during runtime, there are five instances of each Client class and one Server object. This means that there would be 10 logical connections with the Server object. If a designer wanted to limit the number of channels between the two subsystems to one channel (perhaps there is a physical limitation), it would be impossible using the notation shown in Figure 4.4. To accomplish this, it is necessary to use some sort of buffer mechanism, which allows components within a subsystem to "see outside" and other subsystems to "see inside" in a strictly defined way. Since packages denote collections of components and are not components themselves, package-to-package communication is done through components contained within the communicating packages. The components in question are Type classes.

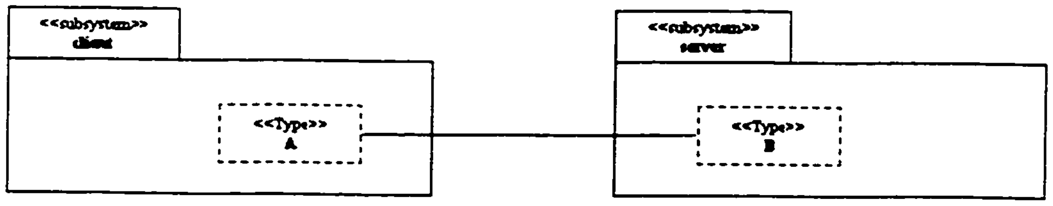


Figure 4.5: Inter-subsystem communication using type classes.

The Type classes shown in Figure 4.5 are depicted using a dotted border to denote the fact that they are *not instantiable*. The services offered by the type classes are implemented by Implementation Classes contained within the subsystem. The Type classes can be seen to behave as interfaces between the two subsystems, and the association between them denoted the communications link between the two and can be thought of as channels between the subsystems.

By using Type classes as interfaces between components in this manner, the subsystems become more modular. Use of straightforward associations as described in Figure 4.4 can result in components that are tightly coupled. By decoupling subsystems as much as possible by strictly defining their interfaces, the possibility of reuse is much greater.

Using type classes to provide a more unified interface to the subsystems is similar to the Façade pattern presented in [32]. The common use of the Façade pattern allows for unidirectional message passing for traditional (sequential) object-oriented systems. For distributed systems with concurrent threads of execution, bi-directional communication needs to be provided between two processes in a distributed environment. The technique therefore uses a facsimile of the Façade pattern, where bi-directional communication is enabled. It should also be noted that the technique allows a subsystem to provide more than one "Façade" or interface to the outside world. That is to say that a "Façade" in our case provides access to implementation classes based on the messages being passed, and is not an aggregation of all the functionality provided by the contained classes.

Figure 4.6 shows an example of two subsystems (client and server) that communicate with each other. The client subsystem consists of two client classes that communicate with the server class contained in the server subsystem. The Type classes that link the two Subsystems (C and D) are empty. They act as a buffer mechanism for connecting the subsystems. The operations are defined in the Type classes that are actually implemented by the Implementation Classes (Type A for instance).

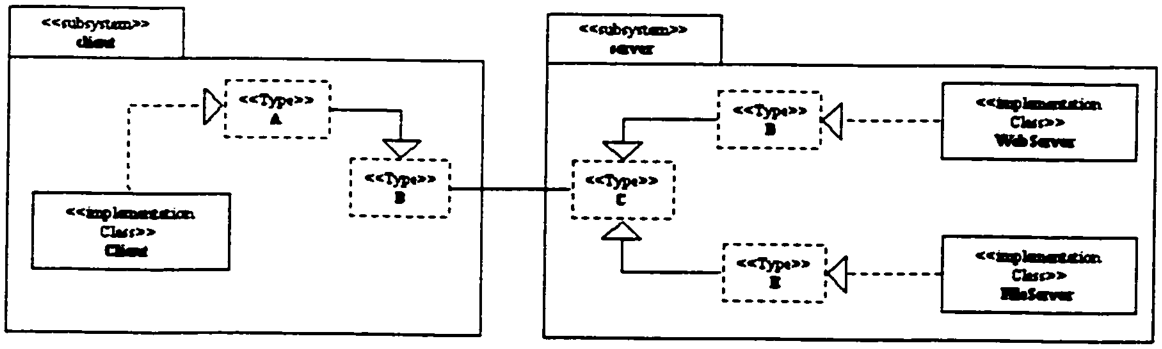


Figure 4.6: Client/Server class diagram.

In Figure 4.6, Type A inherits from type C. Since Type C has an association with type D, type A can communicate with any class derived from D. Therefore by implementing type A, Client1 can communicate with the server in the other subsystems. This relationship is an example of the Liskov Substitution Principle (LSP) which states that a child class can be substituted for a parent class in general [2].

4.2.1.4 Communication with the Environment

A class that has the "actor" stereotype applied to it represents an abstraction that lies just outside the system being modeled. An actor is defined to be outside the system, but may still be of interest when specifying how they interact with external agents. [7]

For the purposes of describing the architecture of a system using UML, it is necessary to show the interaction with the environment. In this case, the environment is loosely defined as anything external to the system in question. Figure 4.7 shows an actor called "Client" which communicates with the system "server".

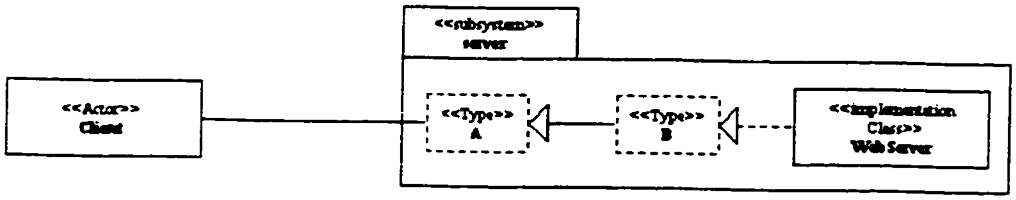


Figure 4.7: Communication with the environment.

Operations defined inside the actor class represent messages that can be sent to the environment by the system. Using actor classes allows for the *explicit* specification of allowable messages that can be exchanged between the environment and the system; this would not be possible without them.

4.2.2 Object Diagrams

Class diagrams describe the static relationships between objects and subsystems. They do not convey what the system will look like when it is running. The main problem is that multiple instances of the same class that interact with different objects are difficult to show in a class diagram. For this reason, Object Diagrams are used to show class instances explicitly. It is possible to show the entire system in one diagram, but this is not practical for large systems. To avoid overly large diagrams, Object Diagrams are used in two ways.

4.2.2.1 Internal Structure Diagrams

As previously described, a subsystem contains classes that communicate with each other or with other subsystems. In order to show exactly how many instances of each class are present within a subsystem, an object diagram for each subsystem is created. The internal structure diagrams for the client and server subsystems described in Figure 4.6 are shown in Figure 4.8.

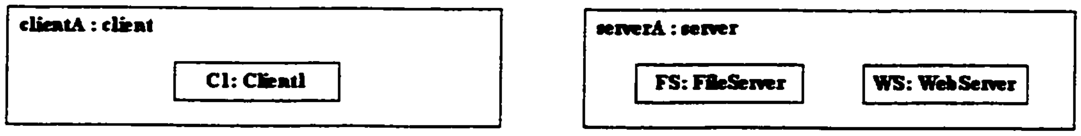


Figure 4.8: Internal structure of Client/Server example.

At runtime, the system will have one instance of the client and server subsystems. Both subsystems will have exactly one instance of their contained classes. The channels between subsystems are not shown in these diagrams. They will be shown at a higher level of abstraction as explained in the next section.

4.2.2.2 Interconnection Diagrams

Once the internal structures for all subsystem have been determined, the actual connection between subsystems must be shown. Only subsystems are shown in the interconnection diagrams; this results in straightforward diagrams that reflect the architecture of the system. It was mentioned in the previous section that links that go outside of a subsystem are omitted in the Internal Structure Diagrams. These links are shown in the Interconnection diagram. These diagrams show a more abstract view of the subsystems where the internal details of the subsystems are omitted. The Interconnection Diagram for the client/server model (Figure 4.6) is shown in Figure 4.9.

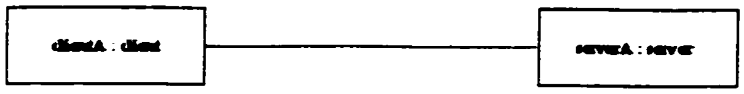


Figure 4.9: Subsystem interconnections for Client/Server example.

The link between the client and server could translate into one or many channels depending on the class diagram (Figure 4.6). It is known from the class diagrams through which type classes subsystems communicate, and hence the signals they send to each other. It is therefore unnecessary to show them in these diagrams.

4.3 From UML to SDL

In this section the mapping of a UML architecture into an equivalent SDL architecture is discussed.

4.3.1 SDL Channel Description

The precise specification of channels (and their signal lists) between communicating components was found to be the most difficult obstacle to overcome. We have considered three possibilities.

4.3.1.1 Type and Implementation Class Stereotypes

In a UML Class Diagram, a *type* stereotype can be used to show that a class has some behavior but that it cannot be instantiated. Another class that is stereotyped *implementationClass* must implement the Type class [7, 18].

The Type classes are used to show in more detail how the communication between objects happens. The Implementation Classes that implement the Type classes become Process Types in SDL. By using these two stereotypes to describe a system, one can tell at a glance which classes are instantiable and will become elements (Process Types) in a SDL specification.

The disadvantage of using these stereotypes is that they may not be widely known. They are defined in the UML specification, but are omitted in most basic UML books.

4.3.1.2 The Interface Stereotype

Communication between objects can be done through Interfaces in UML. The problem with interfaces is that in UML and SDL, they define exported operations available to other objects. This makes showing bi-directional communication difficult.

The convention in UML is to show a dependency arrow between an object and an interface [7]. This type of relationship is commonly understood as being unidirectional. In order to shown bi-directional communication, associations between Interface Classes could be drawn, but this is not the accepted way to use Interface Classes and would be confusing to most people familiar with UML.

4.3.1.3 Abstract Classes

Communication between two classes could be shown using abstract classes instead of using Interface classes. The advantage of this method is that it uses UML in a commonly understood way that achieves the same results as described in the preceding two sections.

The only difference between using Type and Implementation stereotypes with simply using abstract classes is the semantics. For these two cases, the class diagrams would be the same except for the stereotypes.

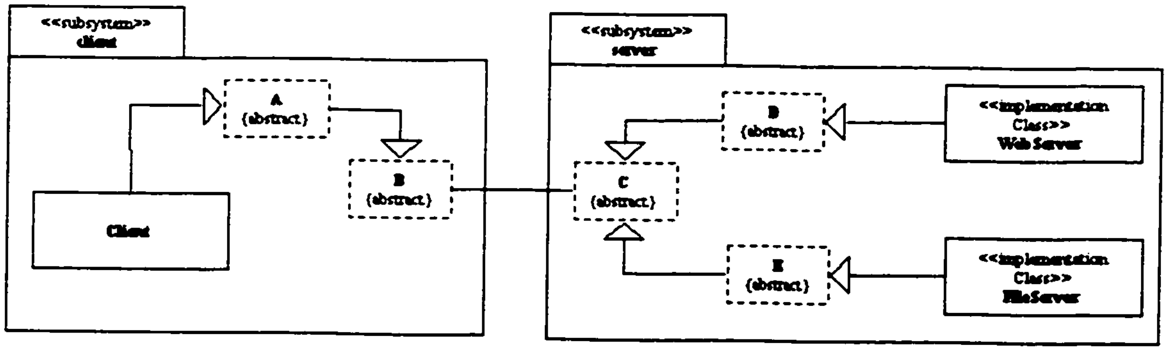


Figure 4.10: Using abstract classes.

Figure 4.10 shows the system defined in Figure 4.6 using the abstract constraint instead of Type and Implementation Class stereotypes. Note that this diagram could map very easily into two SDL Block Types that communicate through one channel. It is intended that C and D would become Gates in SDL.

The main difference between Type/Implementation Class and abstract classes is that Type and Implementation classes can only be related through a "realizes" relationship whereas abstract and concrete classes have no such limitation (they can be related with associations, aggregations, etc) [18].

Showing bi-directional communication using Interfaces goes against UML convention and therefore cannot be used in this way. One option is to use Interfaces to show unidirectional communication and thus use two unidirectional channels to describe one bi-directional channel. This idea will lead in practice to more complex diagrams. Figure 4.11 depicts the system in Figure 4.10 using Interface classes.

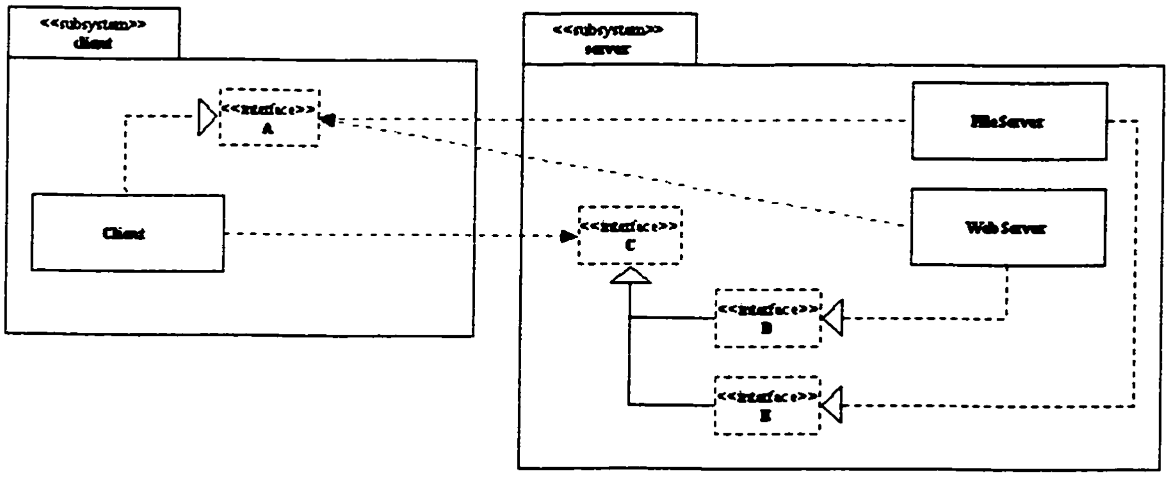


Figure 4.11: Client/Server system using unidirectional channels.

4.3.2 Translation of UML Architectures into SDL

The following sections describe the actual mappings that are used to translate a UML architecture into an equivalent SDL architecture.

4.3.2.1 Class Diagrams

UML Class Diagrams are used to show static relationships between elements in a system. The static Class Diagrams also show how components communicate with each other. Channels and the valid signals that can be sent over them are explicitly defined. The UML elements that are used and the way in which they are employed in the Class Diagrams are shown in the following subsections.

4.3.2.1.1 Subsystems

Since a subsystem is a self-contained element that has its own thread of execution, it maps nicely into a SDL Block Type. An example of the subsystem notation is shown in Figure 4.12.a. Figure 4.12.b shows the equivalent SDL representation.

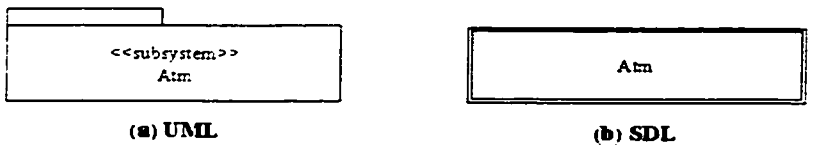


Figure 4.12: Instantiable subsystem.

Block types are the main building blocks used for describing distributed systems in SDL. Blocks communicate with each other through channels [3, 18]. We must therefore find a way to describe these channels in UML. The goal is to make subsystems as weakly coupled as possible to preserve the concepts of object-oriented design. Communication between components will be discussed in later sections.

4.3.2.1.2 Classes

In general, Blocks in SDL contains other Blocks or Processes [3, 4]. The subsystems described in the previous section will contain classes or other subsystems. Obviously, classes map well into Process Types in SDL.

We are interested in two UML stereotypes for describing the active components in a system and their interconnections. These are the *Implementation Class* and *Type* stereotypes.

4.3.2.1.2.1 Type Stereotype

A *type* stereotype applied to a class is used to specify operations applicable to a set of objects without defining the physical implementation of that class. These types of classes cannot be instantiated (much like an abstract class). All valid UML associations can be used between Type classes.

4.3.2.1.2.2 Implementation Class Stereotype

An Implementation Class may realize a number of different Types. An Implementation Class realizes a Type if it provides all of the operations defined for that Type. An object may have at most one Implementation Class. However, an object may conform to multiple different Types. As with Type classes, all valid UML associations can be used between Implementation Classes.

4.3.2.1.2.3 Implementation Class and Type Stereotypes

Implementation Classes and Type Classes can be used to explicitly define how objects communicate with each other, including the definition of a valid set of signals that can be sent over a specific channel.

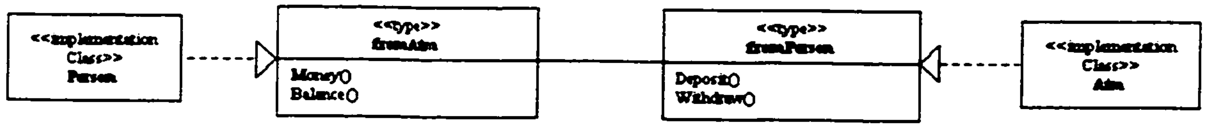


Figure 4.13: Communication between components.

Figure 4.13 shows two Implementation Classes, *Person* and *Atm*. The two type classes define the valid signals that each Implementation Class can receive. The *Atm* class implements the *fromPerson* Type and the *Person* class implements the *fromAtm* Type. The association between the two Type classes signifies a channel and allows the *Person* class to communicate with the *Atm* class. The operations defined in the Type classes define the set of valid signals that can be sent over the channel. For instance, a *Person* can send *Deposit* and *Withdraw* signals to the Atm class.

Using Type and Implementation classes in this manner allows for the unambiguous specification of channels between components. Suppose a vending machine class was added to the system shown in Figure 4.13. This new class addition is shown in Figure 4.14.

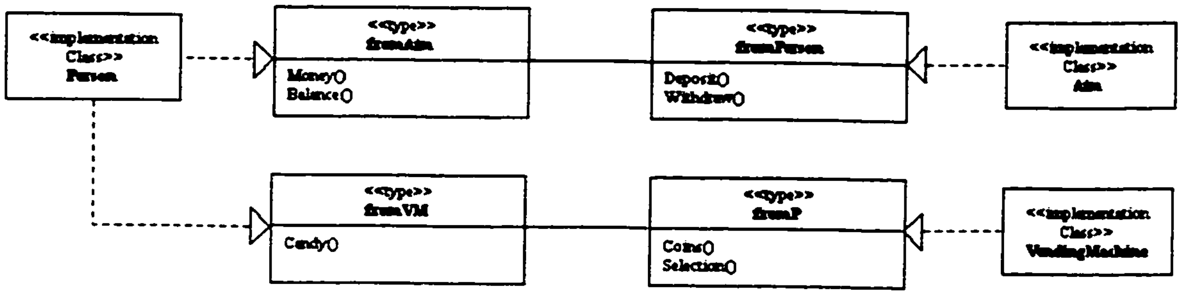


Figure 4.14: Adding a new class.

Only the signals *Candy, Coins* and *Selection* can be sent over the channel that connects *Person* with *VendingMachine*. It would be undesirable to allow the Atm class to send a Candy message to the Person class. This is a clear demonstration of why the choice to use the implementation class and type stereotypes was made.

Implementation Classes translate into SDL Process Types. The Types that a given Implementation Class realizes map into SDL Gates and the operations defined within these types become the signal lists for the gates. The equivalent SDL representations of the implementation classes shown in Figure 4.13 are depicted in Figure 4.15.

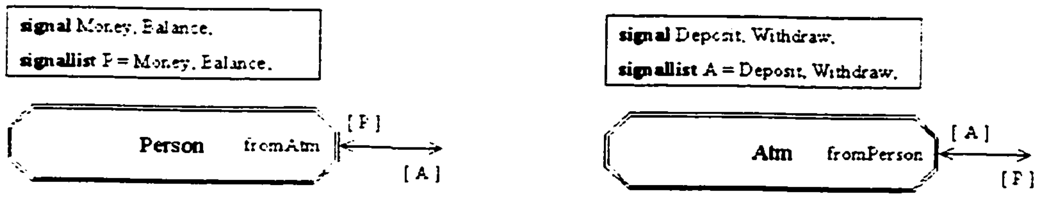

signal Money, Balance.
signallist F = Money, Balance.
Person
fromAtm
[F]
[A]
signal Deposit, Withdraw.
signallist A = Deposit, Withdraw.
Atm
fromPerson
[A]
[F]

Figure 4.15: Process types derived from UML.

It has been mentioned that channels connecting classes are shown by drawing associations between *Type* classes. The UML Class Diagrams have been used to define allowed connections between classes in a given subsystem, inter-subsystem communication is discussed in the next section.

4.3.2.1.3 Inter-Subsystem Communication

It has been shown how two classes can communicate by using *Type* classes. A mechanism for communicating between *subsystems* is also needed. By using *Type* classes the same way, channels can be explicitly defined between subsystems.

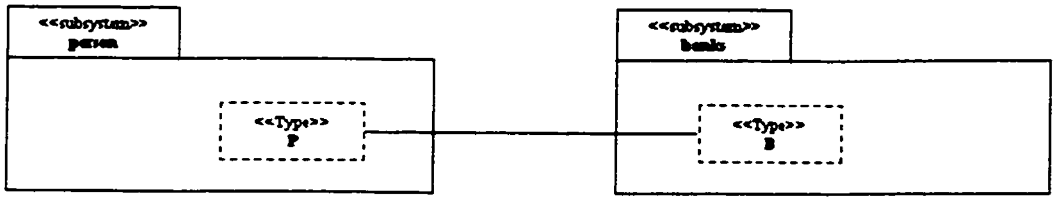


Figure 4.16: Communication between subsystems.

Figure 4.16 shows how paired *Type* classes can be used to define channels between subsystems. The *Type* classes become SDL Gates. The rest of the classes inside the subsystems are hidden for simplicity. Figure 4.17 shows the full system.

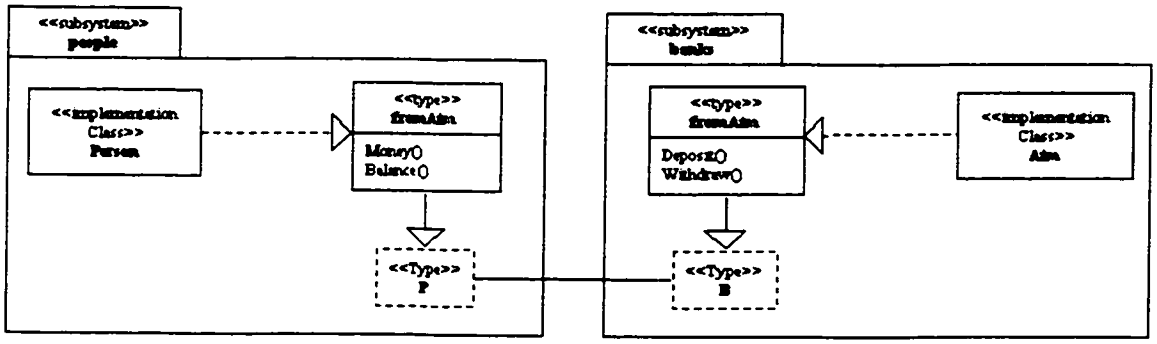


Figure 4.17: Classes communicating between subsystems.

In the subsystem *people*, *fromAtm* inherits from *P*, which does not have, any signals defined of its own. The type P will become an SDL Gate, and the signal lists will be the union of all signals defined in the Type classes that inherit from it. The subsystems will map into SDL Block Types, which are shown in Figure 4.18.

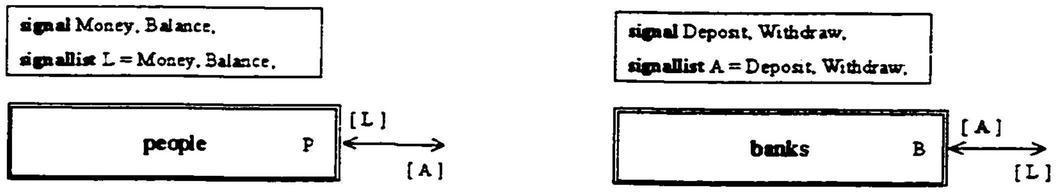

signal Money, Balance,
signallist L = Money, Balance,
people
P
[L]
[A]
signal Deposit, Withdraw,
signallist A = Deposit, Withdraw,
banks
B
[A]
[L]

Figure 4.18: Block types derived from UML.

4.3.2.2 Additional Concepts

So far, only basic relationships between classes have been addressed. There are a few specialized cases that need to be looked at. These are discussed in the following subsections.

4.3.2.2.1 Inheritance

The concept of inheritance is present in both UML and SDL. For the purposes of translating from UML to SDL, only classes with the Implementation Class stereotypes are allowed to have inheritance relationships. Multiple inheritance is not allowed. An example of inheritance is shown in Figure 4.19.

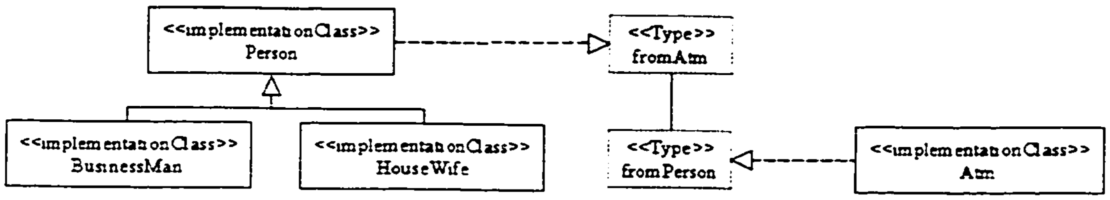


Figure 4.19: UML inheritance.

The *BusinessMan* and *HouseWife* classes inherit from the *Person* class. As a result, both can communicate with the *Atm* class.

Figure 4.20 shows the SDL representation of the inheritance relationship between the *Person*, *BusinessMan* and *HouseWife* classes shown in Figure 4.19.

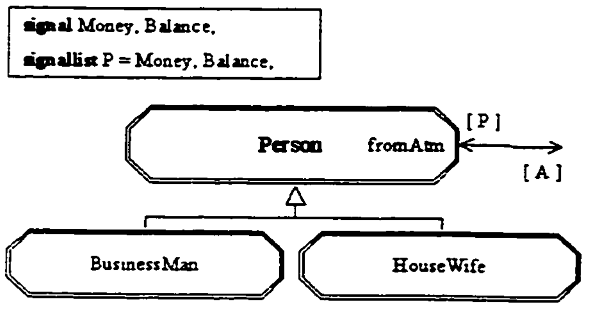


Figure 4.20: SDL inheritance.

4.3.2.2.2 Abstract Classes

Abstract classes and packages in UML are mapped to abstract process and block types respectively. Abstract components cannot be instantiated and must be specialized by other components. This is accomplished by using inheritance where sub-classes will inherit from an abstract class.

4.3.2.2.3 Regular Class

A normal class that does not have any stereotypes applied to it becomes a SDL Newtype that is similar to a "struct" in C. Figure 4.21 shows how a class is translated into a SDL Newtype.

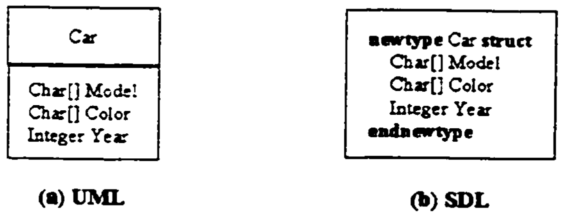


Figure 4.21: SDL newtype.

4.3.2.2.4 Interface Class

An interface in UML describes the externally visible and accessible behaviors of components. Similarly, an interface in SDL lists the externally visible set of stimuli (input signals, procedures and variables). An interface with an agent's name is implicit. Therefore, UML interface classes can be easily mapped to SDL interfaces.

4.3.2.3 Object Diagrams

Object Diagrams are used to show class instances explicitly. It is possible to show the entire system in one diagram, but this is not practical for large systems. To avoid overly large diagrams, Object Diagrams are used in two ways.

4.3.2.3.1 Showing the Internal Structure of Subsystems

A subsystem contains classes that communicate with each other or with other subsystems as previously described. In order to show exactly how many instances of each class there are, an object diagram is used for each subsystem. An example of Object Diagrams for the subsystems in Figure 4.17 is shown in Figure 4.22.

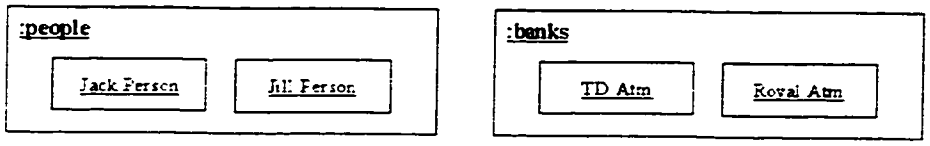


Figure 4.22: Subsystem object diagrams.

The purpose of these diagrams is to show how a subsystem will initially look when it is first instantiated. For example, the *people* subsystem will have two instances of *Person*, namely Jack and Jill every time a new instance of *people* is created. Communication links between components within a subsystem are shown in these diagrams, but inter-subsystem communication links are omitted. These will be shown in another Object Diagram.

The equivalent SDL representations of the object diagrams in Figure 4.22 are shown in Figure 4.22 and Figure 4.23.

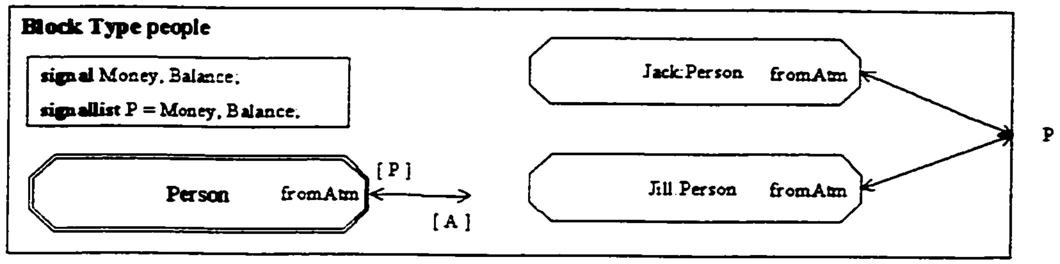


Figure 4.23: Block Type *People.*

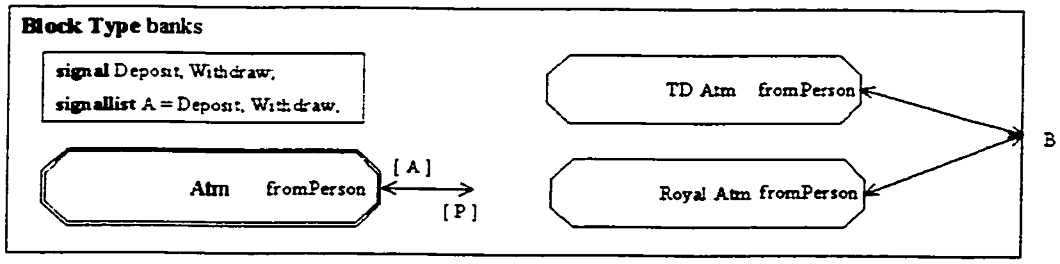


Figure 4.24: Block Type *Banks.*

The SDL types (double bordered components) in the diagrams represent instantiable elements similar to classes. The unconnected arrows represent the connection points of the types where channels can be connected upon instantiation. The square brackets "[ ]" enclose the allowable signals to and from a connection point of a type. The instances (simple bordered components) show these connections. If we look at Figure 4.24, TD:Atm is an instance named "TD" of the type "Atm" (same convention as UML).

4.3.2.3.2 Showing Inter-Subsystem Communication

The previous section dealt with specifying what the internal architecture of subsystems look like. The next step is to show the instances of the subsystems and how they are linked together. This is accomplished by using Object Diagrams where the objects are the subsystems and the internal structures are omitted. Using the subsystems described in Figure 4.17, an Object Diagram showing how actual instances of the subsystems can be interconnected is shown in Figure 4.25.

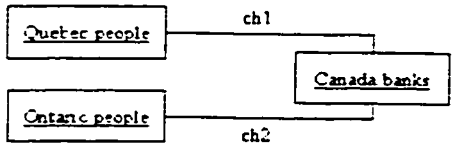


Figure 4.25: Interconnecting subsystem instances.

The links can be given names that will translate into channel names in SDL. The equivalent SDL representation of Figure 4.25 is shown in Figure 4.26.

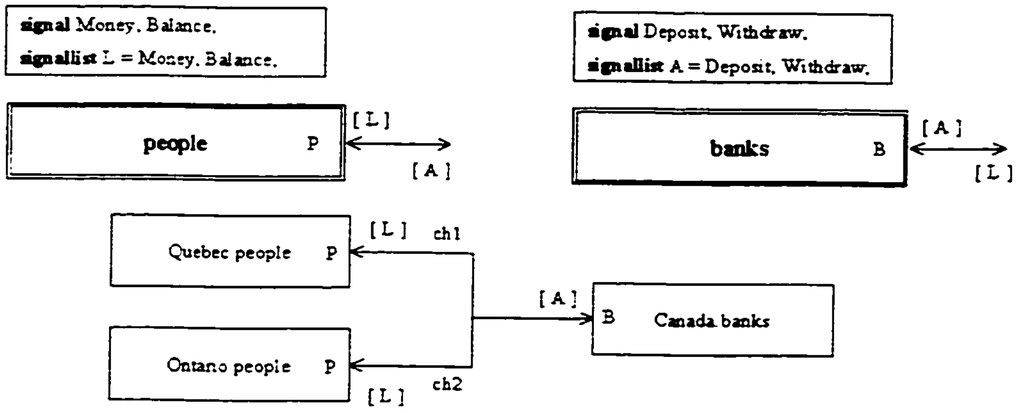


Figure 4.26: SDL system representation.

4.3.2.4 Summary of Translations

Table 1 shows how different UML components are mapped into SDL components.

UML	SDL
Subsystem	Block Type
Implementation Class	Process Type
Abstract Subsystem	Abstract Block Type
Abstract Implementation Type	Abstract Process Type
Class	New Type
Type	Gate
Associations Between Type Classes	Channels
Operations described by Type classes	Signal List
Interface	Interface
Inheritance between Implementation Classes	Inheritance between Process Types
Inheritance between Subsystems	Inheritance between Block Types

Table 1: UML to SDL mapping rules.

4.4 Related Work

The Z.109 [28] recommendation imports SDL into UML by making use of the extension mechanisms available in UML. Thus the "SDL UML" language that is described in Z.109 is a specialized subset of UML. An advantage of the use of the UML extension mechanisms is that all SDL concepts can be imported into UML. This is good for designers that are already familiar with SDL. Someone unfamiliar with SDL would have to learn it in order to use the Z.109 specification. This means that there is a relatively steep learning curve for this method. It should also be noted that the Z.109 is not part of the OMG Specification of UML (v1.3) but is a customized version of UML for the purpose of using SDL and UML together. Telelogic is developing tools that support translation of UML into SDL based on the Z.109 recommendation [27].

On the other hand ObjecTime has introduced a methodology for describing real time systems using UML [30, 31]. In this approach, concepts from ROOM [33] are introduced into UML via stereotypes. There is no attempt to map UML into SDL, but it is straightforward to map the ROOM-UML elements into SDL. The approach has been extended into mapping SDL into UML [34]. The main reason behind this mapping is that UML is more popular than SDL and consequently it is better to use UML for design, simulation and verification. However, it should be noted that UML does not have formal semantics yet and tools for simulation and validation are not available yet.

The Integrated Method or TIMe [35] uses UML for object modeling and MSC for showing the interactions between objects. SDL is used to fully specify the system in terms of architecture and behavior. The TIMe method has loose mapping rules for going from UML to SDL that are based on the Z.109 recommendation. As such, TIMe method requires that designers manually do the translations. This can lead to errors and inconsistencies. The approach discussed in this thesis automates the translation process and is combined with existing tools [14, 15, 19] for automatically translating MSCs into SDL extended finite state machines.

In [36], Verschaeve and Ek present three scenarios for combining SDL and UML. They defined a mapping between the two notations including the behavior models. The mapping of basic structures such as subsystem and class are similar to our method. The main differences are in the specification of signals and inter-subsystem communication and the usage of object diagrams. Another difference is in the behavioral specification, where our technique uses MSC to describe the behavioral aspect of a system, Verschaeve and Ek use the UML state diagrams.

4.5 Automatic Teller Machine Example

Figure 4.27 shows a description of a system in UML. The system is composed of one subsystem. The subsystem contains three Implementation Classes, *User*, *Atm* and *Banks*. These are connected through Type classes in the same manner as discussed previously. Figure 4.28 is an object diagram of the *ATM_Block* subsystem. Figure 4.29 shows the SDL representation of the system with one instance of the *ATM_Block* block type. Figure 4.30 shows the internal specification of *ATM_Block* that is derived from Figure 4.27 and Figure 4.28.

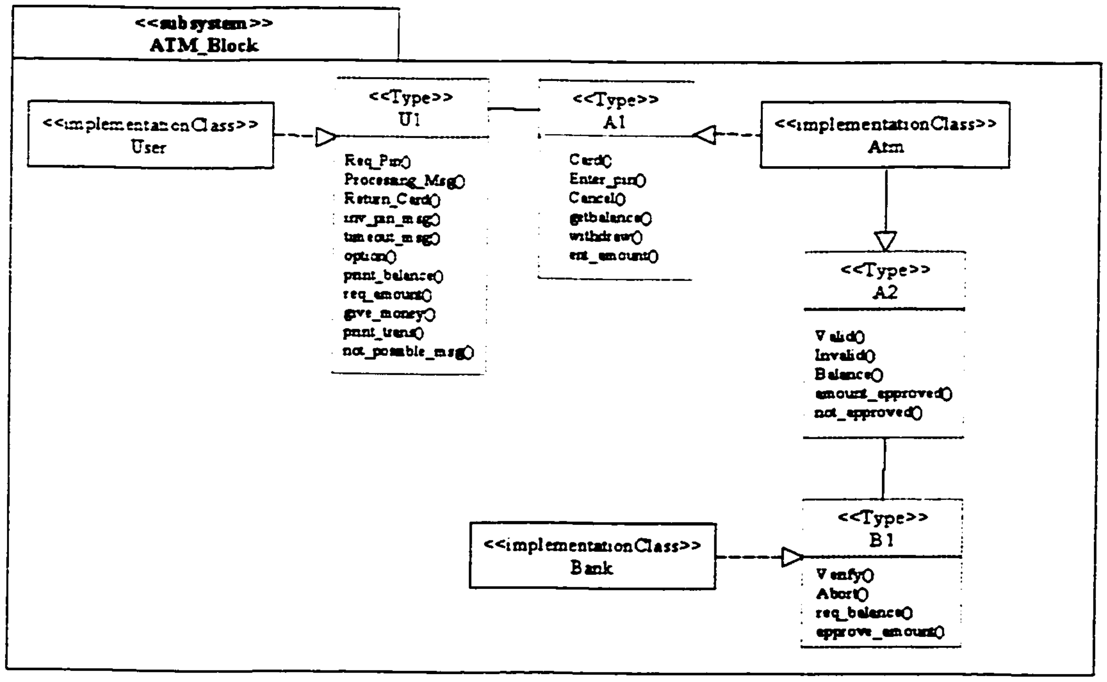


Figure 4.27: *ATM_Block* Subsystem.

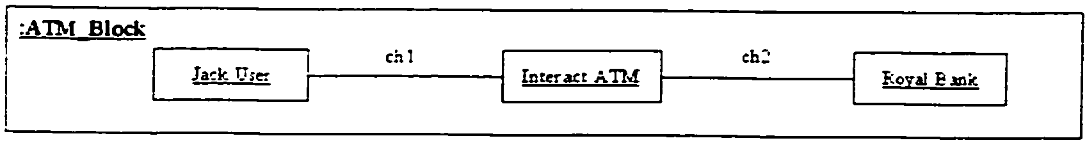


Figure 4.28: *ATM_Block* Object Diagram.

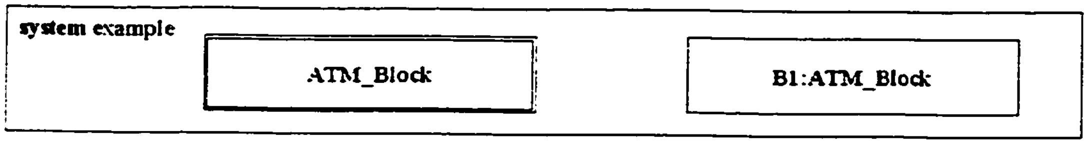
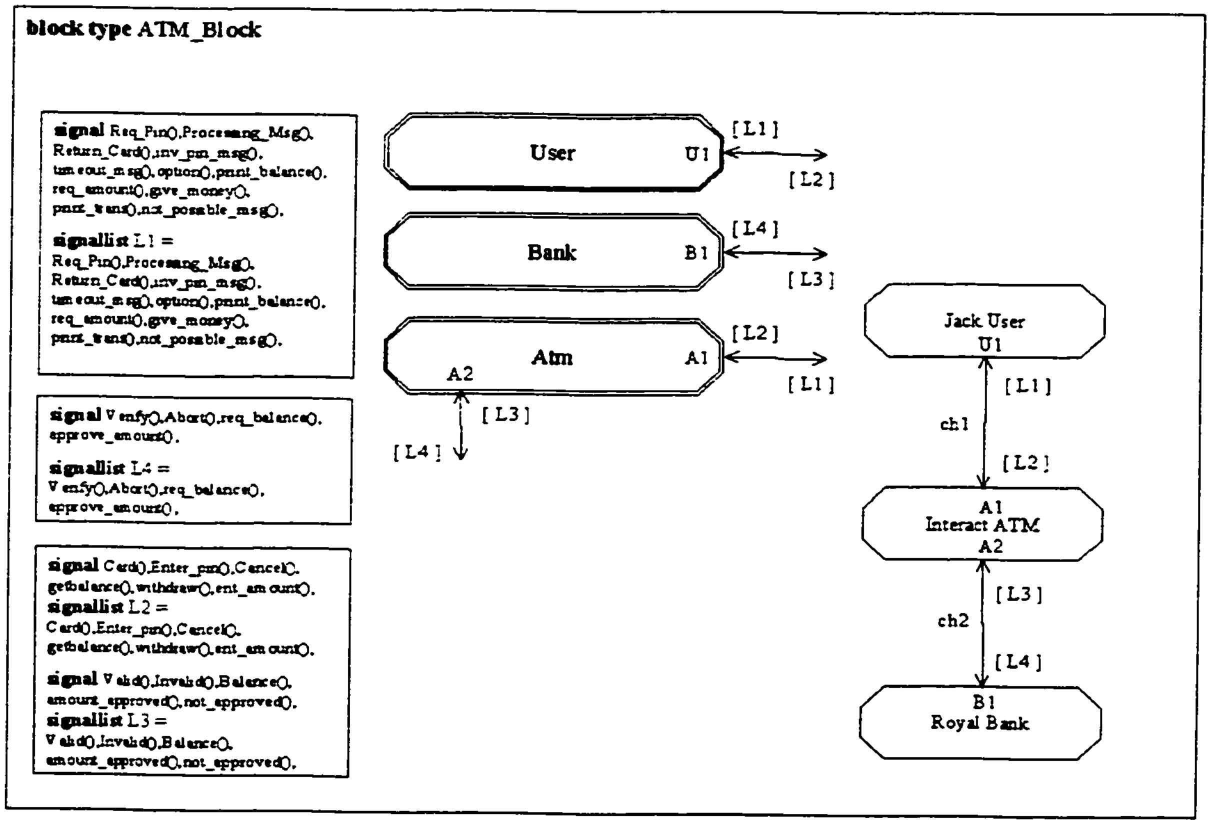


Figure 4.29: SDL Representation.



block type ATM_Block
signal Req_Pin(),Processing_Msg(), Return_Card(),inv_pin_msg(), timeout_msg(),option(),print_balance(), req_amount(),give_money(), print_trans(),not_possible_msg().
signallist L1 = Req_Pin(),Processing_Msg(), Return_Card(),inv_pin_msg(), timeout_msg(),option(),print_balance(), req_amount(),give_money(), print_trans(),not_possible_msg().
signal Verify(),Abort(),req_balance(), approve_amount().
signallist L4 = Verify(),Abort(),req_balance(), approve_amount().
signal Card(),Enter_pin(),Cancel(), getbalance(),withdraw(),ent_amount().
signallist L2 = Card(),Enter_pin(),Cancel(), getbalance(),withdraw(),ent_amount().
signal Valid(),Invalid(),Balance(), amount_approved(),not_approved().
signallist L3 = Valid(),Invalid(),Balance(), amount_approved(),not_approved().
User
U1
[L1]
[L2]
Bank
B1
[L4]
[L3]
Atm
A1
A2
[L2]
[L1]
[L3]
[L4]
Jack User
U1
[L1]
ch1
[L2]
A1
Interact ATM
A2
[L3]
ch2
[L4]
B1
Royal Bank

Figure 4.30: *ATM_Block* SDL Specification.

4.6 Inres Protocol Example

This section shows another architecture that models the Inres protocol. Figure 4.32 shows the class diagram of the system. The environment can communicate with the *inres* subsystem, and the *inres* subsystem communicates with the *medium* subsystem. The implementation classes are Inres, Coder and Medium, the rest are type classes, except for the Environment class which is an actor class.

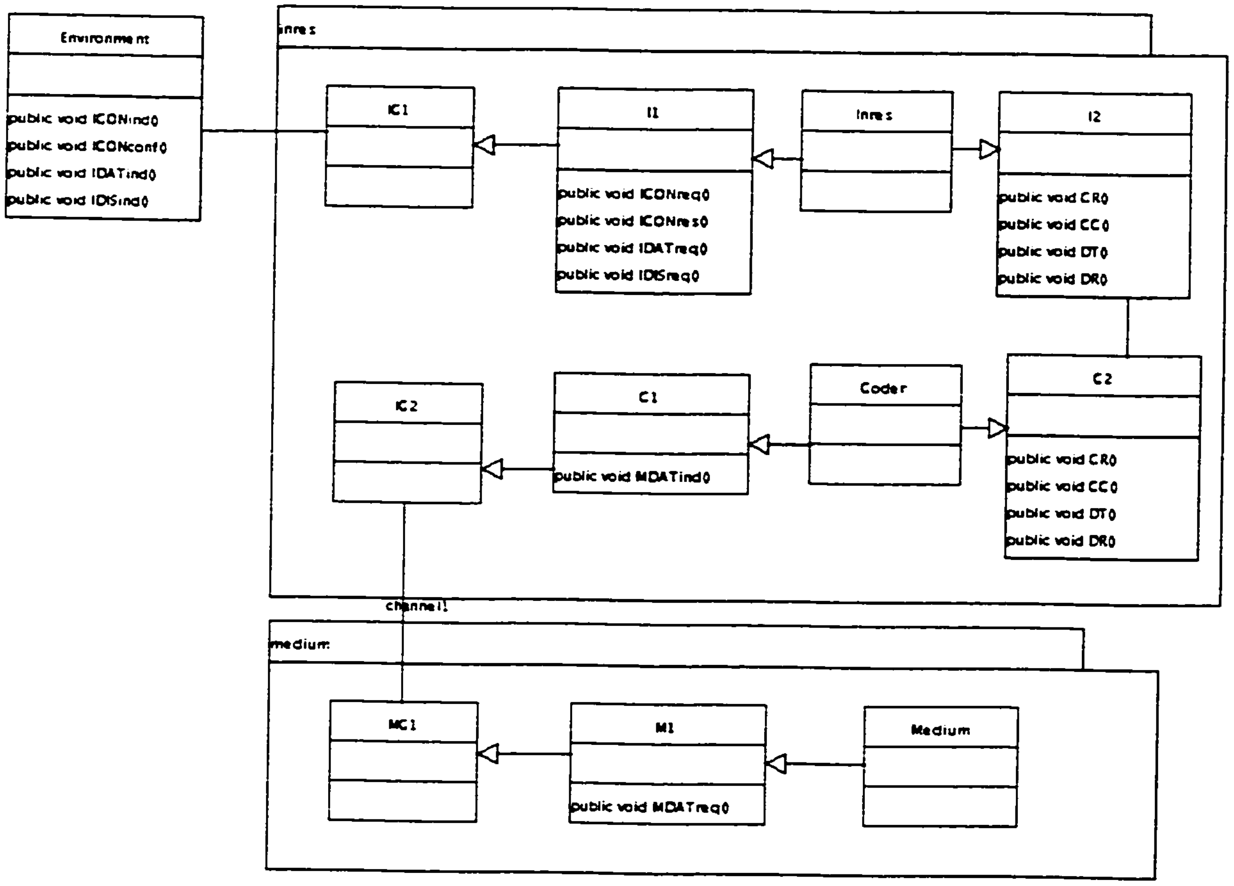


Figure 4.31: Inres Protocol Class Diagram.

Figure 4.32 shows the top-level object diagram for the system. Two instances of the *inres* subsystem communicate with each other through one instance of the *medium* subsystem.

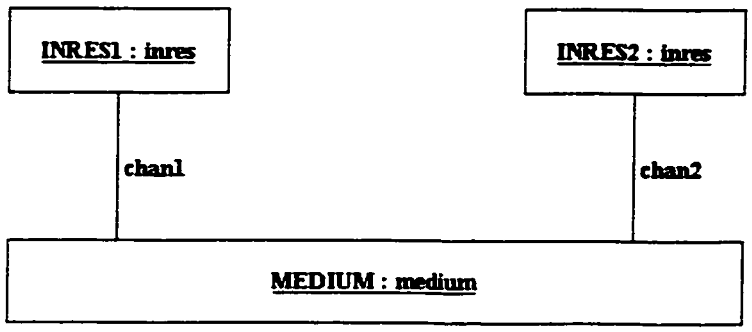


Figure 4.32: Inres protocol top-level object diagram.

The object diagrams which show the internal structure of the *inres* and *medium* subsystems are shown in Figure 4.33.

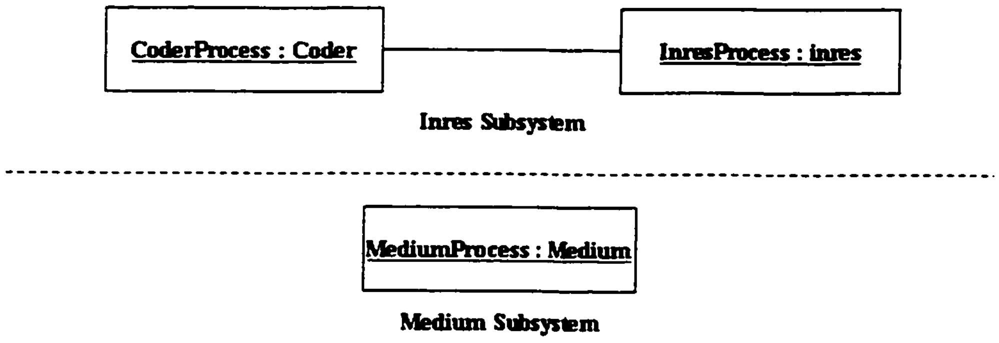


Figure 4.33: Object diagrams for the *Inres* and *Medium* subsystems.

Figure 4.34, Figure 4.35 and Figure 4.36 show the equivalent SDL architecture obtained by applying the UML-to-SDL mapping rules.

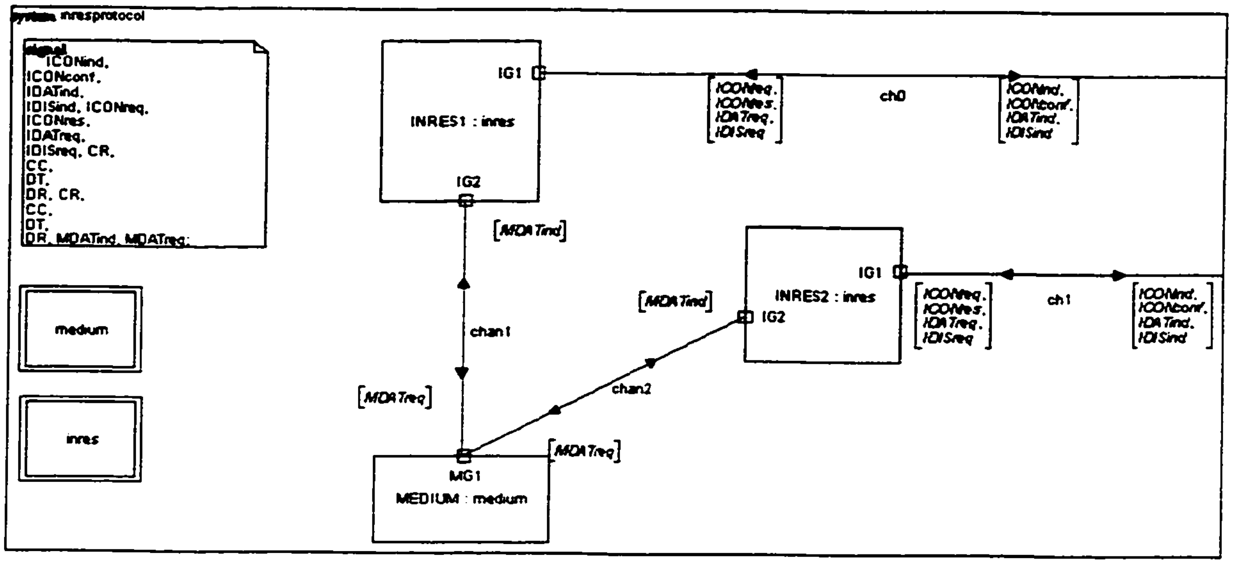

system inresprotocol
signal ICONind, ICONconf, IDATind, IDISind, ICONreq, ICONres, IDATreq, IDISreq, CR, CC, DT, DR, CR, CC, DT, DR, MDATind, MDATreq;
medium
inres
INRES1 : inres
IG1
IG2
ch0
ICONreq, ICONres, IDATreq, IDISreq
ICONind, ICONconf, IDATind, IDISind
[MDATind]
chan1
[MDATreq]
chan2
INRES2 : inres
ch1
MG1
MEDIUM : medium

Figure 4.34: Inres protocol system architecture in SDL.

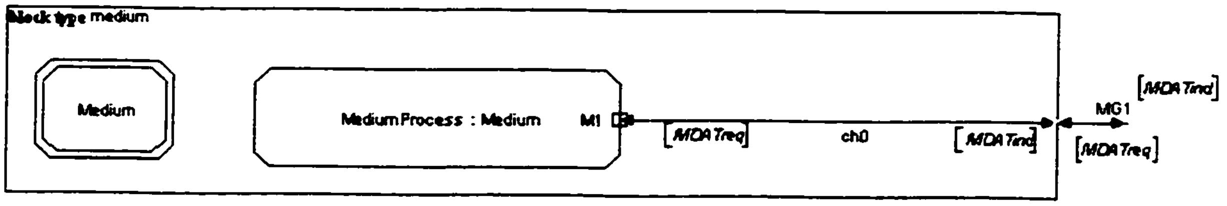


Figure 4.35: *Medium* subsystem specification in SDL.

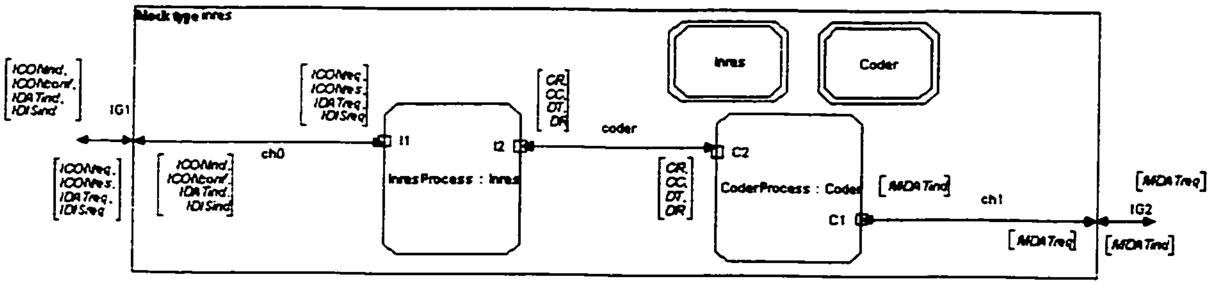


Figure 4.36: *Inres* subsystem specification in SDL.

Chapter 5

Tool and Application

This chapter gives an overview of the UML to SDL and MSC refinement tools. The high level architecture of each is described, and then the limitations of each tool are discussed. The chapter concludes with an illustration of the use of both methodologies together.

5.1 Overview

As previously stated, two methodologies have been developed with the intention of using them in conjunction to aid in the design of distributed systems from requirements to design. The UML to SDL methodology was implemented in Java [37]. The MSC Use Case methodology was then implemented and added to the tool. The structure of the two combined implementations is shown in Figure 5.1. The *gui* package is the front-end graphical user interface that allows both tools to be accessed.

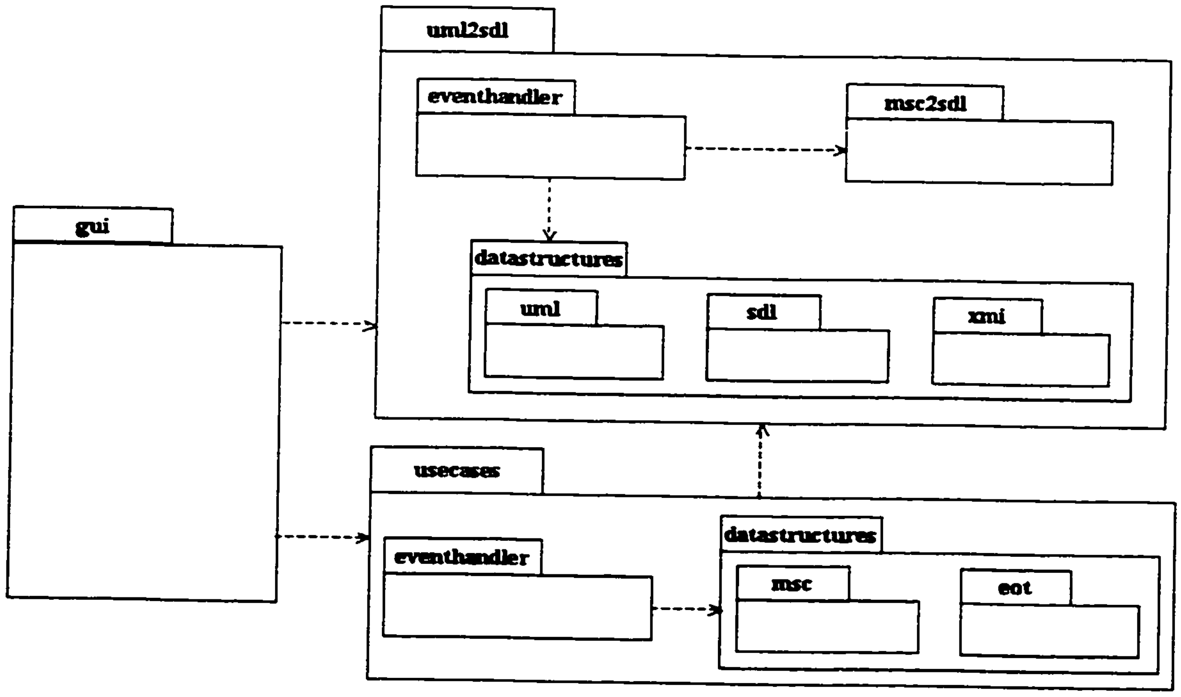


Figure 5.1: Tool Class Diagram

5.2 UML to SDL Translation

To create UML models, ArgoUML[38] was used. ArgoUML is a freely available UML editor that saves the model information in XML Metadata Exchange (XMI) [39] format, which is text-based and can easily be parsed and manipulated. The XMI file is parsed by the UML to SDL tool and a UML data structure is created. The UML data structure can then be translated into an SDL architecture and outputted in a format recognized by ObjectGeode. The SDL architecture can then be viewed and edited in ObjectGeode.

The *uml2sdl* package in Figure 5.1 represents the implementation of the UML-to-SDL algorithm. The major components of the tool are:

- *eventhandler:* Processes events generated by the user and performs functions such as reading in XMI files and parsing them into a UML data structures. It will then perform the translation from UML to SDL and create an SDL data model, which is combined with the MSC input file by the MSC-to-SDL tool.

- *Uml:* Contains classes that represent UML elements. This package allows a UML data structure to be created by the *eventhandler* package.

- *Sdl:* Contains classes that represent SDL elements. This package allows a SDL data structure to be created by the *eventhandler* package.

- *Xmi:* Contains all the keywords used in the XMI input file. This package allows the XMI input file to be parsed so that the *evenhandler* package can create a UML data structure.

- *Msc2sdl:* The MSC2SDL tool is a C program that is accessed using the Java Native Interface (JNI). This package contains mechanisms that allow the java program to access the C program to combine the MSC behavioral specifications with the SDL architecture to generate a full SDL specification.

As previously mentioned, ArgoUML 0.7 is used to create the UML models. Version 0.8.1a is currently available. The newer version is much better, but it is not backwards compatible. The developers of ArgoUML changed the format of the XMI output by the tool, and as a result, version 0.7 *must* be used to create the models. To upgrade to the newer version, the algorithm that parses the XMI files will need to be modified.

A bug that is somewhat problematic was discovered while testing the UML-to-SDL tool. When ArgoUML 0.7 loads a previously saved model, some information is lost. It seems that for some reason, the links in object diagrams are set to have null endpoints, instead of referencing objects. If the model is then saved and used as input to the UML-to-SDL tool, an error will occur. This error has been fixed in the current version of ArgoUML.

5.3 Stepwise Refinement of MSCs

Figure 1.2 (shown on page 5) shows how the MSC refinement tool is combined with existing tools. The tool can guide the user through the refinement process and verify that the resulting MSCs conform to the original use cases and follow the system architecture at every step.

The *usecases* package in Figure 5.1 shows the structure of the tool. The major components of the tool are:

- *eventhandler:* Processes events generated by the user and performs functions such as reading in the MSC files and constructing event order tables to check if the conformance relation holds for a set of MSCs.

- *msc:* Contains classes that represent MSC elements. The *eventhandler* package uses these classes to create msc data structures.

- *eot:* Contains classes that allow the *eventhandler* package to use the MSC data structures to create event order tables.

The MSC refinement methodology is guided by an architecture specified in UML. By using the *uml2sdl* package, the tool can read a UML model from disk and then translate it into SDL via the *msc2sdl* package. The resulting SDL architecture is then used to guide the refinement process.

The MSC refinement tool must be used in conjunction with a MSC editor. In our case, ObjectGeode is used to edit the MSCs. This requires that for every refinement, the user must do the following:

i. Load the MSC into ObjectGeode.

ii. Refine the MSC.

iii. Save the MSC.

iv. Use the MSC refinement tool to check the new MSC against the old one.

This process is not very user-friendly. The ideal situation would be to have the refinement algorithm and editing capabilities integrated into a single tool.

Another issue is the fact that the MSC-to-SDL tool requires that global conditions be included in the design MSCs to properly translate them into SDL. As a result, the designer must add global conditions to all the MSCs at the end of refinement process. The ideal scenario is for the use of conditions to be optional.

5.4 SIP Example

In this section, the generation of an SDL specification from a UML architecture and MSC use cases is illustrated. The example starts with the description of the different scenarios that can arise during a telephone call established using the Session Initiation Protocol (SIP). The second step is to specify the architecture of the system using UML, and translating it into SDL via the UML-to-SDL algorithm. The third step is to refine the use cases into design MSCs using the refinement methodology. Finally, the MSC-to-SDL tool generates the SDL behavioral specifications by using the design MSCs and SDL architecture as input.

5.4.1 MSC Use Cases

The Session Initiation Protocol (SIP) is a text-based signaling protocol used to create and control multimedia sessions between two or more parties over IP based networks [40].

The use cases presented in this section were derived from the SIP call flows presented in [41]. The call flows presented model two scenarios:

i. One user of the system calls another, and after they are finished talking, the called party hangs up.

ii. The user tries to complete a call, but gets a busy signal and hangs up.

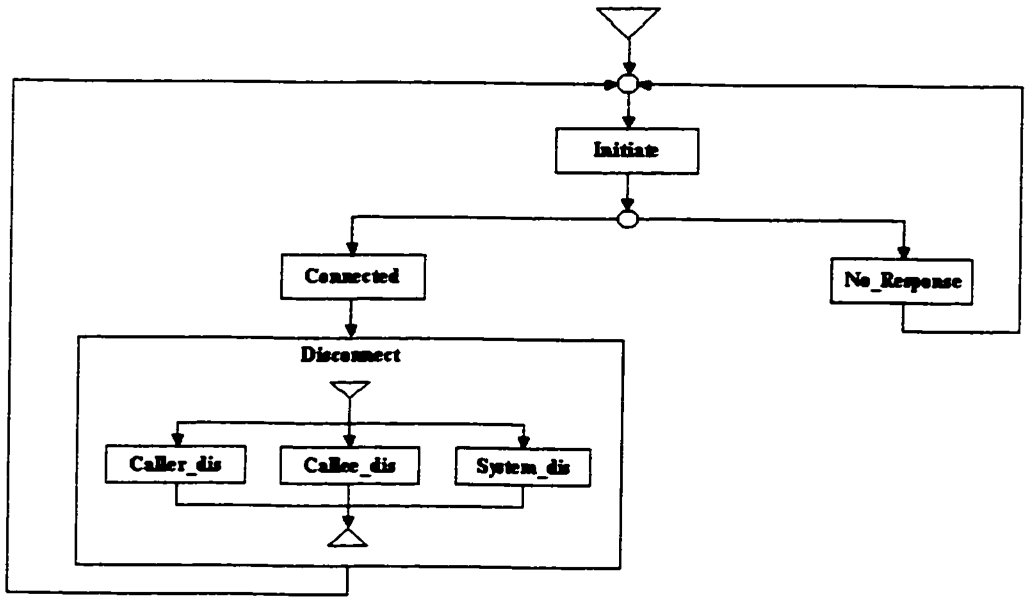


Figure 5.2: Telephone call using SIP.

Figure 5.2 shows the different MSCs that make up the system. The first action is to initiate the telephone call by sending the appropriate messages across the network to the destination, after which the call will either be completed successfully, or unsuccessfully if the called party is busy. If the call was properly established, the called party will eventually terminate the call. The MSC use cases that depict this behavior graphically are shown in Figure 5.3.

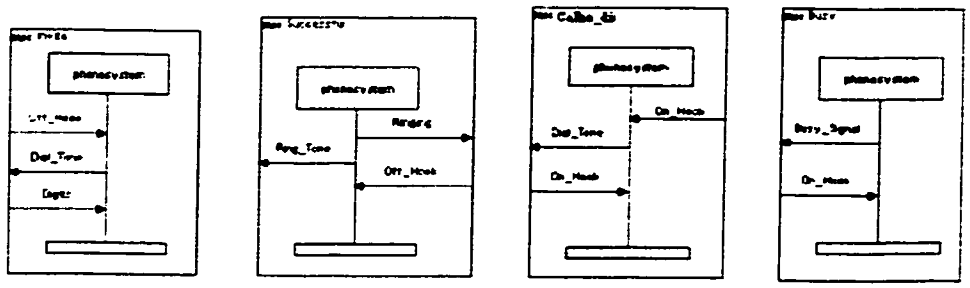


Figure 5.3: Use cases for SIP call model.

5.4.2 Architecture Specification

The static relationships between classes are shown in Figure 5.4. The actor class *User* can communicate with the *telephonesystem* subsystem via the *Receiver* type class. The telephonesystem encapsulates two subsystems:

i. *Pots:* Represents the conventional telephone system. Contains the implementation class *Pbx*, which models a conventional switch.

ii. *Sip:* Represents an IP based network that offers SIP capabilities. Contains the implementation classes *Gateway* and *Proxy*, which model a SIP gateway and proxy respectively.

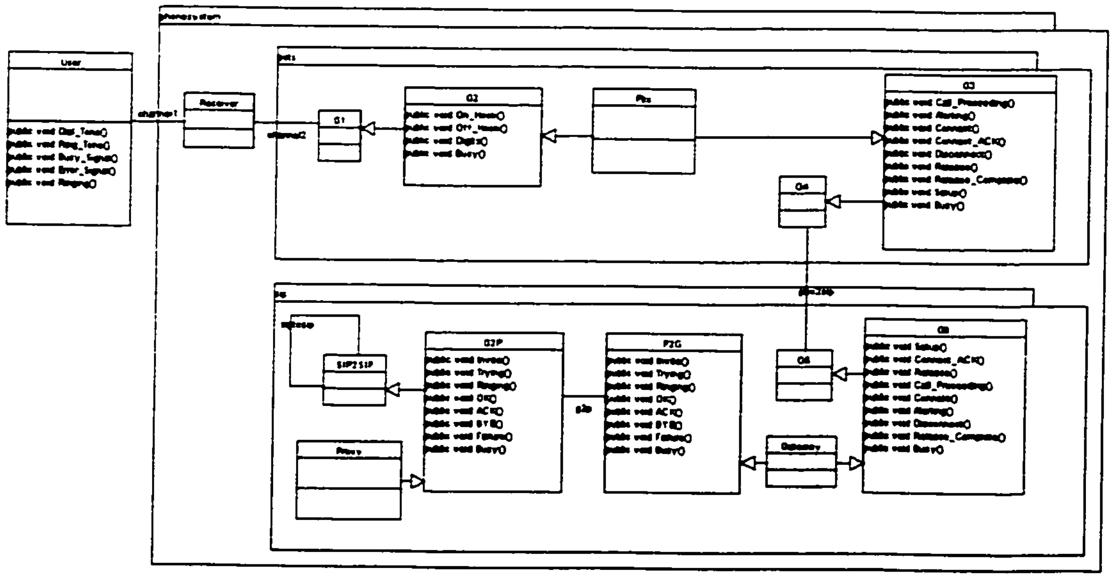


Figure 5.4: Class diagram for the telephone system.

The object diagrams depicting the dynamic system architecture is shown in Figure 5.5:

i. The top-level system object diagram.

ii. Shows the configuration of the *phonesystem* subsystem. Note that there are two instances of the *pots* and *sip* subsystems. The diagram shows clearly that POTS1 communicates with POTS2 via SIP1 and SIP2.

iii. Shows the *pots* and *sip* subsystem respectively.

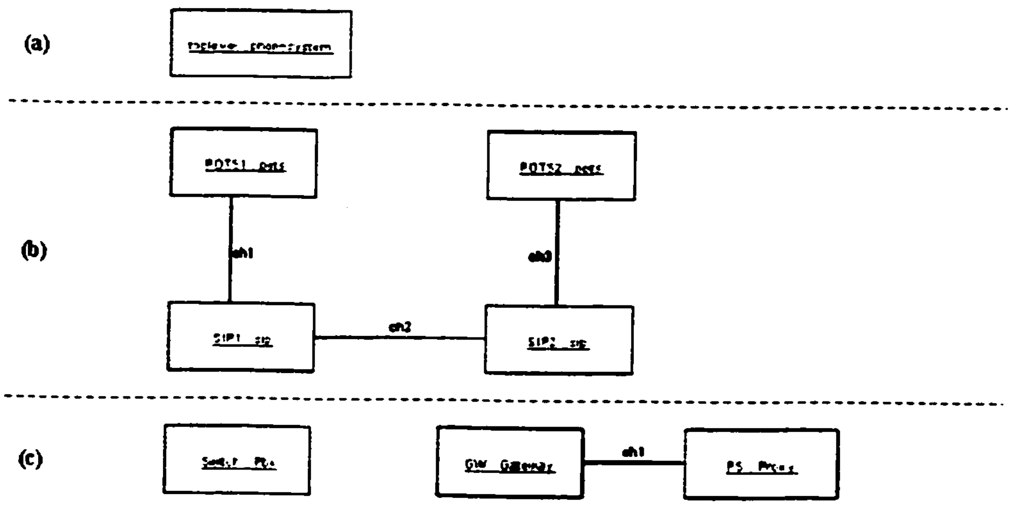


Figure 5.5: Subsystem object diagrams.

Figure 5.6, Figure 5.7, Figure 5.8 and Figure 5.9 show the corresponding SDL architecture obtained from the UML-to-SDL tool.

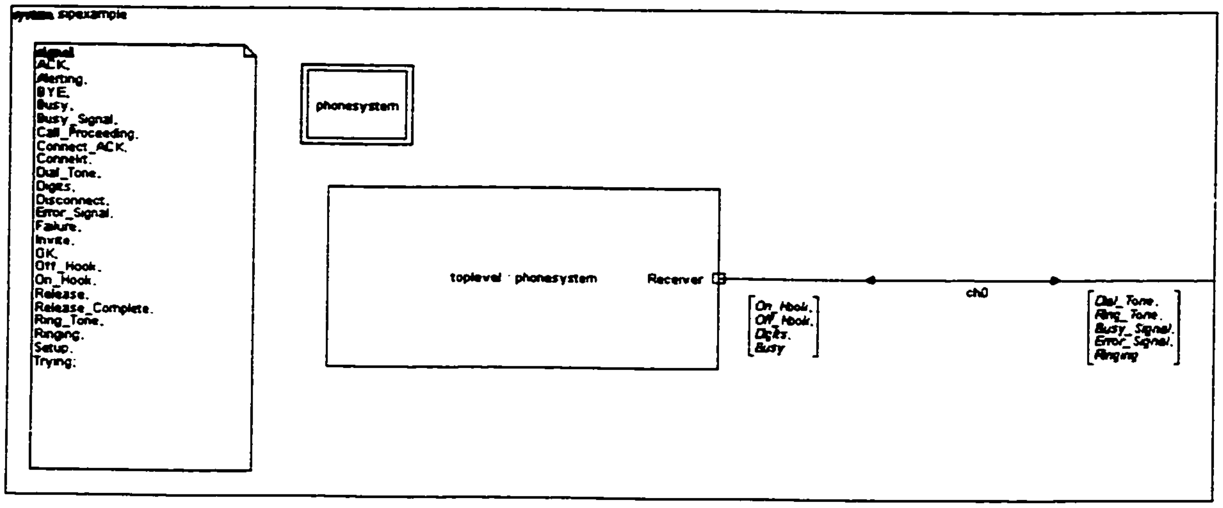


Figure 5.6: *sipexample* SDL system specification.

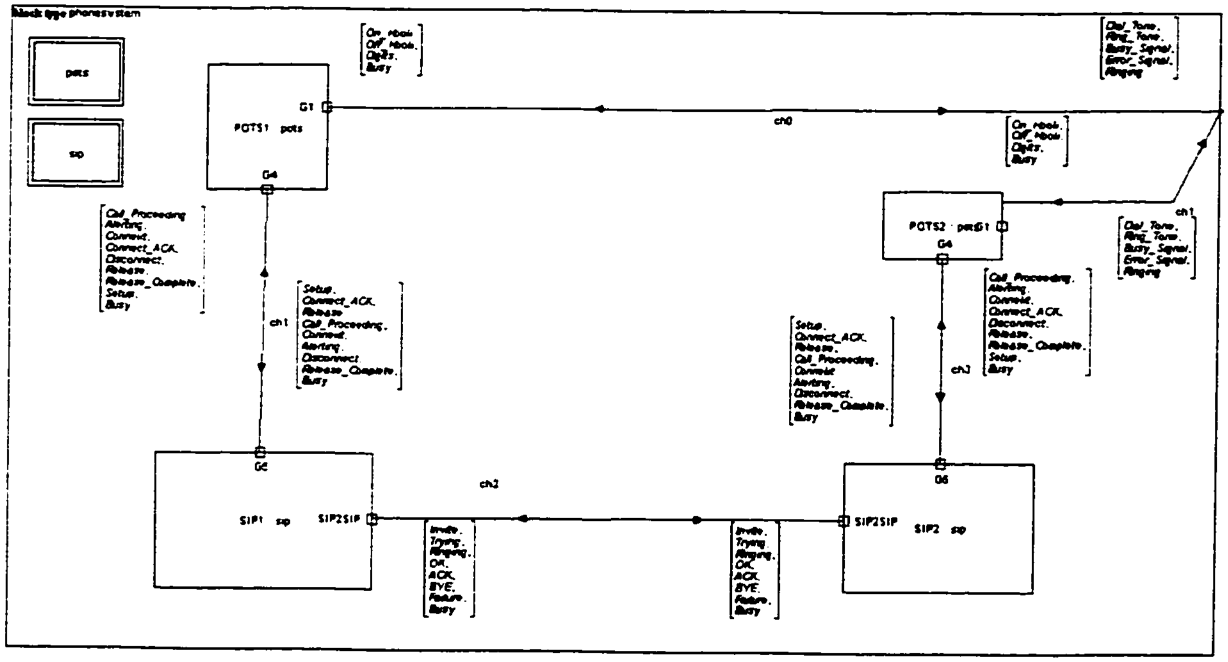


Figure 5.7: *phonesystem* block type specification.

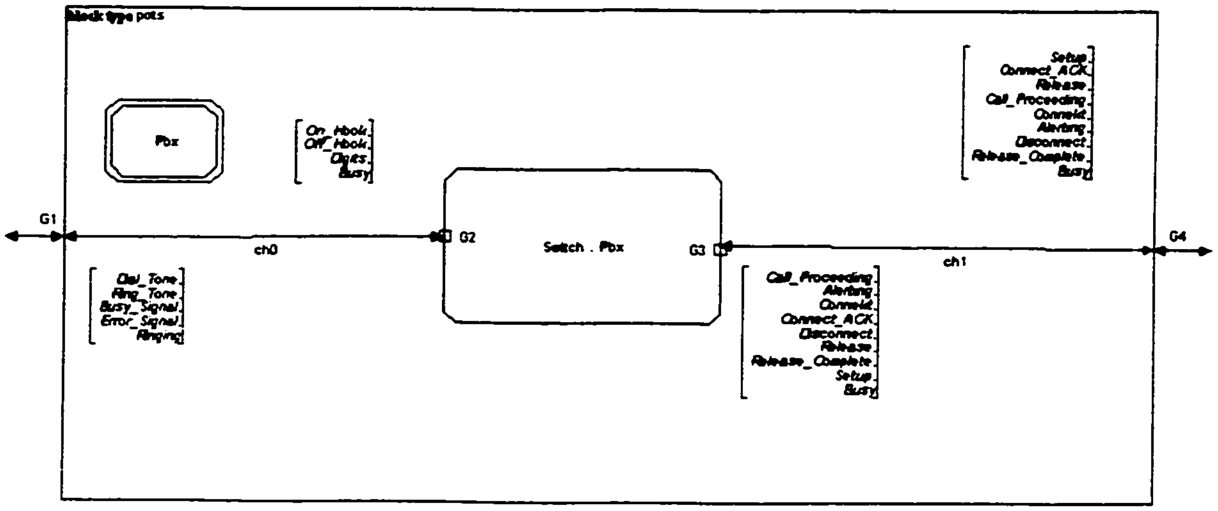


Figure 5.8: *pots* block type specification.

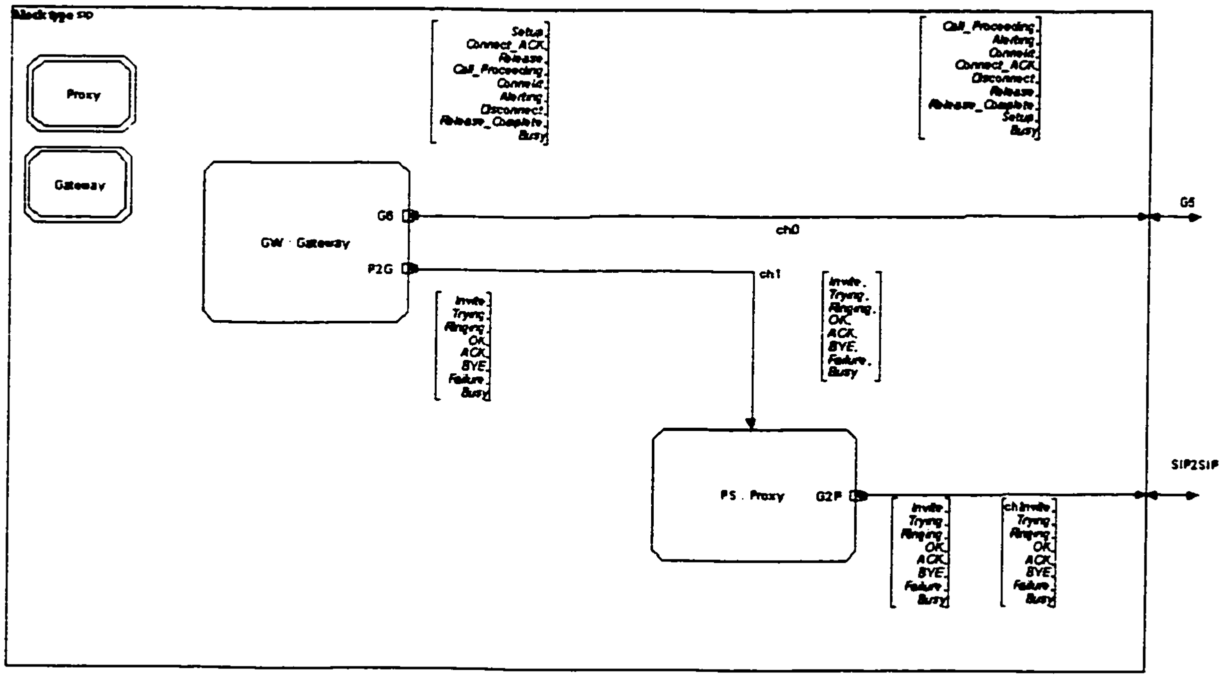


Figure 5.9: *sip* block type specification.

5.4.3 Refinement of Use Cases into Design MSCs

The architecture of the system is three layers deep. The use cases represent the view of the system at the top layer; therefore a maximum of two vertical refinements can be performed on them. Figure 5.10, Figure 5.11, Figure 5.12 and Figure 5.13 show vertical and horizontal refinements of the use case MSCs from Figure 5.3. Notice that the single axis shown in the use cases has been decomposed into four to reflect the architecture. In each case, the refined MSCs conform to the use case MSCs.

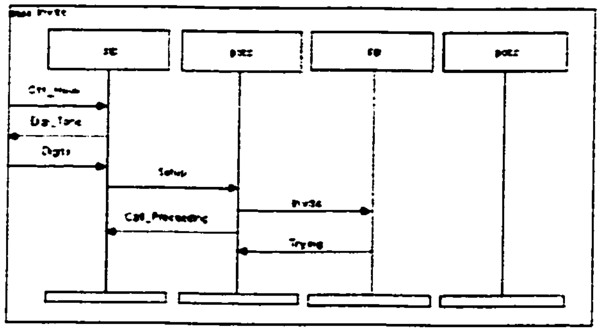


Figure 5.10: Vertical and horizontal refinement of the *Invite* bMSC.

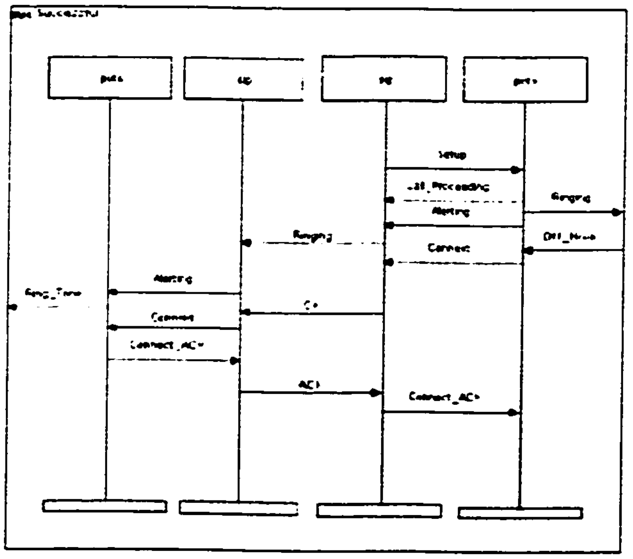


Figure 5.11: Vertical and horizontal refinement of the *Successful* bMSC.

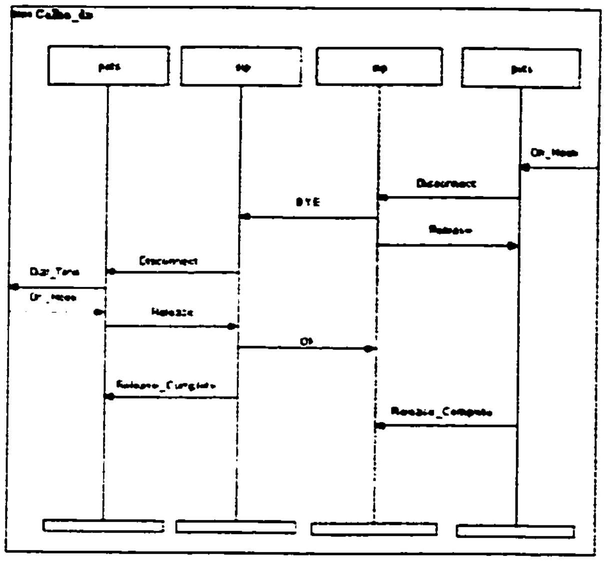


Figure 5.12: Vertical and horizontal refinement of the *Callee_dis* bMSC.

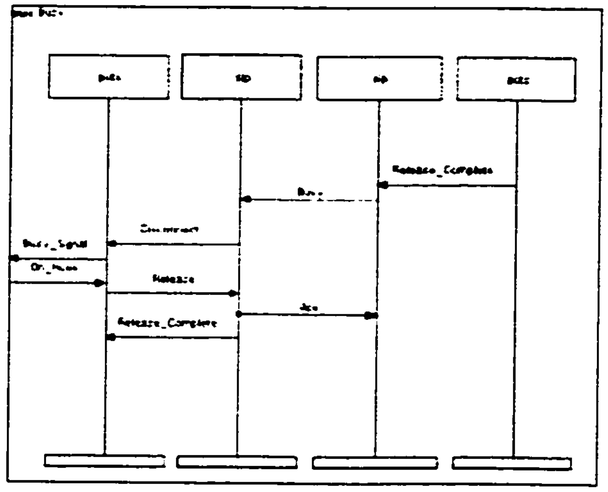


Figure 5.13: Vertical and horizontal refinement of the *Busy* bMSC.

A second vertical refinement decomposes the *sip* instance from the second layer MSCs into two instances, Proxy and Gateway, again reflecting the architecture of the system. Figure 5.14, Figure 5.15 and Figure 5.16 represent design MSCs that depict the lowest level of the system architecture, and which conform with the layer two MSCs. Also, global conditions have been added to the design MSCs so that the behavioral specifications will be compatible with the MSC-to-SDL tool.

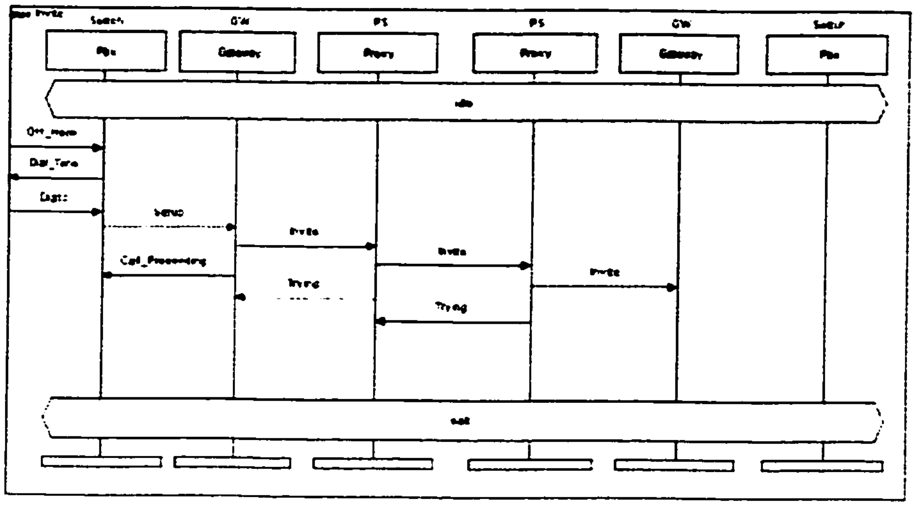


Figure 5.14: *Invite* design bMSC.

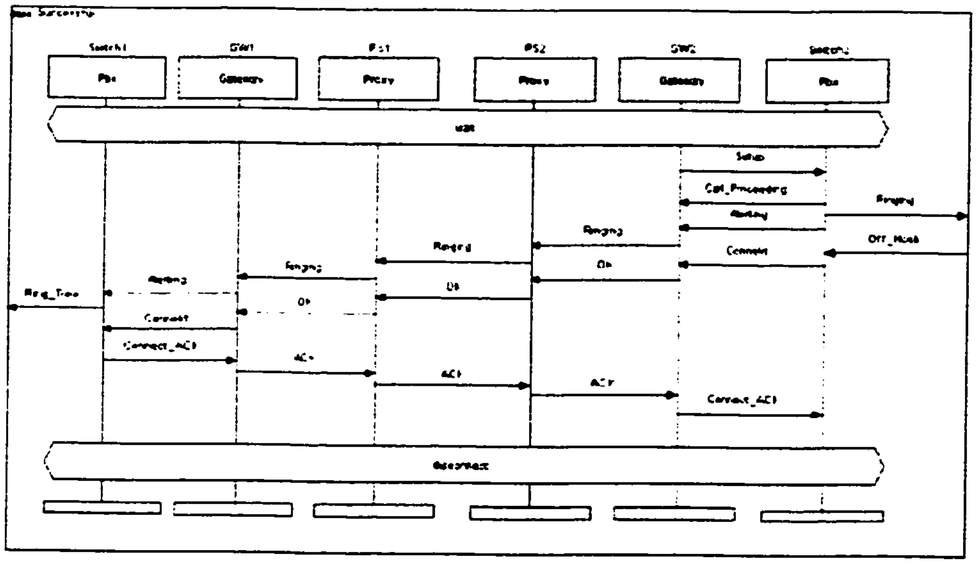


Figure 5.15: *Successful* design bMSC.

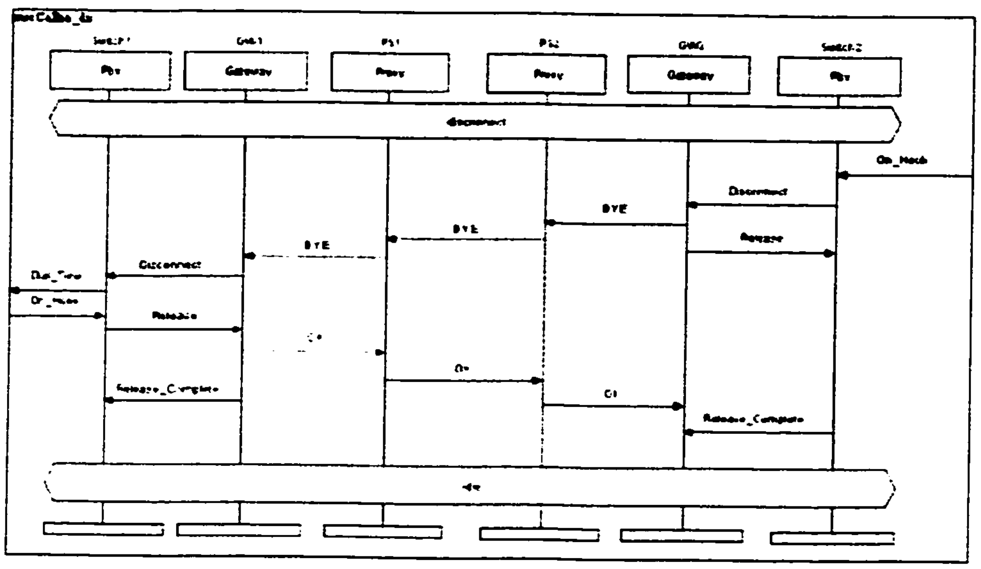


Figure 5.16: *Callee_dis* design bMSC.

The MSC in Figure 5.17 does not conform to the MSC in Figure 5.13 because there is no restriction that B (referring to the figure) can only happen after A, which is incorrect with regards to the previous MSC.

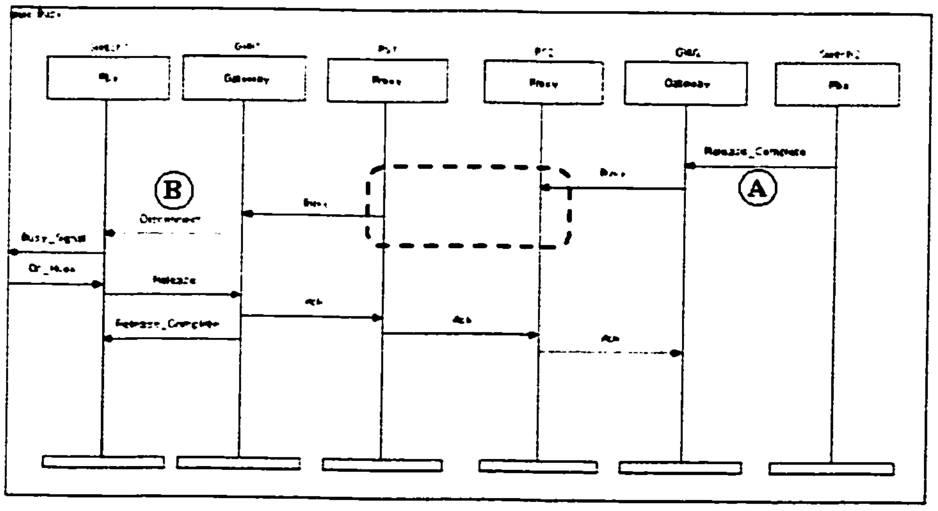


Figure 5.17: Non-conforming refined bMSC *Busy*.

Figure 5.18 shows a design MSC that conforms with the previous MSC because the ordering relationship between the "Release_Complete" and "Diconnect" signals has been maintained.

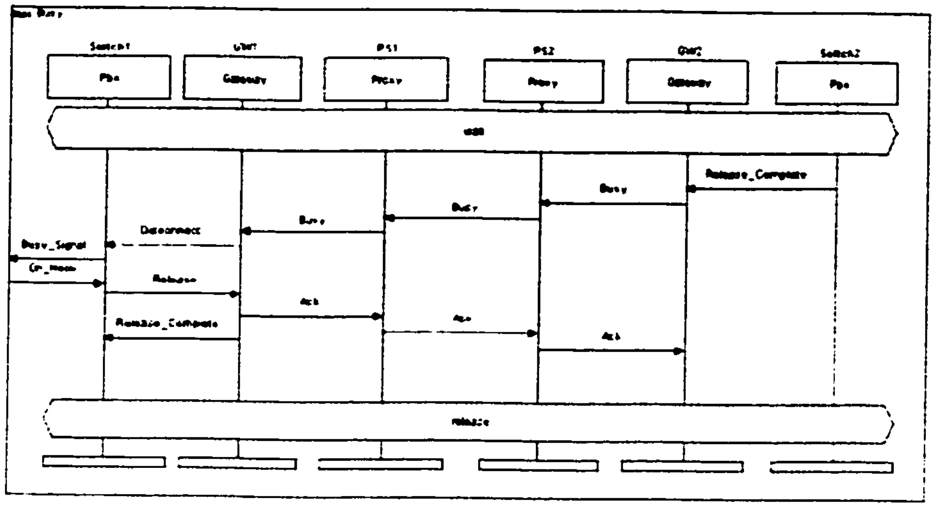


Figure 5.18: *Busy* design bMSC.

5.4.4 Generated SDL Behavior

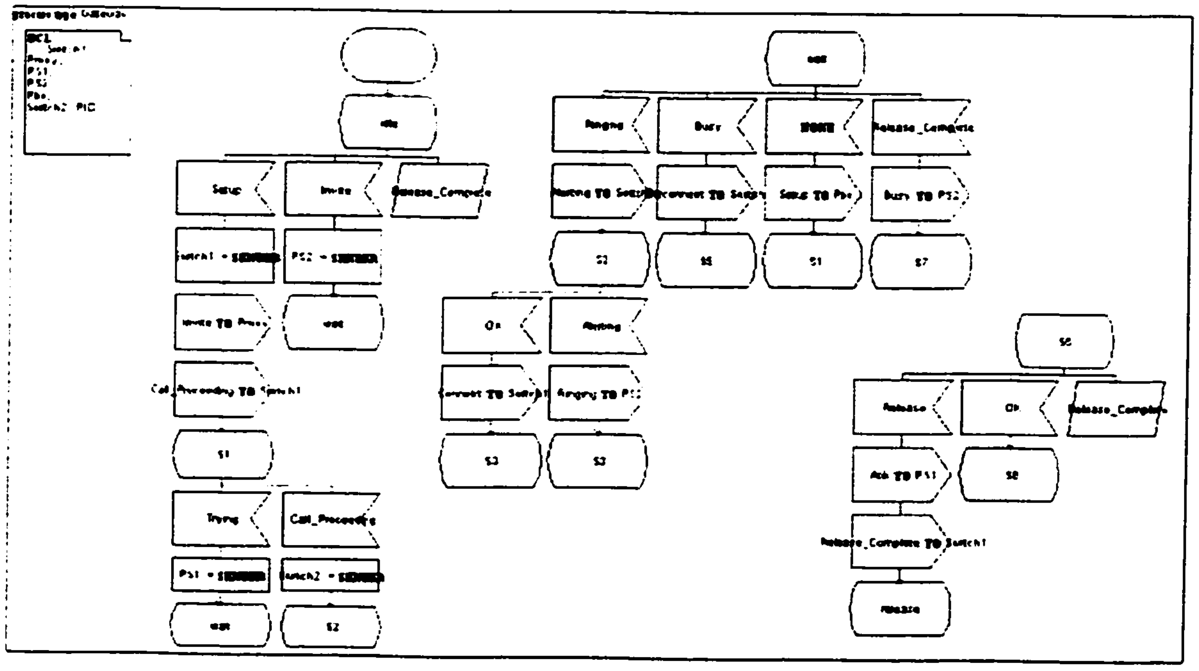


Figure 5.19: *Gateway* process type specification.

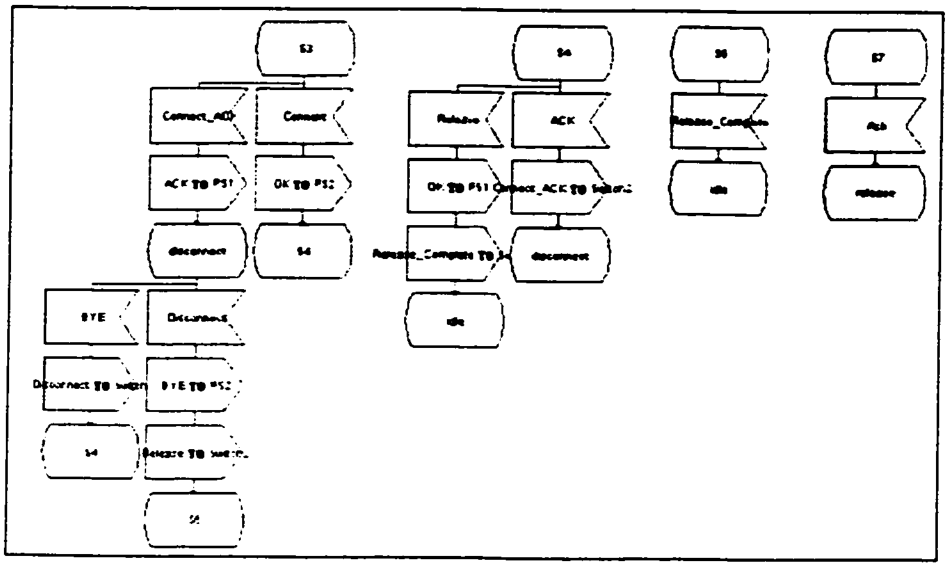


Figure 5.20: *Gateway* process type specification (continued).

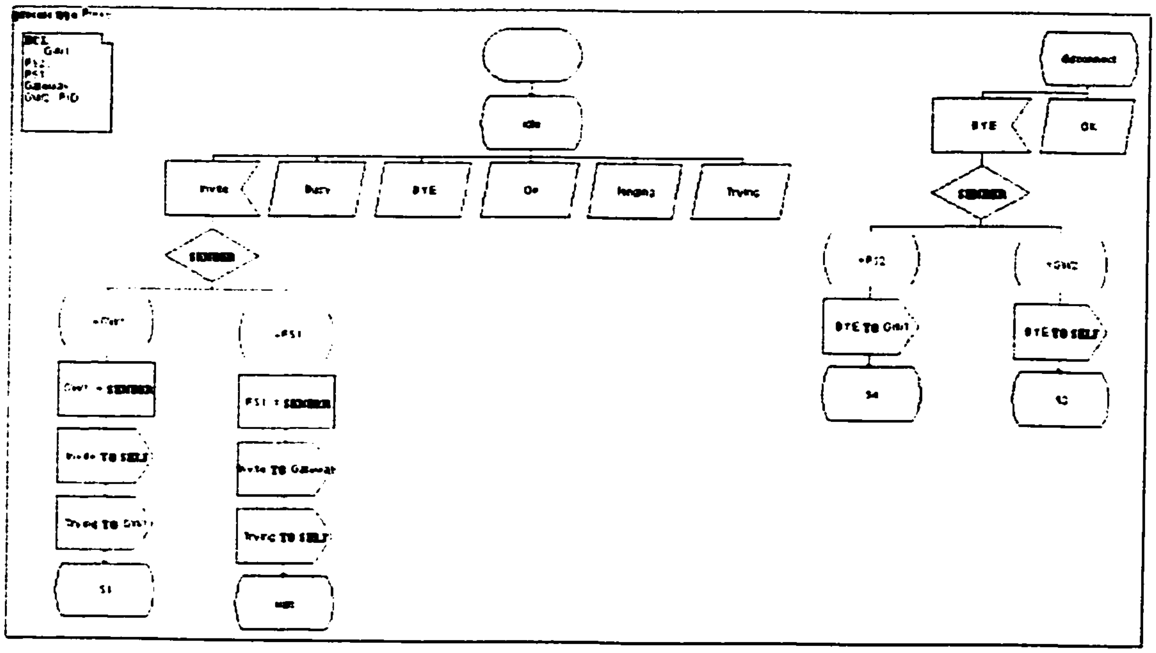


Figure 5.21: *Proxy* process type specification.

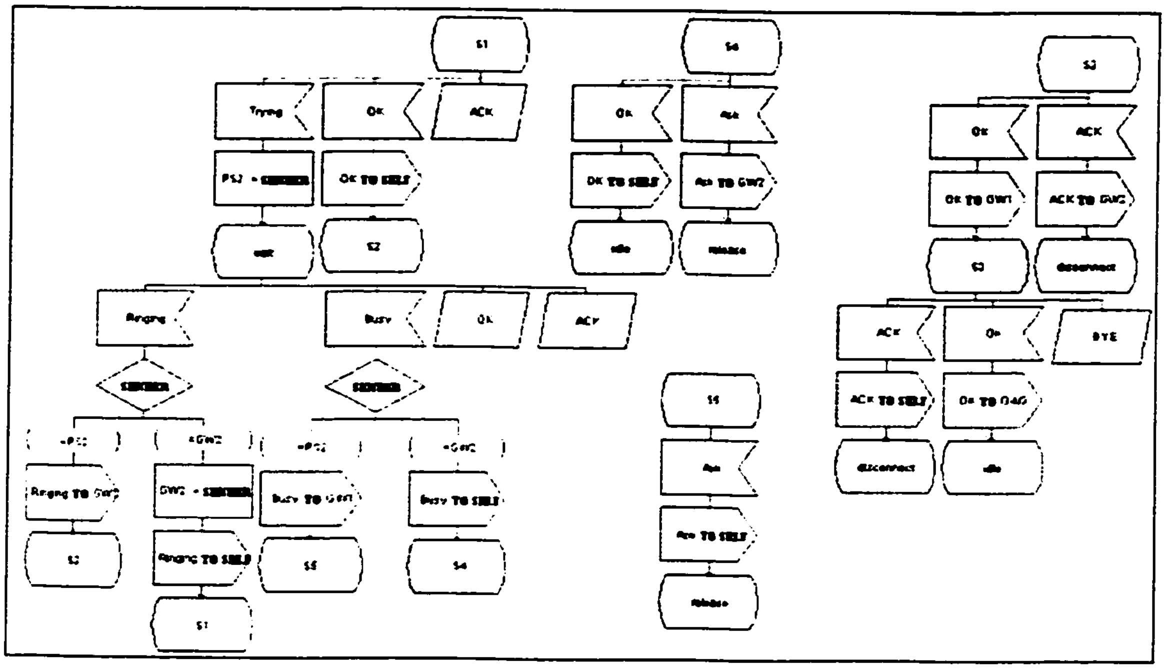


Figure 5.22: *Proxy* process type specification (continued).

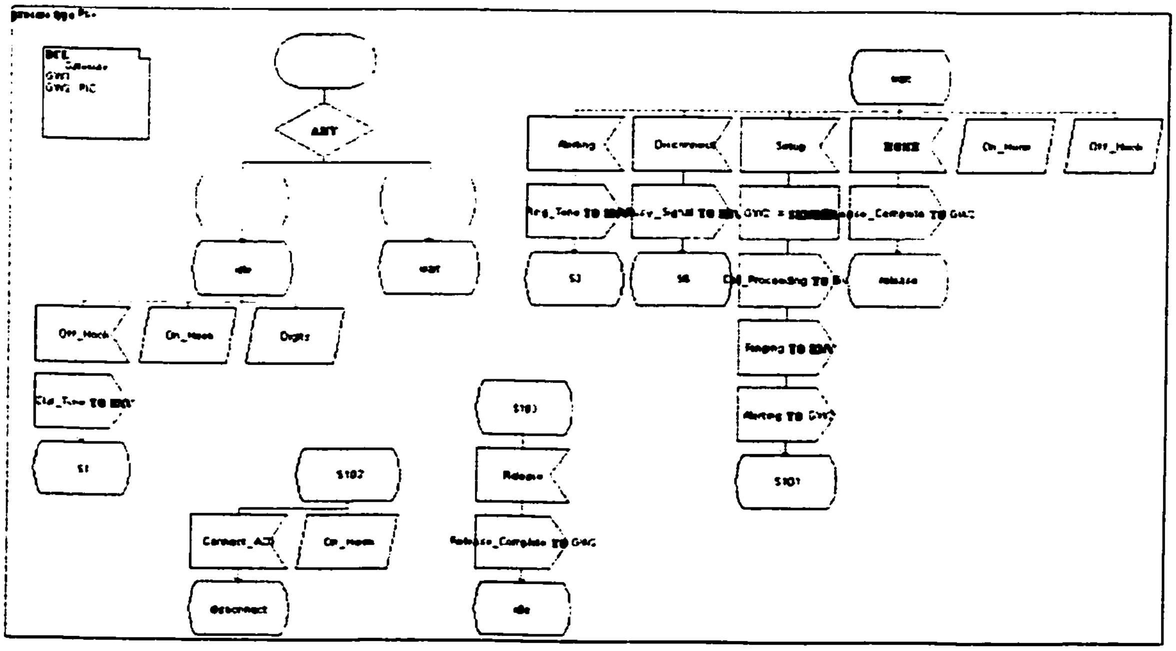


Figure 5.23: *Pbx* process type specification.

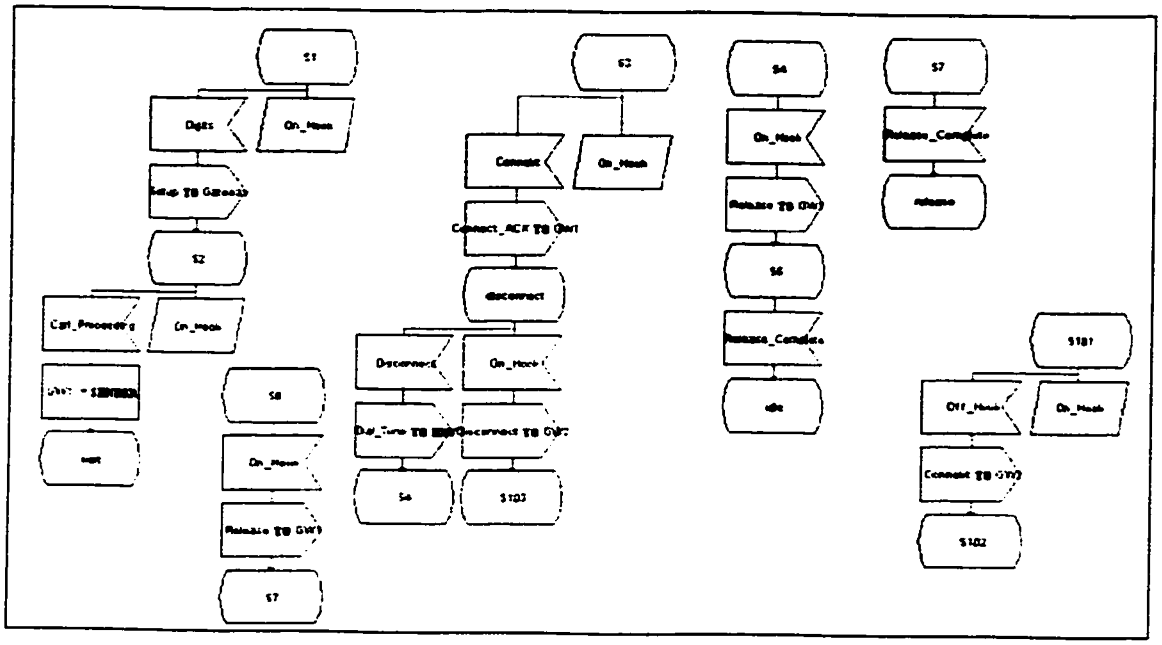


Figure 5.24: *Pbx* process type specification (continued).

Chapter 6

Conclusion

6.1 Contributions of this Thesis

An approach that uses the UML, MSC and SDL languages together from the requirements to design phases was presented. The MSC language was used as a starting point to capture user requirements. UML was then used to describe the architecture of the system. Once the architecture was complete, the MSC use cases were refined in a stepwise manner into design MSCs.

A stylized use of UML enables the architecture to be mapped into SDL, but the UML semantics are preserved without importing SDL via the UML extension mechanisms. A UML architecture is translated into an corresponding SDL architecture using the UML to SDL algorithm (and tool). The design MSCs describing the behavior of the system, and the SDL architecture, are combined into a *full* SDL specification using the MSC-to-SDL tool. The resulting SDL specification can be simulated, validated, etc. It can also serve as the starting point for code generation using existing tools, such as ObjectGeode [5] and SDT [6].

Current unstructured methods of creating software systems are not adequately suited to handle the increasing complexity of software systems. The use of formal and automated techniques such as the ones developed in this thesis will enable the development of such systems, which may otherwise be unfeasible.

6.2 Future Work

The approaches developed in this thesis are aimed at creating a suite of methodologies and tools to automate the development process as much as possible to increase the quality of software, reduce development time and errors, and increase confidence in the finished product. In the pursuit of answers, more questions are found instead. In the field of software engineering, it is impossible to define a single methodology that will work in every case, which means that there is always room for improvement. With that in mind, possible extensions to the work presented in this thesis include:

i. The MSC use case refinement approach can be expanded to support more of the MSC language such as inline expressions.

ii. Conformance checking algorithms between two HMSCs that have different roadmaps can be explored.

iii. In general, it is possible to express a single bMSC with inline conditions as a HMSC with simpler bMSCs. It would be useful to develop algorithms that allow a designer to compare a bMSC with inline expressions to a HMSC.

iv. It may be interesting to develop a conformance relation that checks if a modified SDL process behavior conforms to the design MSCs. This would have value if additions to, or modifications of, the SDL specification were necessary.

v. Real-time systems, which are notoriously difficult to develop, could be used with a high degree of confidence if verified using conformance checking algorithms such as the ones presented. It would therefore be greatly beneficial to include real-time aspects in the modeling languages used for development, namely UML, SDL and MSC.

References

1 G. Hassan, Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley, July 2000.

2 I. Sommerville, *Software Engineering, Fifth Edition*, Addison-Wesley, 1997.

3 ITU-T, Z.100, "*Specification and Description Language – SDL '2000*", November 1999.

4 J. Ellsberger, D. Hogrefe and A. Sarma, *SDL: Formal Object Oriented Language for Communicating Systems*, Prentice-Hall, 1997.

5 ObjectGeode, Telelogic, Toulouse, France, 2001.

6 SDT Tau, Telelogic, Sweden, 1999.

7 OMG, "*Unified Modeling Language Specification*", Version 1.3, June 1999.

8 H. E. Eriksson and M. Penker, *UML Toolkit*, John Wiley & Sons Inc., 1998.

9 I. Jacobson et al, "*Object-Oriented Software Engineering*", Addison-Wesley, 1992.

10 ITU-T, Z.120, "*Message Sequence Charts – MSC'2000*", November 1999.

11 S. Mauw, M. A. Reniers and T.A.C. Willemse., "*Message Sequence Charts in the Software Engineering Process*", Report 00/12, Department of Computer Science, Eindhoven University of Technology, 2000.

12 E. Rudolph et al, "*Tutorial on Message Sequence Charts (MSC'96)*", FORTE/PSTV'96, Kaiserslautern, Germany, Oct. 1996.

13 M. Andersson and J. Bergstrand, *Formalizing Use Cases with Message Sequence Charts*, Masters Thesis, Lund University, Sweden, 1995.

14 G. Robert, F. Khendek and P. Grogono, "*Deriving an SDL Specification with a Given Architecture from a Set of MSCs*", in SDL'97: Time for Testing - SDL, MSC and Trends,

Proceedings of the eight SDL Forum, A. Cavalli and A. Sarma (eds.), Evry, France, Sept. 22 - 26, 1997.

15 M. Abdalla, F. Khendek and G. Butler, *"New Results on Deriving SDL Specifications from MSCs"*, Proceedings of SDL Forum'99, Montréal, Canada, June 22-25, 1999.

16 www.uml.org

17 www.rational.com

18 J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

19 M. Abdalla, *Automatic Generation of SDL Specifications from MSCs*, Masters Thesis, Electrical and Computer Engineering, Concordia University, 1999.

20 F. Khendek, S. Bourduas, D. Vincent, *"Stepwise Design with Message Sequence Charts"*, to appear in Proceedings of FORTE 2001, Korea, August 2001.

21 A. Muscholl and D. Peled, *"Analyzing Message Sequence Charts"*, Proceedings of SAM'2000, Grenoble, France June 26-28.

22 A. Muscholl, D. Peled and Z. Su, *"Deciding properties of message sequence charts"*, in proceedings of FoSSaCS'98, LNCS 1378, Springer, 1998.

23 R. Alur, G. Holzmann and D. Peled, *"An analyzer for message sequence charts"*, Software Concepts and Tools, 17(2), 1996, pp. 70-77.

24 S. Mauw and S. A. Reniers, *"An algebraic semantics for Basic Message Sequence Charts"*, The Computer Journal, 37(4), 1994, pp. 269-277.

25 S. Mauw and M. Reniers, *"Refinement in interworkings"*, Proceedings of CONCUR'96, LNCS 1119, Springer, 1996.

26 S. Bourduas, F. Khendek and D. Vincent, *"From MSC + UML to SDL"*, Technical Report, Concordia University, November 2000.

27 Telelogic, *"The UML to SDL Tool and the Z.109"*, www.telelogic.com

28 ITU-T, *"Recommendation -Z.109"*, 1999.

29 Telelogic, *"Why UML without SDL Cannot Get Real-Time Right"*, http://www.telelogic.com.

30 B. Selic, J. Rumbaugh, *"Using UML for Modeling Complex Real-Time Systems"*, White paper, ObjecTime Limited, Ottawa, Canada, 1998.

31 A. Lyons, *"UML For Real-Time Overview"*, White paper, ObjecTime Limited, April 1998.

32 H. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, September 1998.

33 B. Selic, *Real-Time Object Oriented Modeling*, John Wiley, 1994.

34 B. Selic, J. Raumbaugh, *"Mapping SDL to UML"*, Rational Software White Paper, 2000.

35 R. Braek et al, *"TIMe at a glance"*, TIMe version 4.0, SINTEF, Norway, 1999.

36 K. Verschaeve, *"Three Scenarios for Combining UML and SDL 96"*, Proceedings of SDL Forum'99, Montréal, Canada, June 1999.

37 http://java.sun.com

38 http://argouml.tigris.org

39 OMG, *"XML Metadata Interchage Specification"*, Version 1.1, November 2000.

40 B. Douskalis, *IP Telephony: The Integration of Robust VoIP Services*, Hewlett-Packard Professional Books, Prentice Hall, 2000.

41 cisco.com/univercd/cc/td/doc/product/software/ios121/121newft/121t/121t3/sipcf2.pdf