Automated Enrichment of SDL Specifications with MSCs

Muhammad Umer Waqar

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 at

Concordia University

Montréal, Québec, Canada

November 2001

0-612-68446-6

ABSTRACT

Automated Enrichment of SDL Specifications with MSCs

Muhammad Umer Waqar

Formal methods and Formal Description Techniques (FDT) are becoming more and more important for developing complex real-time and distributed systems. Message Sequence Charts (MSC) and Specification and Description Language (SDL) are *standardized* FDTs for specification of telecommunication protocols in particular and distributed systems in general. We present an automated approach for *enriching* an SDL specification with MSC. This approach can be used for *incremental development* with SDL and MSC or for *maintenance* of a system in SDL.

Enrichment of SDL with MSC involves enrichment of SDL architecture and behavior. Our automated approach consists of three interconnected steps i.e. *pre-phase*, *MSC2SDL phase* and *post-phase*. In the pre-phase SDL architecture is enriched using new MSC. We give algorithms for enriching SDL architecture using MSC. MSC2SDL phase generates new SDL behavior using new MSC and enriched architecture. This phase uses MSC2SDL tool. Finally, the post-phase merges old and new SDL behavior to get enriched SDL. We have developed 14 rules of behavioral merger based upon a formal *extension* relation which guarantees to preserve all the old behavior and prevents introduction of new non-determinism in the enriched specifications. Furthermore, we have developed tools for *pre-phase* and *post-phase* and they have been connected with MSC2SDL tool to get a tool set for automated enrichment.

Dedicated to my beloved parents:

Abdul Sattar Aziz and *Maqbool Sattar*

ACKNOWLEDGEMENTS

This thesis would not have been possible without the guidance. encouragement. and support of Dr. Ferhat Khendek. His mentoring. coaching. availability and kindness was the driving force to keep me up and working. I am grateful to France Telecom R&D. and people involved. for their contributions and funding this research work. Special thanks to my colleagues and friends. Stephan Bourduas and Zheng Tong. for valuable comments. discussions. participations and jokes. I would also like to thank my parents and respected teachers to motivate me for a degree in research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

1.1 Real World vs Machine World

Computers do not think. They follow. They follow what they are told to do. They do not like abstractions. When giving instructions to them. we have to be very precise in what we want from these ingenuous. jargon-following slaves of ours. We are still living in an era where computers are primarily being operated by humans. Human beings act. behave. communicate. socialize and command abstractly. They like to be vague. They communicate ambiguously. Their languages are ambiguous. They follow different dialects of the same language. When it comes to getting services from the man-made machines. they have to be precise. Some *smart* people develop skills to communicate with machines. They can tell the machine what they want the machine to do. Such people are called *developers. programmers* or lately *software engineers.* At the dawn of computer age. programmers used to program in *assembly* languages. As we know. assembly languages are very precise and context sensitive depending upon the microprocessor and its architecture. This precise and concrete nature of these languages made them too hard to learn and master. Programmers started looking toward making their lives easier by shifting to what was later called *high-level languages.* They developed compilers. assemblers and interpreters to translate these high level languages into machine languages. This was a big relief on the programmer's life and it triggered a software boom anticipating what could be done by computers. It created further areas of research and development, more jobs and a life revolving around high-tech devices. It created an information age in which getting some information when it is required needs just a click of a button. All this was due to the fact that humans found a way to communicate with machines more abstractly as compared to *machine languages* or *assembly languages*. This search for *abstraction* continues. Researchers are still striving hard to attain a higher level of abstraction to command the computers. This resulted in the development of graphical languages and notations like UML[1], SDL[2] and MSCs[3]. This is pictorially depicted in figure 1.1.

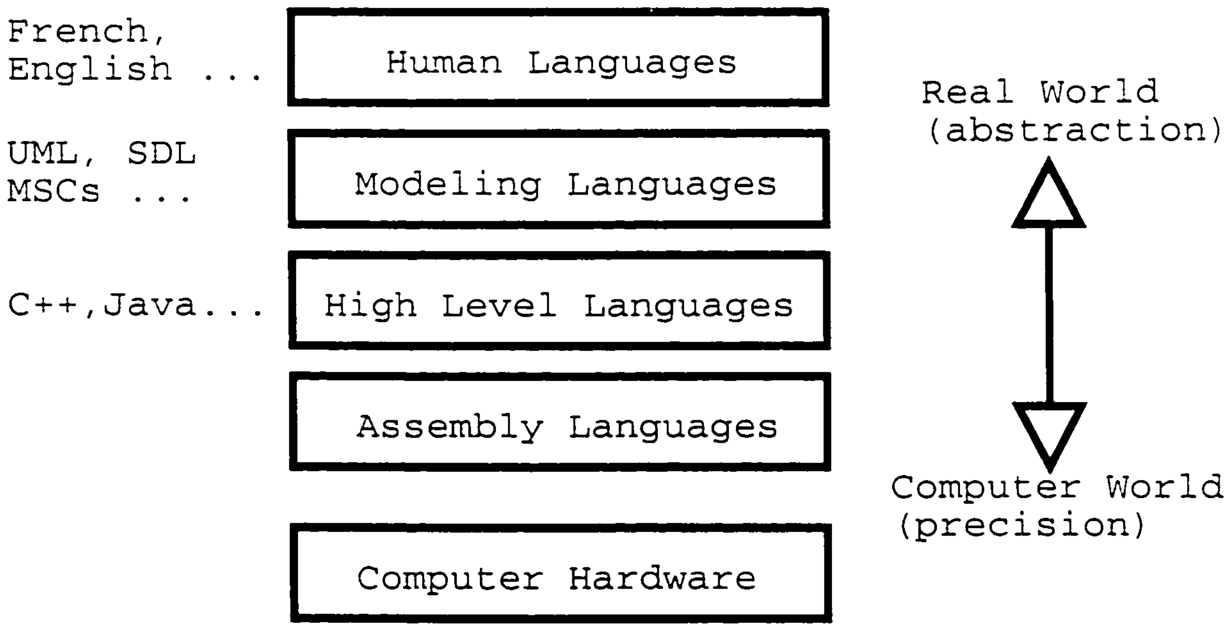


Figure 1.1: Real World and the Machine World.

Hence, as we go away from the hardware and into the real world, we achieve more and more abstraction. High-level languages like C++[4] and Java[5, 6, 7, 8] abstract the hardware and give a lot of programming liberty to the engineer. He or she does not have to care about the memory addresses of variables being re-calculated in a loop. Further search for abstraction led to the development of graphical notations[1, 9, 3, 2]. To cope with complexity, achieve modularity and re-use, the orientation of software development changed from *structured* approaches to the Object Oriented paradigm[10]. Currently, software is being developed using different development processes models[11]. We will have a brief look at these processes.

1.2 Software Development Process Models

These include the classical waterfall model, iterative waterfall model[12], evolutionary model[13], spiral model[14] and the transformational model. Classical waterfall model is based upon traditional engineering principals like civil engineering. It goes back to 1960's. The *classical* waterfall model does not expect any feedback from the later stages to the earlier stages of the model. But this is not always applicable to the engineering of software, hence it lead to the development of the *iterative* waterfall model. *Evolutionary* model is used when the requirements have a changing and evolving nature. Each cycle results in a *prototype* which is then *extended* with new requirements until the system is completed. Evolutionary model has influenced the *Unified process*[15], which uses UML[1] for system development. The *Spiral* model combines all the approaches in other models. Waterfall and evolutionary models can be viewed as a special instance of the spiral model. The *Transformational* model is the one most relevant to our work. It is based upon deriving formal specifications which can be executed. The model is actually an extension of the waterfall model with prototyping. Based upon the formal requirements specification, an executable prototype is derived which is then verified and validated against the environment. The Final product is derived by transforming the formal specifications into an implementation using CASE tools. Major advantages of this model are automated code generation, which might be optimized, and formal verification and validation. However, the complete automation is not always possible. Development of the transformational model, its supporting methodologies and CASE tools is still an area of intensive research.

1.3 Motivation for the thesis

We have already mentioned the benefits of achieving abstraction in systems and software engineering. We hinted that a higher level of abstraction can be attained using formal and semi-formal languages such as SDL , MSC and UML. The *transformational model* suggests usage of formal, executable prototype specifications and then automated code-generation. This can be extended to the maintenance phase where the system exists but needs extension or enrichment. We can call this as the *extended transformational model* which also encompasses the maintenance phase. Classical, traditional approaches suggest *maintaining* the code directly. Not only the maintenance of code is expensive and resource hungry but also bug prone especially when the engineers intend to preserve all the old architecture and behavior of an existing system while trying to enrich it. With the availability of formal languages and their supporting CASE tools, maintenance of the system can be shifted up to the specification level. Our extended transformational model, shown in the figure 1.2, proposes *maintaining* the *specifications* instead of code. Thus, we combine incremental development using prototypes and maintaining the code, which is quite different from the existing ones in literature[11, 16]. When we talk about incremental development in the context of formal specification languages, we have to define *enrichment* or *extension* of an existing system or prototype. Extension relations, in the context of SDL, have been defined in [17]. Also, it has been suggested to use MSC to capture new requirements and then enrich the existing SDL specifications using the MSC. An SDL specification consists of architecture (structure) and behavior. Enriching SDL specification should enrich both the architecture and the behavior. Enrichment of SDL architecture means adding new processes into existing or *new* blocks, adding new signalroutes or channels and adding new signals specification into existing or *new* links. Enrichment of architecture cannot be fully automated because addition of new entities into existing structure *might* be a design decision, which involves rationalization, problem domain, or engineer's experience. Hence, architectural enrichment of SDL specifications with MSC cannot be fully automated. In our work we develop algorithms and tool for architectural enrichment of SDL specifications with MSC. For the behavioral enrichment of SDL specifications with MSCs, we follow the *merger* approach i.e. new SDL behavior is merged with old SDL behavior to get enriched SDL behavior. New SDL behavior is generated from the MSC specification using the *MSC2SDL* tool and the enriched architecture. Merger of the old and new SDL specifications is done on a *transition by transition* basis. We develop 14 rules for merging SDL behaviors in this way. Resulting enriched specification can be verified, validated and can be used to generate code. Furthermore, since the enrichment is done on the basis of formally defined *extension relations*, it guarantees that:

- All the old behavior in the old system will be preserved in the enriched system.
- There will be no new non-determinism in the enriched system, in addition to the one already present in the old specification.

The aim of this thesis is to develop an approach for the extension/enrichment of existing SDL systems with MSC. It may be used for automated incremental development of specifications at design level or for maintaining the specifications to get the enriched code using CASE tools. The key point here is that the methodology is automated, as compared to others[18, 19, 20].

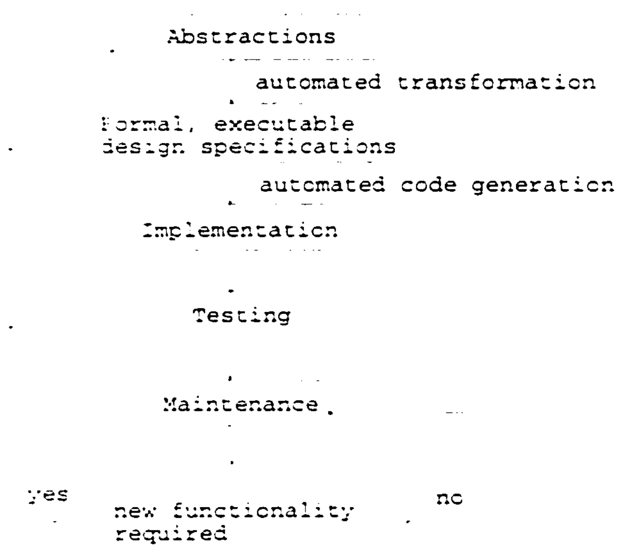

Abstractions
automated transformation
Formal, executable design specifications
automated code generation
Implementation
Testing
Maintenance
yes
new functionality required
no

Figure 1.2: The extended transformational model.

This work is a continuation to related work done by our group for developments of formal specifications using SDL, MSCs and UML. The group started its main work on attempts to produce algorithms to automate the translation from MSC to SDL and developed MSC2SDL tool[21, 22, 23, 24]. Based upon this tool, further research has been done on step wise refinement of message sequence charts [25] and an automated methodology from UML + MSC to SDL[26].

1.4 Organization of the thesis

The thesis is organized into seven chapters. As mentioned in the previous section. the main focus of this thesis is to produce an approach for enrichment of SDL specifications with MSC. The resulting work consists of algorithms. rules and tools which support the *extension* of SDL with MSC. Below we give a road-map to the organization of the thesis.

- **Chapter 2**: In the second chapter we discuss the formal languages in general and SDL and MSC in particular. We give all the background knowledge required to understand the remaining of the thesis.

- **Chapter 3**: We introduce an automated methodology for automated enrichment of SDL with MSCs. Then we explain this methodology in detail. As we will see. the methodology consists of two major steps i.e. architectural enrichment and behavioral enrichment. We do the behavioral enrichment by behavioral merger of the old and new SDL specifications.

- **Chapter 4**: We develop and discuss algorithms for *architectural* enrichment of SDL specifications with MSC. elaborating with simple examples.

- **Chapter 5**: In this chapter we develop and discuss algorithms for *behavioral* enrichment of SDL specifications with MSCs. As mentioned before. we suggest doing the behavioral enrichment using behavioral merger of the old and new specifications. Formal *relations* are defined for this behavioral merger. These relations are then used to develop *rules* to assist *manual* or *automated* behavioral merger of SDL specifications. The rules developed in this chapter are the guidelines for implementation of such a behavior merger tool.

- **Chapter 6**: In this chapter we introduce and elaborate the tool suite developed based upon the theoretical work. We also demonstrate the usage of the tools with examples.

- **Chapter 7**: In the last chapter we conclude upon the work done in general and show possible future directions of research.

Chapter 2

Formal Description Techniques

2.1 Introduction

To understand formal description techniques, one should know *formal specification languages*, *formal methods* and *formal systems*. A formal specification language is a mathematically based language which has a well defined *syntax* and *semantics*. Formal methods refer to techniques founded on discrete mathematics and logic in representation of the information necessary for the construction of software systems. These techniques include syntactic and semantic analysis, formal verification and formal validation. Modern day formal methods are generally accompanied by powerful CASE tools. A system having a formal language and using formal method is called a *formal system*. Since a formal system is based upon discrete mathematical foundation and logic, it constitutes a well defined *syntax*, *semantics* and a deductive mechanism. The syntax of a formal language is described by its *grammar*, which actually defines a set of *sentences*. These sentences are given meaning through definition of the semantics of the language. Well defined and non-ambiguous semantics of the formal languages make them outshine non-formal languages. The deductive or the inference mechanism allows the derivation of new well-formed formulae from those already present in the language.

The techniques used in the development with formal methods are called formal description techniques (FDTs). Several FDTs exist. They were originally developed to cope with the complexity of concurrent, real time and distributed systems. They are more and more being used for safety-critical systems. Their strength lies in their standardization through international bodies. A standardized FDT ensures harmony in the development and also global comprehension. Estelle[27], LOTOS[28], SDL[2] and MSC[3] are the standardized FDTs. SDL and MSC, standardized by ITU-T, are the most used ones and they are more and more being used in conjunction with each other. We will give brief introductions to SDL and MSCs later in this chapter.

FDTs promise early detection of errors. An error detected after the system has been developed is about 1000 times more expensive[16] as compared to one discovered during the specification stages. FDTs support early detection of errors through *simulation*, *verification* and *validation* of the model.

As shown in figure 2.1, Development with FDTs start with an English language description followed by a formal description. It is called the *model* of the system. The model is then simulated to see if the system works. Some functional bugs may be caught at this stage. The simulation is iterative in nature i.e. as a bug is caught, it is corrected in the specification model. Simulation is followed by formal verification and validation.

2.1.1 Simulation

Simulation is done at any stage of development with FDTs. It is a partial proof that the system does what it is supposed to do. It does not guarantee detection of all the errors. Simulation, however can serve as a good tool by the developer to gain confidence and also communicate with the client. Simulation results are normally shown in a graphical trace language.

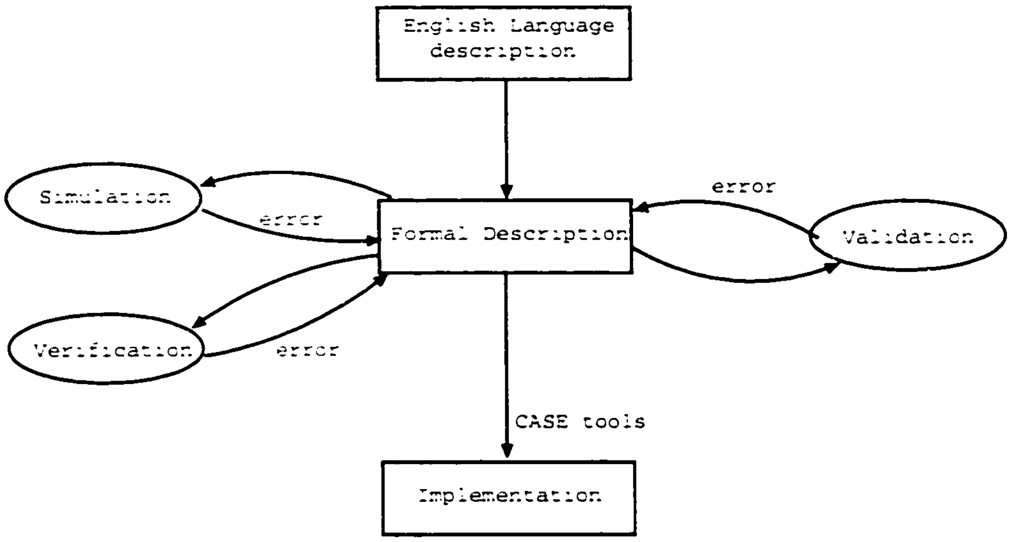


Figure 2.1: Usual approach of formal development with simulation, verification and validation.

2.1.2 Verification

Verification is a formal proof that the system works properly. During the verification *general properties* of the system are checked. This includes verifying that the system has the *safety* and *liveness* properties. **Safety** property means that *something bad will not happen* in the system. Examples of safety property are absence of deadlocks and unspecified receptions etc. *Deadlock* is a situation in which all the entities (processes) are blocked and waiting for an input while the communication (channel) queues are empty. A system in a deadlock cannot move on. *Unspecified reception* is a situation when an entity has an unexpected signal in its input queue, ready for consumption. What the entity will do with such a signal depends upon the semantics of the FDT being used. For example, SDL has a non-blocking semantics, such that an unexpected input signal in the input queue of a process will not block the process. It will be discarded. Hence SDL does not have unspecified receptions. **Liveness** means eventually *something good will happen* in the system. A system having the liveness property is free from livelocks. *Livelock* is a situation when a system is running but without any progress. It is like a continuous loop without doing any work. Even small systems are difficult to verify by hand. Verification of protocols etc.. is impossible without tools. Verification ensures that the formal model has *safety* and *liveness* properties.

2.1.3 Validation

Each system that is designed has some unique properties which differentiate that system from others. The properties checked during verification (safety. liveness. etc.) are common to all the systems. but validation checks for validity of some *specific properties* inside the system. Validation might prove to be resource hungry and require powerful computers but it is worth knowing that the system being built has the property for which it is being built.

2.2 SDL

SDL[29. 30. 2] is a standardized specification language supported by many CASE tools[31. 32]. It has well defined semantics and is one of the most popular FDTs currently being used[30. 16]. It has a graphical form called SDL/GR (graphical representation) and a textual form called SDL/PR (phrase representation). SDL has a structural part also called *architecture*. It has a behavioral part based upon extended finite state machines as well as a data part. It not only serves as a specification language but also serves as a notation to derive the implementation automatically. It has been designed to support the specification and development of reactive. concurrent. real-time and distributed systems. However major application domain of SDL is telecommunications and protocol specifications.

SDL evolved and developed in Europe and enjoys a lot of support of telecommunication communities. A major revision of the SDL standard in 1992 resulted in Object Oriented extension of SDL. SDL96[30] is a fully Object Oriented specification and description language. Current release of SDL is SDL2000[2] and it has further modeling constructs especially for Object Orientation and is quite close to UML from OMG[33]. In the following subsections, we discuss the *structural (architectural), behavioral* and *data* concepts of SDL.

2.2.1 Structural Modeling in SDL

SDL has very strong support for modeling the structure or architecture of the system with hierarchy starting from top level specification to detailed specifications. It also supports hierarchical object oriented modeling, thus providing it more strength for modular modeling. The structure of the system, however, may or may not directly represent the way a system is implemented. Nevertheless, high level structuring and modeling can be used to derive models which closely represent the implementation details. We will mention the architectural concepts of SDL starting from the top level of hierarchy, that is *system, block, process, services* and *procedures*. SDL allows communication between the entities either through *channels* or *signalroutes*. Since process, services and procedures are major components of SDL behavior, we will elaborate them in detail later. In Object Oriented Terminology, an object is defined by its type and then it is instantiated into instances which are actually used at the run time. In Object Oriented SDL, a construct name without a *type* suffix represents an instance. So when we say a *system* we actually mean a *system instance*. Thus, as we note, a class in SDL is called a *type* and an object is called a *reference*. Also in conventional OO world, an object encapsulates both the structure and behavior but this might not be the case in SDL.

SDL types can be instantiated and/or specialized where ever required in the specification. These *type* specifications can be placed into **packages**, which can be imported into other specifications. This promotes re-use of specifications. Further, in SDL the types can be *parameterized* using *formal context parameters*. For the consideration of the specification re-use it is preferable to use *type*d specifications instead of a *structured* approach.

System and System Type

A System is the highest level of hierarchy in SDL modeling. It contains all the other constructs at the lower level of hierarchy such blocks and processes. The interpretation of an SDL system is the interpretation of the contained SDL processes running in parallel. A system type defines the *blue print* of a system instance. System types can be specified and kept in packages while these packages may be imported in other packages and the system types may be instantiated into references. A system *type* specification is shown in figure 2.2. The text area at the upper right corner of the figure specifies the type of signals transmitted in the system. The block instances are connected through channels and the signals transmitted through these channels are specified within brackets right next to the arrows which indicate the direction in which the respective signals are sent. The system type can be parameterized using *formal context parameters* and can even be specialized from other system types using *specialization*. The system type in figure 2.2 is shown to be instantiated as **sysinst** in the box at the right of the figure.

Block and Block Types

The second level of structural modeling constructed after a system is a *block*. A block may contain processes or other blocks. However. it may not contain a mixture of processes and blocks. A block may be specified as a *block type* and can be instantiated any where required. The system instance *sysinst* in figure 2.2 is an instance of system type *sys*. It consists of two block instances *b*1 and *b*2 which are of block type *bt*. The identifiers *g*1 and *g*2 inside the block symbols are gate identifiers used for connecting the channels to process instances inside the block instances. In the example the block type *bt* as well as the block instances are defined in the system

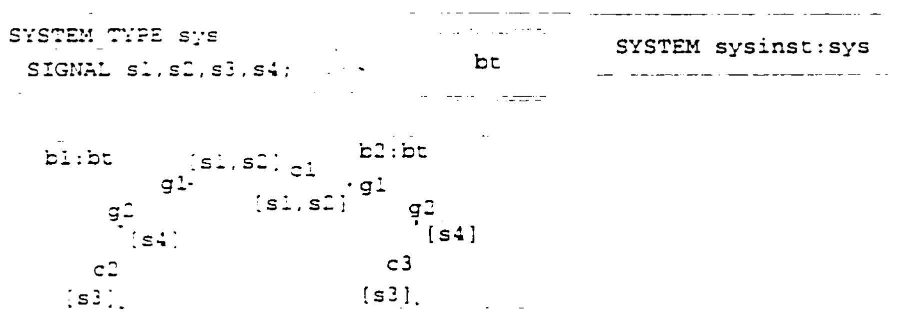


Figure 2.2: A System type containing a remote reference to a block type and two instances of that block type

type *sys*. but the properties of the block type are not specified in this diagram. This is because it would require actual nesting of the diagram which. in general. is not convenient with respect to readability and handling by tools. The block type symbol in figure 2.2 is just a reference indicating that the block type is conceptually defined here. but that the block type physically is defined in a separate diagram. Figure 2.3 shows the example of a block type diagram. A block type may be remotely specified in a re-usable package which may be imported wherever needed. It may also be parameterized using formal context parameters. Also. since the block type specification specifies without a particular context. it specifies the channels as gates. Hence when the block type is instantiated. the channels in the actual context are connected to the block gates.

Process and Process types

A process represents an independent entity. which encapsulates the behavior of the system. A process may be specified as a process instance or as a process type. In the block type of figure 2.3. two process instances are defined. p1 of process type pt1 and p2 of process type pt2. Process instances are connected by *signal routes*

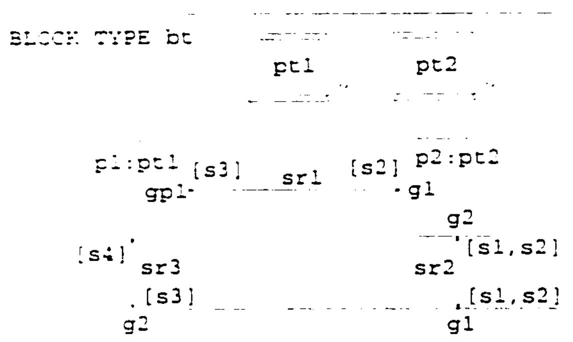


Figure 2.3: A block type

which convey signal instances to/from other process instances in the same block (as the case is for signal route $sr1$) or conveys signals to/from the channels connected to the block (as the case is for the $sr2$). A process type may also be parameterized with formal context parameters, or specialized into other process types. A process type specification may contain *service* type specifications or instantiations. Since a process type is specified with no actual context, so the signal routes are specified as gates associated with signal routes. The actual signal routes are connected to the gates when the process type is instantiated. The process type specification for $pt1$ is given in figure 2.4.

Service and Service Types

A process may have *service* or *service types* as elements to give further structure to the behavioral modeling. Normally the behavior of a process is defined as a state machine, but indeed, it may also be defined in term of number of service instances. Each such service instance is a separate state machine, but only one is executing its graph at a time. When the executing service reaches its state, the service capable of consuming the next signal in the input port of the process instance instance takes over interpretation, i.e. service instances share the input port of the process instance

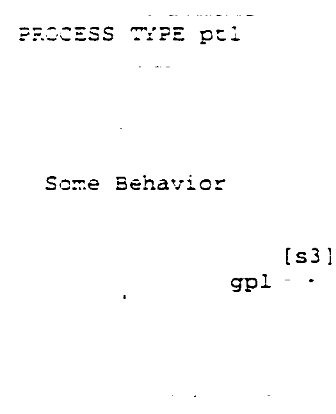


Figure 2.4: A process type specification

as well as the variable and the value. A service instance is capable of consuming a signal if the signal is not saved in the given state and if the signal can be received by the service instance. Two different service instances must not be capable of receiving the same signal. Services are useful when the behavior of a process can be described as a number of independent activities (only sharing data). e.g. the two directions of a protocol. or if some activities are common for process instances of different process types. In figure 2.5. the process type pt2 is defined in terms of two service instances s1 and s2 (of the service type st1 and st2 respectively). The service type st1 is shown in figure 2.6. Note that. graphically. all *type* specifications have the similar symbol as the *instance* specification except that the boundary is doubled.

Channels

Communication between blocks and between blocks and the environment is only possible along the defined *channels.* Channels can be unidirectional or bidirectional communication devices. The communication structure between blocks is static. Channels may be specified as delaying or non delaying communication devices. Communication on the channels is free of errors and preserves the order of the

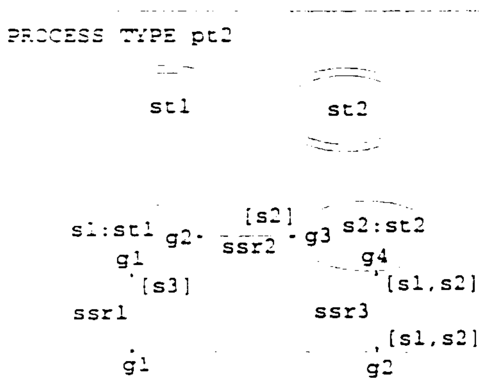


Figure 2.5: A process type specification with service types

transmitted signals. In order to refine the properties of channels. i.e. to model error or reordering of the of signals. channels may be refined by channel *substructures*.

Signal Routes

Communication between different processes and between processes and the block interface is done via signal routes. Signal Routes are non-delaying and may or may not be explicitly given in the block diagram. The most important difference between a channel and a signal route is their usage. Signal routes are used within blocks to connect processes. while channels connect blocks. In case a signal is sent to a process within a different block. the signal travels along the signal route in the same block. the channels connecting to the blocks. and finally the signal route defined in the block where the receiving process is located. Another difference is that the signal routes are always non-delaying. while channels may delay the signals.

Signals and Signal Lists

A signal is a primitive type in SDL. It abstracts the notion of message in communicating entities. Signal Lists group the signals together at the desired end of the link

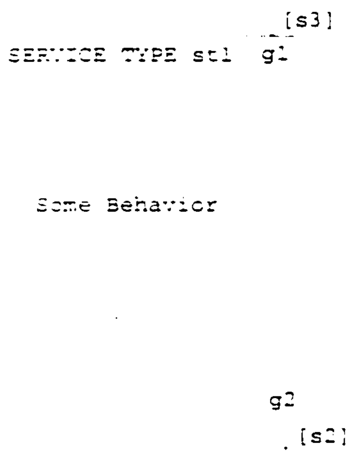


Figure 2.6: A service type specification

(channels and signalroutes).

2.2.2 Behavioral Modeling Concepts in SDL

Behavioral modeling in SDL is based upon the concept of Finite State Machines (FSM). A finite state machine is basically an input output automaton. Finite state machines are defined below.

Finite State Machines

Formally, a finite state machine is a 5-tuple (S, I, T, s_0, F) where:

S is a finite nonempty set of states.

I is a finite nonempty set of inputs.

T is a function from $(S - F) \times I$ to S, called the transition function.

$s_0 \in S$ is an initial state, and

F is the set of final states.

A state is an abstraction of the static condition symbolizing the event sequences leading to it. In general when an object assumes a particular state, it can be receptive to only a selective set of events (messages). The object changes its state only if one of these events occurs. The event that causes a state transition is called a *trigger* or a *triggering event*. An action is associated with each transition. The behavior of the object at any point is characterized by the history, sequence of alternating states, and actions. The state transition diagram gives all the possible behaviors of the object modeled by it.

SDL processes as Finite State Machines

The finite state machines are the active entities in an SDL system and are called *processes* in SDL. Processes are the main concept in SDL to define the dynamic behavior of the system. The most important graphical symbols used within an SDL process are given in figure 2.7

In addition to these basic symbols, many other constructs for behavioral modeling are supported in SDL. An example of a simple SDL process specification for a simple *Volume Switch* is shown in figure 2.8.

When the process *Volume Switch* is in the idle state, it waits for a signal *stimuli*. Upon getting a stimuli, it tells the actuating device that it is on the *on* state by sending a signal to it. Thus the volume switch enters an *active* state. In the *active* state if it receives a *vol_up* signal, it sends an *inc_current* signal to the amplifier to increase the volume. Similarly, in the *active* state, if it receives a *vol_down* signal, it sends a *dec_current* signal to the amplifier. While being in the active state, if the volume switch receives an off signal, it goes back to idle state. Now, to compare the SDL process behavior with a FSM diagram, we show a corresponding FSM diagram

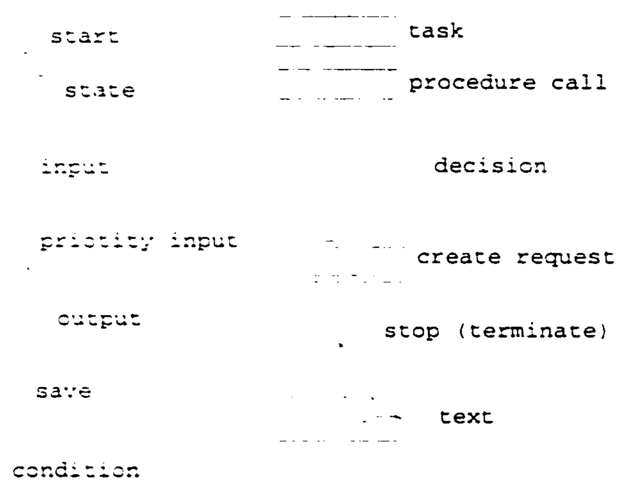


Figure 2.7: Most commonly used symbols in SDL

of the *volume switch* in figure 2.9.

Extended FSM

SDL extends the basic FSM concept by associating data with the state machines. These are typically called *extended finite state machines*. This denotes the fact that the state of SDL process is defined not only by the explicit states but also by the state of the data objects. i.e. the value of its variables. Data cannot be declared directly in the SDL *block* or *system* diagrams. Data is declared as SDL process. service or procedure level. One process can only access data in the other process through a remote procedural call or through import/export concept. We will discuss the data part of SDL later on.

Each SDL process maintains a set of intrinsic variables which are of pre-defined data type *PId* (process identity). The following variables of this type are defined for each process instance.

Self: defines the PId of the process instance itself

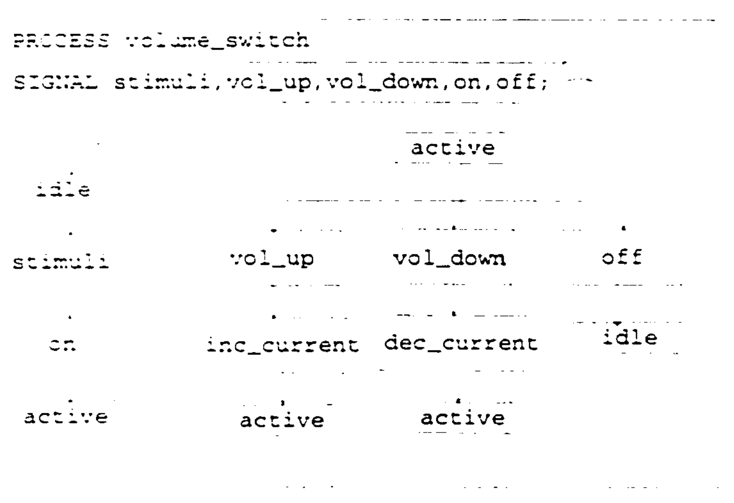


Figure 2.8: SDL specification of a volume switch

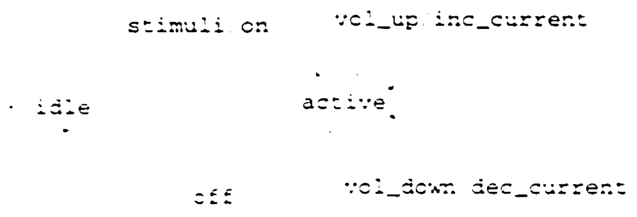


Figure 2.9: An FSM specification of the volume switch

Sender: denotes the PId of the process instance from which the most recent signal has been consumed.

Parent: denotes the PId of the process instance that has created the instance.

Offspring: denotes the PId of the most recent process instance created by this process instance.

Communicating Extended Finite State Machines (CEFSM)

An SDL specification typically comprises of more than one communicating process. Communication between the processes is based upon the concept of communicating extended finite state machines (CEFSM). SDL processes communicate with each other and with the environment by exchange of *signals*. A *signal* is identified by a signal type identifier. In addition a signal may carry data by means of *parameters*. The data carried as parameters to signals may be of any size. All the signals used in SDL have to be well defined as signal types. In addition, the channels or signal routes supposed to carry these signals have to be well attributed and properly defined.

Input Queue

Each SDL process has an input queue of unlimited capacity. So SDL processes have non-blocking semantics. The input queue is organized according to FIFO (first in first out) principle. However there exist priorities for inputs also possibility of *saving* signals for later consumption. So the FIFO principle is not always preserved. We will discuss saving of the signals later.

Concurrency and Latency

Each process in SDL is an independent, asynchronous CEFSM. Conceptually there is no interdependence between one process and another except the explicitly defined interprocess communication and exported/shared variable.

The time delay involved in between the triggering and actual happening of different events is undefined. A process instance might get executed as soon as one of its trigger conditions hold. However the execution might as well be delayed. Similarly the execution of a transition may or may not consume time. However the transitions of the process instances are executed sequentially, that is to say, the next transition of a process instance may not execute until the previous one has executed. Also the transition is SDL are not *atomic*. Thus there might be arbitrary inter-leaving of different actions concurrently executed by different process instances.

Triggering of Transitions

An SDL process in a certain state may be triggered into a transition in any of three ways. Consuming an expected implicit signal in the input queue, receiving a timeout signal from an expired timer (a timeout signal is also considered like any other SDL signal) or due to a changing condition of its variables. As a result of any of the triggers, a set of actions may be performed, typically including the (asynchronous) sending of signals to other SDL processes.

Timers

SDL supports two ways to deal with time, i.e., directly access the time and use of timers. To access the time, two predefined data types are used in SDL, namely data type **time** and **duration**. The *now* construct allows one to access the current time e.g. 9:45am which can be stored in a variable of type *time*. The data type *duration* specifies the time span in the defined units of time. Timers can be **set** or **reset**. A keyword **active** can be used to inspect if a timer has expired or not. When a timer expires, a special *timeout* signal is generated. The timeout signal is considered as an input to the process instance that owns the timer and is handled in a similar way as any other input signal. A timer is local to the process instance. A *reset* of the timer before it expires simply deletes the timer. A *reset* when the timer has expired but the timeout signal has not yet been consumed yet also results in deletion of the timer. In the case when the timer has expired, and the timeout signal has been consumed, a *reset* has no effect. In the case when the timer is *set* again before it expires, it is set to a new value. An example of a simple machine which after accepting *money*, gives a choice to select *coffee* or *tea* is given in figure 2.10. If nothing is selected within 10 units of time, the money is returned to the user.

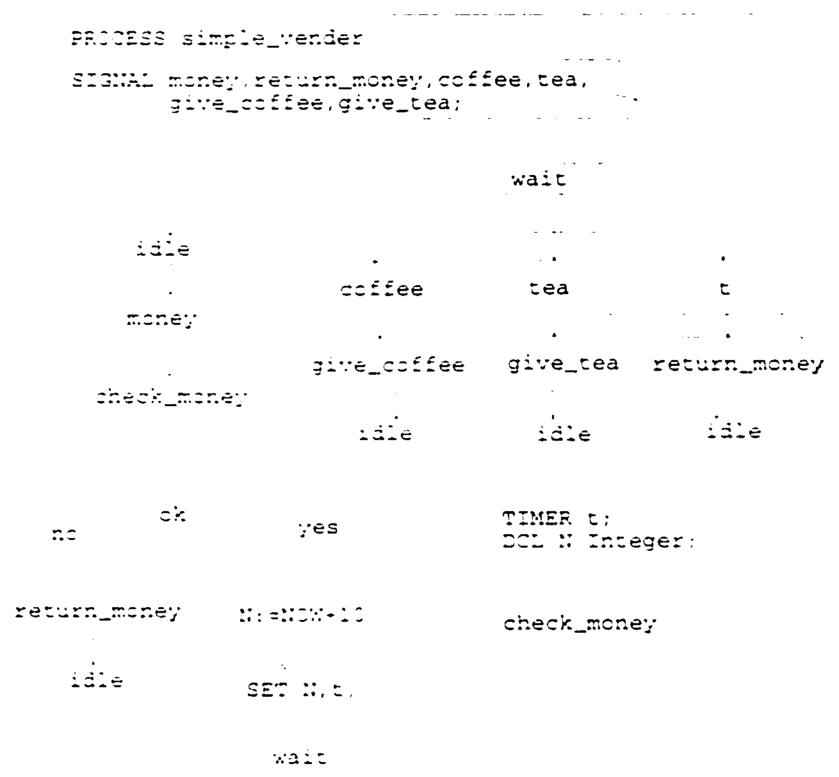


Figure 2.10: A simple machine which uses timers

Priority Inputs

All systems may have inputs which have to be prioritized for proper modeling. In SDL behavioral modeling this is supported by a *priority* input signal. A signal specified in a priority input for a particular state will be the first one to be consumed. Actually when a state has priority input in addition to other inputs, all the other signals are implicitly saved. An example of a priority input concerning the simple vendor machine will be when the process is in the *wait* state, a *cancel* signal should be given higher priority, as shown in figure 2.11.

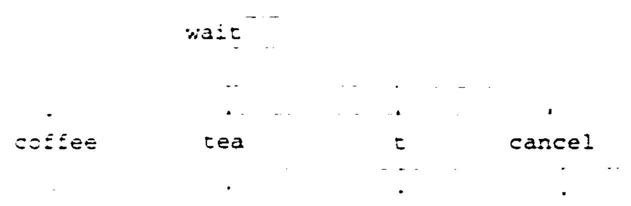


Figure 2.11: A priority input

Modeling non-determinism through NONE and ANY constructs

SDL strongly supports the modeling of non-determinism through two very important constructs which are **NONE** and **ANY**. A *none* signal is like a null signal which can trigger a transition from a state. The keyword *none* within an input symbol specifies that the transition may be triggered non-deterministically, i.e., at any time when the process is in the specified state. A transition using a *none* signal is called a *Spontaneous Transition*. Most important usage of the *none* signal is the modeling of the failure of the system. Other important construct supporting non-determinism is *any*. It is specified inside a choice symbol between two branches (transitions). Any branch may be chosen non-deterministically. An example of extreme non-determinism modeling is shown in figure 2.12. A process *P* in state *s*1 can either chose to follow a branch b1 or b2, without even waiting for an input signal. This is an example of mixing a spontaneous transition with a non-deterministic choice.

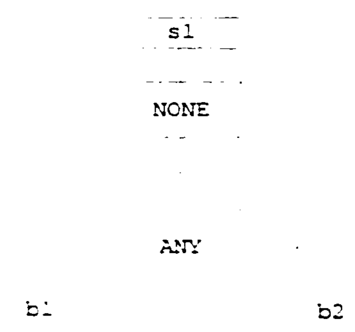


Figure 2.12: Non determinism in SDL

SAVE Construct

In case a signal or timeout is encountered in the input queue. while it is not specified in the current state of the process. the signal (or timeout) is simply discarded. To prevent this. i.e. to prevent the signal from being discarded. the **SAVE** construct of SDL can be used. In such a case the respective signal is saved and is reconsidered in the next state.

Continuous Signal

So far we have discussed three kinds of triggers. i.e.. an explicit input signal. a timeout signal from a timer or a spontaneous transition. There is a fourth kind of triggering event called the *continuous signal.* It implicitly depends upon the state of the process. A continuous signal is specified in angle brackets. for example when the *vendor machine* is in the wait state. an internal error may happen. This is modeled using a continuous signal. as shown in the figure 2.13. Continuous signals have lower priority as compared to other triggering events. If however. several continuous signals are specified. one of them is chosen non-deterministically.

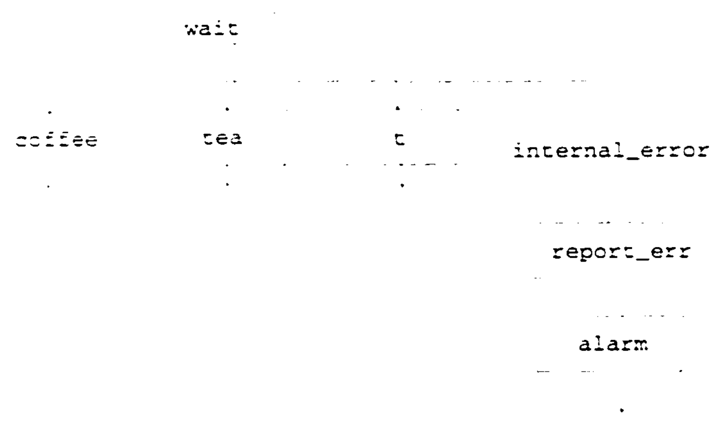


Figure 2.13: Continuous Signal in SDL

Addressing mechanism in SDL

Addressing is a very important issue in telecommunications. In SDL, there are two basic mechanisms for addressing.

- **Explicit Addressing:**

  Explicit addressing directly specifies the address of the receiver or the called process instance. In SDL this is supported by a **TO** clause in the output constructs followed by the identifier of the receiving process instance i.e. its PId. If only one instance of the called process exists, then the name of the process followed by **TO** serves the same purpose.

- **Implicit Addressing**

  Implicit addressing doesn't explicitly specify the receiving process instance. Implicit addressing is supported by the **TO** clause followed by the name of the process (instead of the process instance) or by **VIA** clause followed by the name of the signal route or channel. Depending upon the architecture, both **TO** and the **VIA** clauses may or may not uniquely specify the receiving process. Or, the receiver may not be mentioned at all and it may be implicitly defined by the signal sent. If the receiving process cannot be identified uniquely, several process instances become potential receivers. If, however, no receiver is identified, the signal is discarded.

2.2.3 Data part in SDL

The data concept in SDL is based on abstract data types. An abstract data type defines a type of data object by its functional properties, i.e., by a set of operators applied to it. Abstract data types focus on functional properties of data objects. Thus an abstract data type defines the result of the operations applied to a data object, rather than defining how the result is actually obtained. Also, we can use C kind of data types in SDL. Another approach to specify data in SDL is Abstract Syntax Notation One ASN.1[34]. In many ways, the use of data in SDL resembles the use of data in programming languages:

- Expressions evaluate to values.
- A value belongs to a certain data type.
- Variables are used for storing values for later use. A variable can only hold values of a certain data type.
- Predefined data types are available (e.g Integer) and new types can be defined.
- Each data type defines the name of specific values, e.g. the number '7' is a name defined in the Integer data type. The names for the values in SDL are called *literals*. Each data type also defines *operator* performing operations on values, e.g. '+' is an operator for addition of Integers values.

A data type in SDL thus has a strict interface which is the *literals* and *operators* for that data type. Data types are defined in the *text* symbols. The predefined data types are listed in table 2.1.

Name	Literals	*Operators*
Boolean	True.False	*not. and. or. xor.* =>
Char	character enclosed by ' '	<. <=. >. >=. *Num. Chr*
Integer	0.1....	−. +. −. *. /. <. <=. >. >=. *Float. Fix*
Real	0.... 1.... 102.35	−. +. −. *. /. <. <=. >. >=
PId	Null	*none*
Duration	as Real	+. −. >. *. /
Time	as Real	+. −. −. <. <=. >. >=
Charstring	characters enclosed by ' '	*MkString. Length. First. Last.* //. (*index*). *Substring*(*string.start.position.length*)

Table 2.1: Predefined data types in SDL

The way the data types are defined in SDL and especially the way the properties of literals and operators are defined. differs considerably from the programming language approach. Consider. for example the definition of the predefined *Integer* data type:

```
NEWTYPE Integer
LITERALS 0,1,2,3,4,5,6,7,8,9;
OPERATORS
''-'': Integer, Integer -> Integer;
```

```
''+''   : Integer, Integer -> Integer;
''-''   :          Integer -> Integer;
        ''*''   : Integer, Integer -> Integer;
        ''/''   : Integer, Integer -> Integer;
        ''<''   : Integer, Integer -> Boolean;
        ''<=''  : Integer, Integer -> Boolean;
        ''>''   : Integer, Integer -> Boolean;
        ''>=''  : Integer, Integer -> Boolean;
        Float   :          Integer -> Real;
        Fix     :          Real    -> Integer;
         /* here the behavior of the operators is defined */
ENDNEWTYPE;
```

First, literals of the data types are defined. Then follows the definition of operators and finally the behavior of of the operators are defined. Each operator has a *signature*, i.e. an argument data type and a result data type. For example, the "+" operator takes two values of Integer data types as arguments and returns an Integer value as a result. The operators for a data type need not to be distinct, but operators with the same name must have different signatures.

An operator is used by specifying the operator name followed by the arguments enclosed in parenthesis (e.g. "+"(2,3) and Fix(1.3)). However, SDL allows common names for arithmetic and relational operations to be applied in infix form (e.g "+"(2,3) can be written as 2+3). Often, operators or literals with the same name are defined for several data types. For example the "+" operator is defined for both Integer and Real data types. In addition each data type has the equal("=") and not equal("/=") operators defined implicitly. In general there are four ways to specify the *behavior* of operators:

- As informal text

- As *axioms* specifying the equivalence of expressions
- As actions and
- As externally defined data

Axioms are equivalent expressions *which are always true*. For more information on specifying the behavior of operators see [35].

2.3 MSC

The second FDT we will discuss here, as relevant to this work, is called Message Sequence Charts (MSCs). MSCs have evolved from a rudimentary trace language on scratch paper or white boards of telecommunication engineers to a well defined, rich and formal specification language. As we will see in the detail later, MSC is a trace language with vertical lines representing the life lines of processes, process instances or objects. Research on MSCs resulted in the first draft of the language (MSC92) in 1992 in Geneva [36]. Later ITU released method guidelines and the usage of MSCs to capture high level requirements and specifications was suggested in it [37]. Message sequence charts have their semantics based upon process algebra [38]. A draft, suggested methodology and well defined formal semantics resulted in a standard, formal Object Oriented trace language in 1996 [39]. Current release of MSC is MSC2000 [3]. People have started talking about using MSCs in all the phases of the software engineering process [40].

2.3.1 Basic MSC (bMSC)

In the most simple form the message sequence charts are called basic MSCs (bMSC). An MSC is essentially composed of a set of parallel processes communicating with each other asynchronously. A bMSC is shown in the figure 2.14. It specifies two processes P1 and P2, exchanging messages between each other and between environment. Exchanging messages between the environment allows MSC to be modeled as open systems. In basic MSCs, the system environment is represented by a frame symbol that forms the boundary of the bMSC diagram. There is no specific ordering between messages to and from the environment. However, no specific assumptions are made about the behavior of the environment.

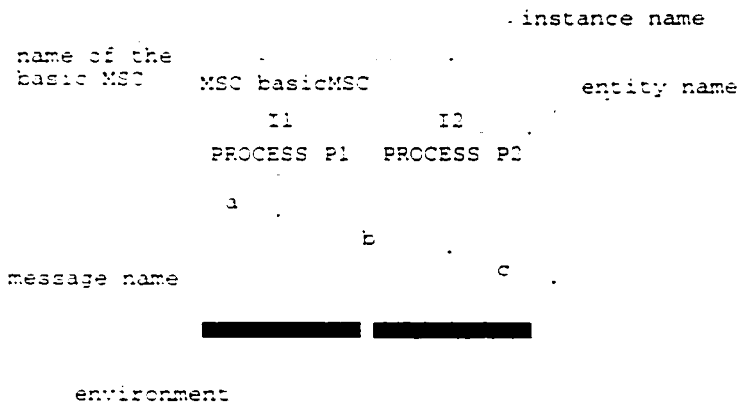


Figure 2.14: A basic MSC (bMSC)

Instances

bMSCs model message exchanges between the instances of processes. Instances are like communicating actors in the environment. They communicate asynchronously with each other. They are shown by vertical lines, also called *life lines*. On these life lines, time travels from top to bottom and from left to right. The start of the instance is modeled by an empty rectangle with the name of the entity or process inside it. The name of the instance is specified at the top of the rectangle. The instance ends with a solid rectangle at the bottom.

Messages

Instances communicate with each other by exchange of messages. Messages are represented by horizontal or sloping arrows associated with message names. The message may also carry date in terms of parameters. In terms of ordering, each sending event precedes the receiving event.

Actions

Actions represent internal activity of the process or instance. They are modeled by a rectangle on the life-line of the process and may contain assignments, expressions or text. Actions correspond to *tasks* in SDL. For example, a bMSC with a *task* is shown in figure 2.15. The action increment a by 2 and assigns it to b. Then the message b is passed to the environment.

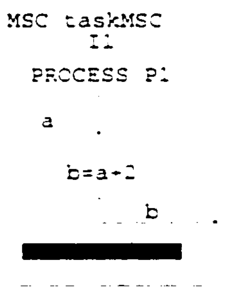


Figure 2.15: A bMSC with task

Conditions

Conditions in MSCs correspond to *states* in SDL. Conditions may be global. non-global or local. To understand consider the example shown in the figure 2.16. A global condition describes the state of the whole system. A non-global condition is used to describe the state of a subset of instances within bMSCs. A local condition describes the private condition of an instance.

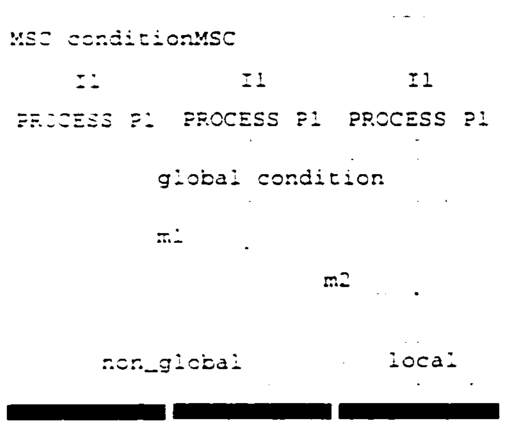


Figure 2.16: A bMSC with global, non-global and local conditions

Inline Expressions

Inline expressions define composition inside the bMSCs. The operators refer to parallel. alternative. iteration. exception and optional regions. A parallel online expression defines the parallel execution of bMSC. No ordering is preserved between events in different sections. In figure 2.17a message *c* happens in parallel to messages *b* and *d*. An alternative online expression defines alternative executions of the bMSC sections. i.e. only one section will be executed in each execution trace. In figure

2.17b after consuming message a, instance $I1$ will send either message B or message C to instance $I2$. An iteration inline expression defines iterative execution of bMSC section. Events in the iteration area will be executed many times (0-infinite). In figure 2.17c instance $I1$ and $I2$ will exchange message a and b at least once. An exception inline expression defines exceptional cases in bMSC. Either the specification in the exception area will be executed or the rest of MSC specification will be executed. In figure 2.17d, instance I1 will send a message b or message c, after consuming message a. An optional inline expression defines an optional execution of bMSC section. Events in the optional area may or may not be executed. In figure 2.17e, after consuming message a, instance $I1$ may send message b before sending message c.

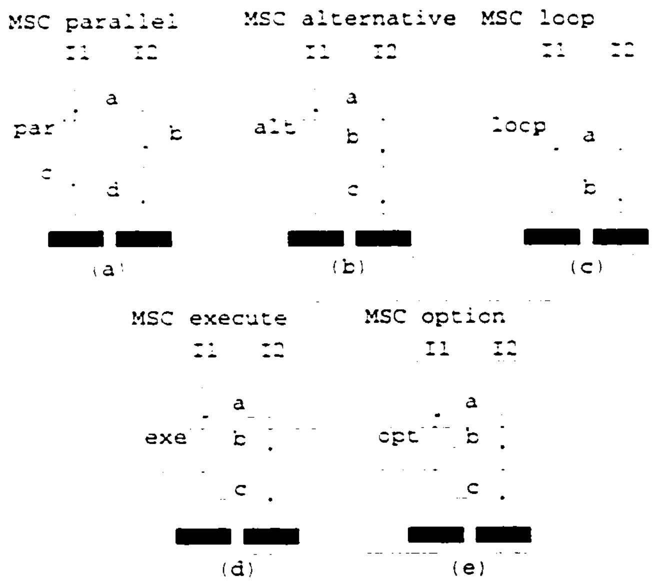


Figure 2.17: Inline expressions in MSC

Timers in MSCs

There is no global time for axis for MSCs. Along each instance axis, the time runs from top to bottom but a proper time scale is not assumed. Timers in MSCs can be used to express timing constraints, i.e., timeout and supervision. Figure 2.18 show the MSC with timers. A timer **set** event is represented by an hourglass connected with the instance axis by a *bent* line symbol. A *sent* event denotes the setting of the timer. A timer **reset** event is represented by a cross connected with the instance axis by a *bent* line symbol. A reset timer event denotes resetting the timer. A **timeout** event is represented by an arrow, which is connected to the hourglass symbol, and the arrowhead points to the instance axis. A *timeout* event denotes the expiration of the timer. For each timer setting event, a corresponding time-out and/or timer reset has to be specified and has to follow it in order. However, corresponding timer events may be split among bMSCs in case the whole scenario comprises of a combination of several bMSCs.

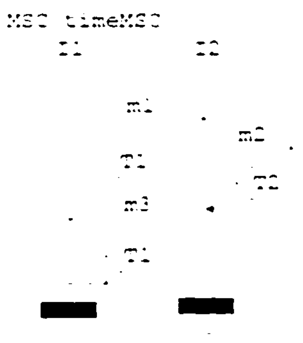


Figure 2.18: An MSC with timer events

Coregions

Events in MSCs are strictly ordered. A sending event always proceeds a receiving event. Coregions are introduced to relax the order in some parts of the instance axis. Within a coregion, the specified communication events are not ordered. This feature is useful because the order of some of the events may not have been decided 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 2.19. The message *a* occurs before *b* and *c*, but there is no order between the occurrence of *b* and *c*.

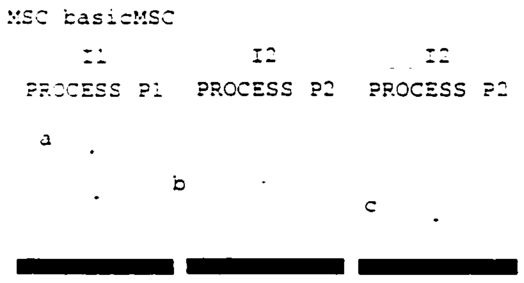


Figure 2.19: A bMSC with coregion

2.3.2 HMSCs

High-level Message Sequence Charts (HMSCs) provide higher level of abstraction than MSCs. HMSCs actually give a system overview through a composition of bMSCs. They provide four operators to connect bMSCs which are *sequential*, *alternative*, *iterating* and *parallel* operators. Furthermore, HMSCs allow combining HMSCs within HMSCs in hierarchical specification. For more details on HMSCs, refer to [3].

Chapter 3

A Methodology for Automated Enrichment of SDL with MSCs

3.1 Introduction

In this chapter we give an overview of our approach of enriching SDL specifications with MSCs. First of all, informally, what is *enrichment* or *extension* of a system? *Users* interact with the *System* in accordance with the overall defined behavior of the system. This may be called the interface to the external world for the system or it may also be called the *external* behavior of the system. In addition to this, internal components inside the system, e.g. packages, blocks and processes interact with other component through their own well defined interfaces. For the external user, this functionality might be transparent but it is inevitably important for the internal functionality of the system. This may be called *internal* behavior of the system. The external behavior of a system might need to be enriched by adding more functionality to accommodate new requirements. This necessitates the enrichment of the behavior of the internal components of the system as well. In general, the complete system has to be extended. Now, one way of dealing with this could be to build a new system from scratch, which not only has the old behavior but the new one too. But this means flushing all the previous resources and efforts down the drain. Other way is to take the old system and add the new requirements to the existing system. Technically it is called the *maintenance* phase of software engineering process. But enriching or extending the existing system is not straight forward and has lot of complexities involved. A general concern is whether the new functionality added into the system disturbs or corrupts the old functionalities. What if the new enrichment *invalidates* some of the properties of the existing system?

We cannot underestimate the importance of *enrichment* or *extension* in the maintenance phase. New requirements are usually specified in an informal dialect of MSCs. As shown in the figure 3.1. the ad-hoc approaches take the old system's code and the new requirements in *informal* MSCs and enrich the code *manually*. This is a very time and resource consuming process. In addition to that the *enriched* system cannot be verified and validated. There is no guarantee that the old behavior will be preserved in the new system. The new system may introduce a new *non-determinism* which may invalidate the old behavior.

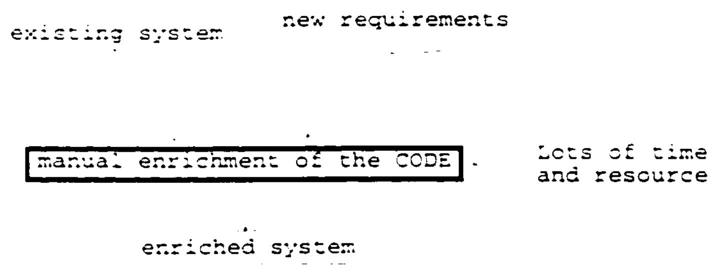


Figure 3.1: Adhoc approaches of System Enrichment

We focus on the idea of *maintaining the specifications* [17]. and develop an *automated* approach which not only supports enrichment of the existing SDL specifications with MSCs, but also promotes *incremental development* of SDL with MSCs. Powerful tools exist [31, 32] which support automated code generation. Hence, contrary to the ad hoc approaches (figure 3.1), our approach is shown in figure 3.2.

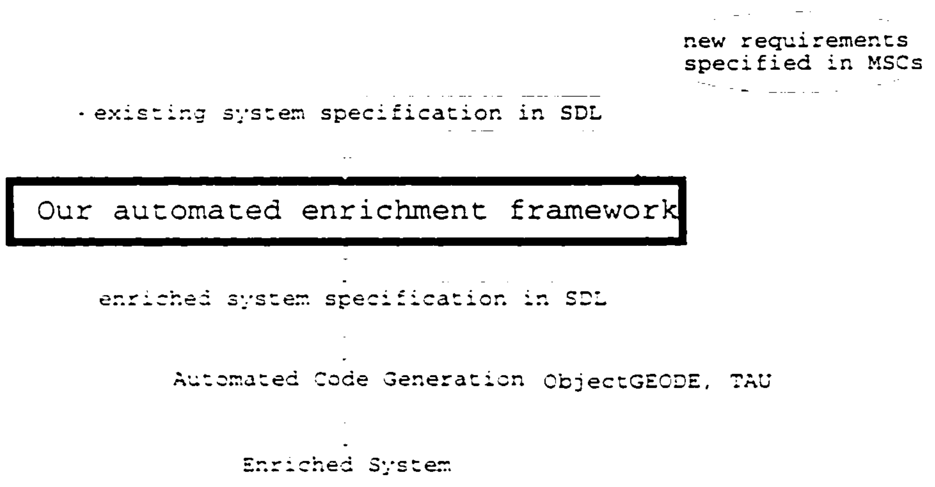

new requirements specified in MSCs
existing system specification in SDL
Our automated enrichment framework
enriched system specification in SDL
Automated Code Generation ObjectGEODE, TAU
Enriched System

Figure 3.2: Our automated approach for enrichment

Our approach consists of three phases, as shown in figure 3.3, i.e. a *pre-phase*, a *MSC2SDL phase* and a *post-phase*.

3.2 Pre-Phase

This is the first phase of our *automated enrichment approach*. It handles the enrichment of SDL architecture using MSC as a new use-case. The pre-phase tool can add

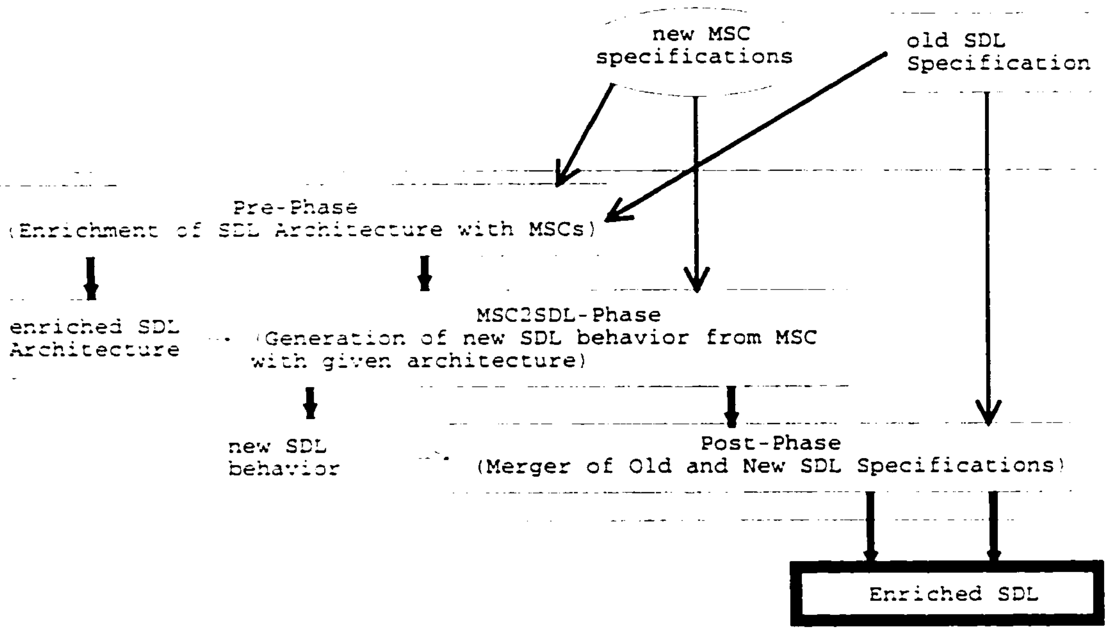

new MSC specifications
old SDL Specification
Pre-Phase
(Enrichment of SDL Architecture with MSCs)
enriched SDL Architecture
MSC2SDL-Phase
(Generation of new SDL behavior from MSC with given architecture)
new SDL behavior
Post-Phase
(Merger of Old and New SDL Specifications)
Enriched SDL

Figure 3.3: Getting enriched SDL using our automated enrichment approach

new processes in existing blocks or new blocks, new signals between existing processes through existing links or through newly added links and it also enriches the declarations and signal lists accordingly. Algorithms and further details about the architectural enrichment of SDL specifications with MSCs is discussed in Chapter 4. As shown in the figure 3.3, there are two inputs to the pre-phase tool i.e. the *new* MSC and the *old* SDL specification. The tool does not disturb the original input specifications and produces a resulting SDL specification with *enriched* architecture.

3.3 MSC2SDL-Phase

This is the central phase of our approach. The MSC2SDL tool [23] is used to generate new SDL specifications from the new MSC and the *enriched* architecture, generated in the *pre-phase*. MSC2SDL tool is a robust tool covering not only most of the constructs of the MSC language but also the HMSCs. It handles *actions*, *timers*, *coregions* and most of the other constructs of the MSC language. The MSC2SDL tool also detects *unspecified receptions*, *deadlocks* and *distributed choice* etc. For a detail on MSC2SDL algorithms and tools see [24].

3.4 Post-Phase

The MSC2SDL phase generates new SDL behavior, which is actually the SDL specification corresponding to the new functionality to be added to the old specification. In the post-phase, our framework uses the approach of merging the *old* and *new* behavior to get the resulting, enriched behavior. The detailed rules and algorithms of merging the SDL behaviors are discussed in Chapter 5.

Thus, our approach consists of three *sequential* and *connected* phases to get an enriched SDL system from an existing system. The *pre-phase* is discussed in Chapter 4 and the *post-phase* is discussed in 5. The MSC2SDL phase is discussed in [24].

Chapter 4

Enrichment of SDL Architecture with MSCs

The architecture in SDL (also called the structural part of SDL) serves to allow modularity in the design specifications. Modularity not only promotes clarity, simplicity but also allows re-usability of specifications. Main entities of the SDL architecture, as discussed in Chapter 2 are System, Block, Process, Channels/Signalroutes and Signals.

The focus of this chapter is to consider the *automation* of enrichment of SDL architecture with MSCs. As mentioned in Chapter 3, there are three key issues in this respect, as following.

- Addition of new Processes into existing or new blocks
- Addition of new Links (Channels/Signal Routes) or enrichment of existing links
- Addition of new Signals

4.1 Mapping MSCs on SDL architecture

In this section we investigate the relationship between MSC and the SDL architecture. Consider a simple MSC in Figure 4.1. It shows two process entities *client* and *server*. The MSC shows the connection establishment primitives and PDU exchanges of a generic protocol of transport layer. The client entity receives a *connection request* (CONreq) message from a higher layer. The client sends a *request PDU* (REQ_PDU) to the server entity. The server sends a *connection indication* message to its higher layer entity. Let us assume the higher layer entity is willing to accept the connection. So it sends a *connection response* (CONresp) message to the server entity. The server then sends a *response PDU* (CONresp) back to the client and the client sends a *connection confirmation* (CONcnf) message to its user entity. Specifying this in MSC is quite straightforward. But when we want to build detailed and concrete design specifications in SDL. we have to map MSC to SDL. Existing research work on MSC to SDL [21. 22] and [23] considers the architecture to be already present before the translation. However. when we are considering the enrichment. we must do the architectural enrichment before the behavioral enrichment (discussed in the methodology in Chapter 3). This means creating new architecture or enriching the existing architecture based upon the new MSC.

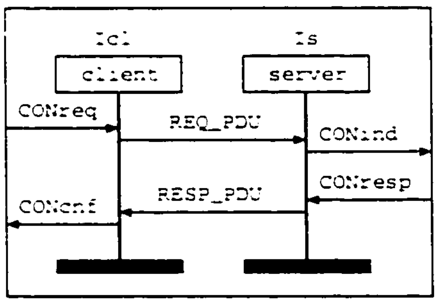


Figure 4.1: A simple MSC showing a connection scenario.

A *process instance* maps to a *process* or a *process type* in SDL. Whenever there is a message exchange between two process instances in the MSC, there must exist a corresponding *link* between those processes in the SDL architecture. The *signals* passed through that link must be specified at the proper and correct end of the link in the SDL architecture. The simple MSC of Figure 4.1 is shown to be mapped upon an SDL architecture in Figure 4.2.

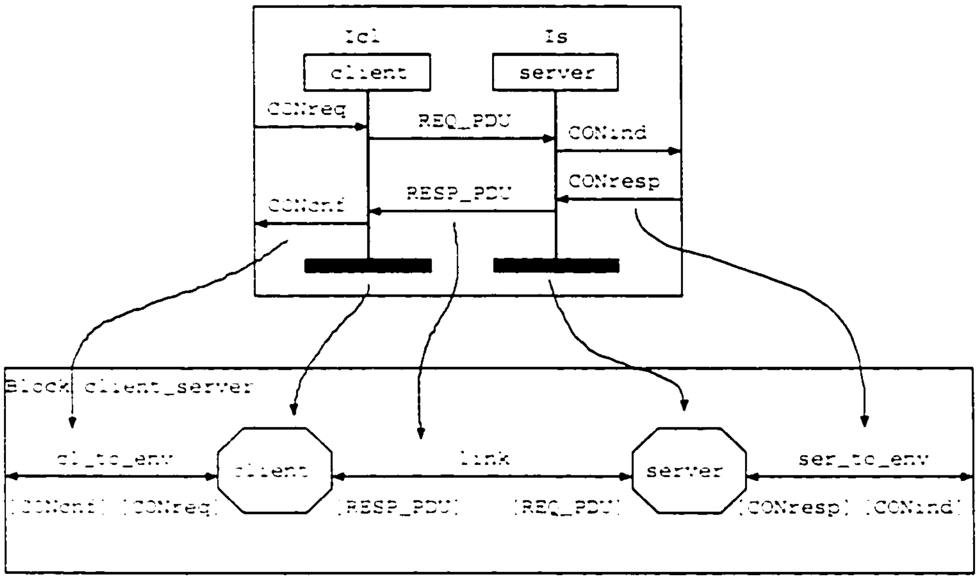


Figure 4.2: MSC to SDL architecture mapping.

Before actually proceeding to the details, we define a few terms for the sake of architectural enrichment and its automation. Two processes in SDL Architecture are said to be **friends** of each other if they are connected through one or more *channels* or *signalroutes*. Each process maintains a *collection* of its friends. So for each process P, we may find out whether another process Q is its friend or not by searching the *friends* collection of P or that of Q. Furthermore, each process maintains a collection of links (channels or signalroutes) between itself and its friends. Each link in the list is qualified with two additional attributes. which are **myEnd** and the **peerEnd**. For a channel *Ch* connected to the process *P*. *myEnd* is the end of channel connected to itself. while *peerEnd* is the end of the link connected to its *friend* process. For each process connected to each link. the *signals* specified to be received are stored in the collection **myEndSignals**. and the signals specified to be sent are stored in the collection **peerEndSignals**. For an elaboration see Figure 4.3. We use these terms to build algorithms for automated enrichment of SDL architecture with MSCs.

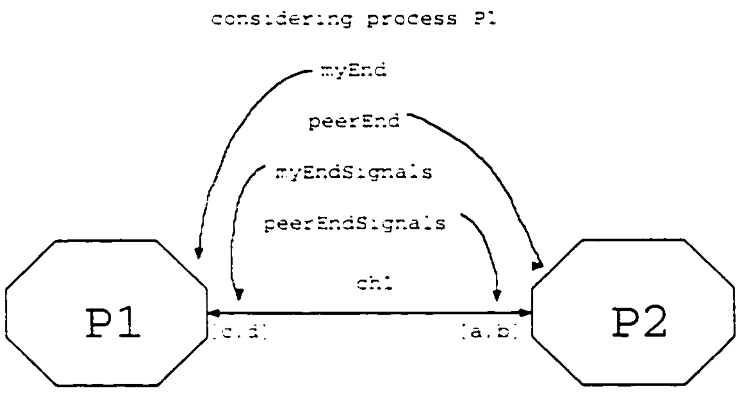


Figure 4.3: Definition of the terms for Architectural Enrichment.

4.2 Adding new Processes into existing or new Blocks

As discussed in Chapter 2 MSCs specify the message exchanges between processes (and process instances). Messages are also handled to and from the environment. MSC do not have structural (architectural) constructs as supported by SDL [39]. thus giving them more abstractional power. SDL specifications on the other hand are more detailed as compared to MSCs. Considering the SDL architecture. it represents which processes are located in which blocks and how they communicate with each other. An MSC may specify message exchange between *old* process(es) and *new* process(es). The automation algorithm should be able to add these new processes into the architecture. When an engineer is drawing MSCs. he probably has not decided yet about the structural modularity of the process instances in these use cases. This structural decomposition is mostly done in the detailed design of the project. There are many design decisions and rationales involved from requirements and analysis stage to design stage. Most of this is an Engineer's experience and expertise. so it cannot be automated. We. however. can consider the automated enrichment of SDL architecture with MSCs if our tools proceed with the user's interaction. This is exactly how we develop our algorithms and tools. The first step in our algorithm is to build a *SDL Structure table* corresponding to the *old SDL* specification. The second step is to build a *MSC Structure Table* corresponding to the *new MSC* specification. These structure tables are the direct representations of data structures inside the code. To elaborate these structure tables. we consider a small. vague. example.

Figure 4.4 shows a simple SDL architecture. It has two blocks. $B1$ and $B2$. Block $B1$ contains processes $P1$. $P2$ and $P3$. Block $B2$ contains process $P4$. There inter-connection is evident from the figure. We build the *SDL Structure Table* for each block in the SDL architecture. While constructing the table. the environment

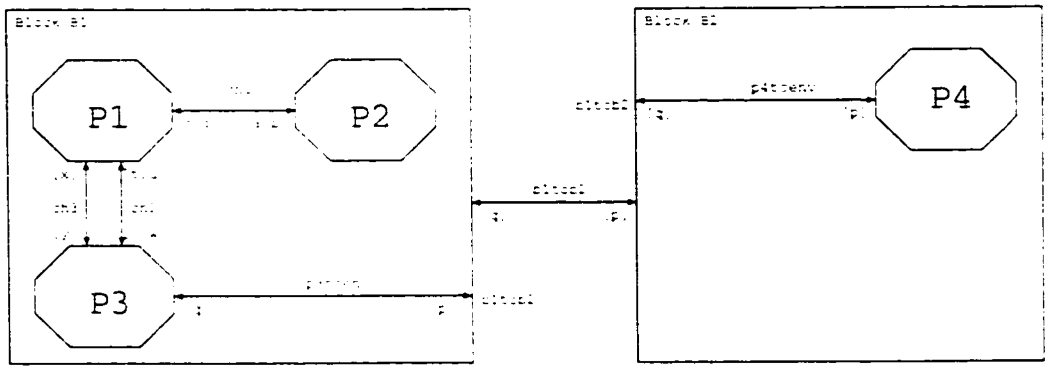


Figure 4.4: A Simple Architecture to construct SDL Structure Table.

of the block is treated like a process entity. The *SDL Structure Table* for block *B*1 in Figure 4.4 is shown in Table 4.1. And the *SDL Structure Table* for block *B*2 is shown in Table 4.2. As can be seen that if the *friends* are connected through multiple channels, a boolean *meExist* is set accordingly. As already mentioned, these structure tables map directly to data structures in our pre-phase (see Chapter 3 tools.

Process	Friend	meExist	Link	myEndSignals	peerEndSignals
P1	P2	F	ch1	c,d	a,b
	P3	T	ch2	t,u	v,w
			ch3	x	y
P2	P1	F	ch1	a,b	c,d
P3	P1	T	ch2	v,w	t,u
			ch3	y	x
	Env	F	p3toenv	q	p
Env	P3	F	p3toenv	p	q

Table 4.1: SDL Structure Table for Block B1

Process	Friend	meExist	Link	myEndSignals	peerEndSignals
P4	Env	F	p4toenv	p	q
Env	P4	F	p4toenv	q	p

Table 4.2: SDL Structure Table for Block B2

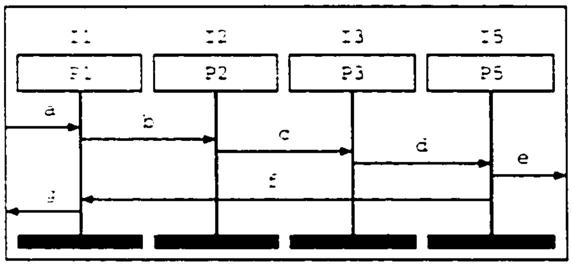


Figure 4.5: New MSC.

Now, consider a *new* MSC specification shown in Figure 4.5. Note that it has processes $P1$, $P2$, $P3$ and $P5$ in it. Processes $P1$, $P2$ and $P3$ are present on the old SDL architecture, shown in Figure 4.4. Process $P5$ is not present, so during the automated enrichment of the SDL architecture, it must be added. But it is not that straightforward as there are many issues involved here. Before discussing these issues, let us construct the *MSC Structure table* for this MSC in Figure 4.5. It is shown in Table 4.3.

Now, let us discuss adding process $P5$ to the SDL architecture. It is possible that the user intended to add $P5$ in block $B1$ or block $B2$, or into a new block. But for automated translation, concrete decisions are required. Hence our algorithms

Process	Friend	Message	Sent/Received
P1	P2	b	sent
	P5	f	received
	Env	a	received
		g	sent
P2	P1	b	received
	P3	c	sent
P3	P2	c	received
	P5	d	sent
P5	P3	d	received
	P1	f	sent
	Env	e	sent

Table 4.3: MSC Structure Table for the new MSC

prompt to request the desired block from the user. The user is given a list of existing blocks to choose from or he may also choose to add the process in a *new* block. In the later case, he is asked for the name of the new block and then it is connected with the existing blocks in the architecture. As we will see later on, user interaction plays a key role in the automated enrichment of the architecture. We are now able to summarize the algorithm to add a new process in the architecture.

Algorithm 1: description

For every process in the MSC Structure Table, except environment (Env), find a corresponding process in the SDL Structure Table, for each existing block, in the old architecture. If the process is found in any SDL Structure Table of any SDL block of the old architecture, no new process is to be added. If not, prompt the user by giving him the list of SDL blocks present in the old architecture, and ask for the block where the new process is to be added, also giving him a possibility to add into a new block. Finally add the new process from the MSC Structure Table into the SDL Architecture. in the desired block or a new block and then update the SDL Structure Table of that block accordingly. Based upon this statement. we can state the algorithm as following:

```
Algorithm 1:
for each process p in MSC structure table
   search p in SDL structure table of each block
      if (p found)
         noNewProcessAdded();
      else if (p not found)
         blockList = getListOfExistingBlocks();
         /**
          * show the list of existing blocks to user
          * plus the option to chose a new block
          */
         showList(blockList);
         /**
          * get the choice of user
          */
         selectedBlock = getChoice();
         if (blockList contains selectedBlock)
            addProcess(selectedBlock);
         else if (blockList does not contain selectedBlock)
            /**
             * ask name of new block
             */
            name = getName();
            /**
             * create a new block
             */
            addBlock(name);
            /**
             * add the new process in this block
             */
            addProcess(name);
            /**
             * connect block to existing blocks
             */
            connectBlocks();
         endif;
      endif;
```

```
    /**
     * update SDL structure tables
     */
    updateSdlTable();
  endsearch;
endfor;
```

Say the user selects to add the new process *P5* in the block *B2*. Figure 4.6 shows the new process added. It is important to note that whenever a new process is added, it always has to follow the addition of new *links* (channels or signal routes) to other processes or environment. We discuss the addition of new Links in the next section.

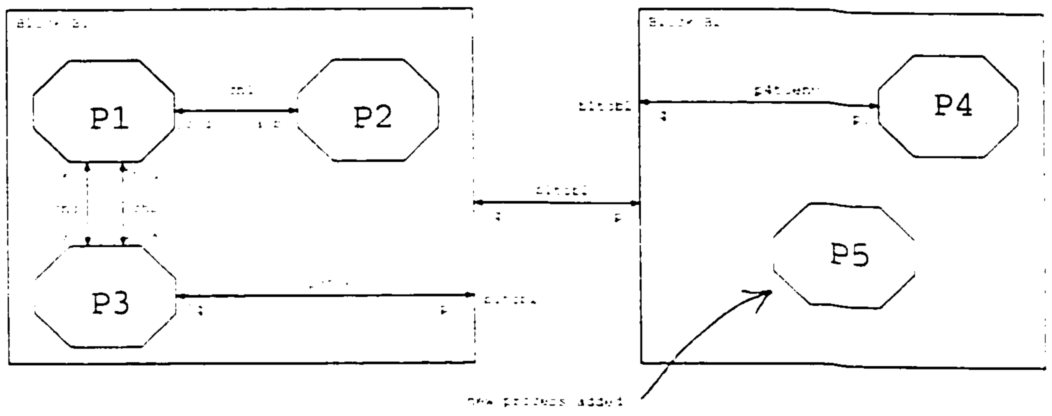


Figure 4.6: SDL Architecture with the new process added.

4.3 Addition of new Links (Channels/Signal Routes)

As we have already discussed in Chapter 2. communicating entities (processes) and container entities (blocks) are interconnected through *links*. A link may be a *channel* or a *signalroute*. The former models a communication link with some non-deterministic delay while the later models the communication link with *no* delay. Addition of new process(es) always follow addition of new links to other processes or environment. However. the addition of new links might also be required even when no new processes are added to the architecture. This is the case in which the *new* MSC specifies message exchanges between two *existing* processes. Since the processes are already existing in the specification. so they must have already been registered as *friends* in the SDL Structure Table. But the user may want to exchange the new messages using a new link between them. Also. the new MSC may specify message exchanges between new processes and environment or new messages between existing processes and environment. In all these situations new links are required to be formed. Considering MSC in Figure 4.5. after addition of new process $P5$. in block $B2$. it has to be made *friend* with the existing processes. The MSC Structure Table 4.3 shows $P3$. $P1$ and Env in the friend's list of $P5$. Since $P5$ is a newly-added process. a link has to be made to each of the $P3$. $P1$ and Env. Our algorithm prompts the user for each link and makes the link accordingly. Further. the algorithm also checks for the *new* message exchanges between between *existing* processes as specified by the MSC. For each new message. the algorithm asks the user whether he wants to exchange the *new* messages through existing links or through new links and then acts accordingly. The updated architecture after the addition of $P5$ is shown on the Figure 4.7.

We give a choice to the user that he may add the new processes anywhere in any block or even in *new* blocks. This liberty makes the algorithms a little bit more complicated. In fact. the more choices we give to the user. the harder the

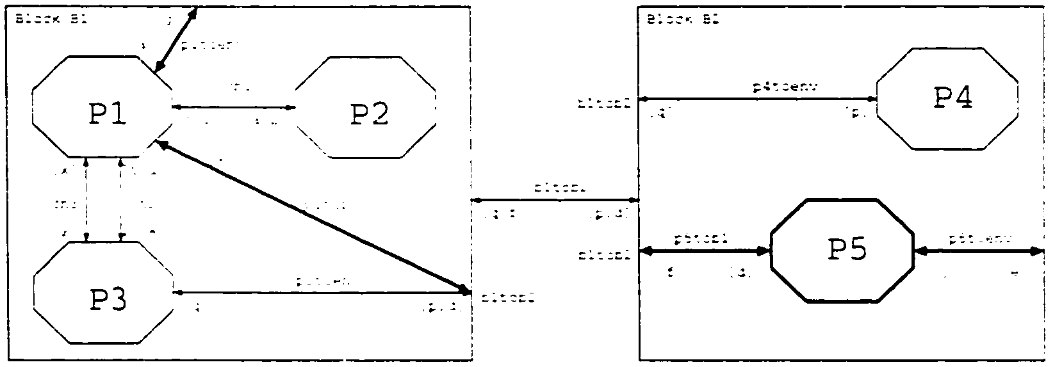


Figure 4.7: SDL Architecture, after addition of new process and links.

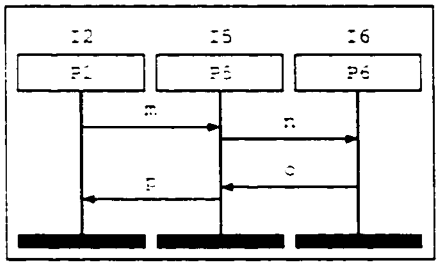


Figure 4.8: An MSC with two new processes to be added.

algorithms become. To elaborate we consider another simple example. Figure 4.8 shows an MSC with two new processes to be added, namely, $P5$ and $P6$. Adding $P5$ and $P6$ to the old architecture of Figure 4.4 gives rise to three different scenarios.

Say the user selects to add $P5$ in block $B1$ and $P6$ in block $B2$. This is similar to the example already discussed. When $P5$ is added into block $B1$, it has to be made *friend* with already existing process $P2$ through an explicitly specified channel. And after adding $P6$ to block $B2$ it has to be made *friend* with the previously added process $P5$. Since $P5$ is another block, two things must be done.

Firstly, our algorithm adds an explicit channel from $P6$ to the *Env* of the block. Secondly, it asks the user whether he wants to propagate the new signals through an existing link between $B1$ and $B2$ (if one exists) or through a new link. In the later case, the algorithm adds an explicit link between $B1$ and $B2$.

The second scenario is slightly more complicated. Say the user adds $P5$ in block $B2$ and $P6$ in the block $B1$. $P5$ has to be made *friend* with $P2$ in the block $B1$. To establish this friendship, an explicit link is made between $P5$ and the environment of the block $B2$. The user explicitly selects the link between $B1$ and $B2$. And then an explicit link is made from $P2$ to the environment of the block $B1$. Then, when $P6$ is added in the block $B1$, it has to be made *friend* with the process $P5$ in block $B2$. As before, the algorithm adds an explicit link between $P6$ and the environment of the block $B1$, lets the user select an explicit link between the blocks $B1$ and $B2$ and then adds an explicit link between $P5$ and the environment of the block $B2$.

The third scenario involves the possibilities when the user selects to add both $P5$ and $P6$ in either block $B1$ or block $B2$. This case is relatively simple because no inter-block connecting is involved here. All the links are to be formed inside the block where the two processes are added. Considering that the user has selected to add the two processes in $B1$, $P5$ is added first and connected to $P2$ through an explicit link. Then $P6$ is connected to $P5$ through an explicit link. It is important to note that as soon as an enrichment is made in the architecture, the corresponding SDL Structure Table is updated accordingly to reflect the change in data structures in the memory. Next we try to summarize the algorithm for adding the new Links into the architecture.

Algorithm 2: description

For every new process added in the SDL architecture, in the Algorithm 1, find its friend(s) in the SDL Structure Table. For each friend, if it lies in the same block as to which the new process is added. connect the new process to its friend with an explicitly specified link. If the friend lies in a different block. then first connect the newly added process to the environment of the block through an explicitly specified link and then connect the two blocks through an explicitly specified link.

```
Algorithm 2:
for each newly added process p in  a block b
   /**
    * get a list of process p's friends in
    * the existing SDL Structure Table
    */
   existingFriends = getFriends(SDL Structure Tables);
   for each friend in existingFriends
      /**
       * get the block where the existing friend
       * is present
       */
      blockOfFriend = getBlock(friend);
      /*
       * check if the process is added
       * in the same block as where its
       * existing friend is present
       */
      if ( b == blockOfFriend )
         /**
          * connect the friends together through
          * an explicitly specified link
          */
         connectPtoFriend();
      /**
       * process p may have been added in a
       * block different from blockOfFriend.
       * This also includes a new block
       */
      else if ( b != blockOfFriend )
         /**
          * connect p the the env of block b through
          * an explicitly specified link.
          * if p is already connected to the env
          * then give a choice to use the existing link
          * or new link
```

```
             */
            completeConnectionToEnv( p );
            /**
             * if block b is not connected to blockOfFriend
             * then complete connections to the blockOfFriend. If
             * it is already connected, then enrich the existing
             * link, or add a new explicitly specified link
             */
            completeInterBlockConnections(b,blockOfFriends);
         endif;
      endfor;
   endfor;
```

4.4 Adding new Signals

Signals are specified at the ends of the links. So each time a new link is formed by applying Algorithm 2. signals have to be added to it to complete the architecture. MSCs specify the exchange of messages between processes. but they don't specify which link to follow. MSCs may specify exchange of new messages between processes which are already connected through one or more links in the existing architecture. But the intentions of the designer might be to pass the new signals through a new link between the processes. In such a case a new link may be formed according to Algorithm 2. Otherwise an existing link is used to add the new signals. In either case new signals are added with the help of *myEndSignals* and *peerEndSignals* of the SDL Structure Table and the *sent/received* column of the MSC Structure Table. relevant to the processes under enumeration. A *sent* signal from $P1$ to $P2$ is added to the $peerEnd$ of the link between them in the *peerEndSignals* collection. A received signal on the other hand is added at the $myEnd$ in the $myEndSignals$ collection. Thus we are able to phrase the algorithm for $AddingnewSignals$.

Algorithm 3: description

For each message m between *friend* f of a process p. in the MSC Structure Table. if it is a sent message from process p to a friend f. find out if a corresponding signal exists in the peerEndSignals collection of the corresponding friend f of the corresponding process p. in the SDL Structure Table. If it does. we don't do any addition of signals. If it does not. find out the collection of channels between friend f and the corresponding process p. Ask the designer explicitly. whether he wants to carry the new signal m through one of those existing channels or through a new one. In case he selects one of the channels from the list. add the signal m to its *peerEndSignals* and update the SDL Structure Table. This process is repeated for each signal between each process and its friends in the MSC Structure Table. In case the message is qualified to be *received* in the MSC Structure Table. just replace the *peerEndSignals* to *myEndSignals* in the above definition.

```
Algorithm 3:
for each message m between friend f of process p in MSC Structure Table
   if ( messageType( m ) == sent )
      peerEndSigList = getPeerEndSigList(p,f);
      if ( peerEndSigList contains m )
         /**
          * no signal enrichment required
          */
      else if ( not peerEndSigList contains m )
         channelCollection = getChannelCollection(p,f);
         /**
          * give a choice to the user to either
          * select one of the links from the
          * list or a new link
          */
         choice = selectChannel(channelCollection);
         if ( choice is existing link )
            enrichPeerEndSignals(choice,m);
         else if ( choice is new link )
            addNewLink(chice,p,f);
            enrichPeerEndSignals(choice,m);
         endif;
         endif;
      else if ( messageType( m ) == received )
         myEndSigList = getMyEndSigList(p,f);
         if ( myEndSigList contains m )
            /**
             * no signal enrichment required
             */
         else if ( not myEndSigList contains m )
            channelCollection = getChannelCollection(p,f);
            /**
             * give a choice to the user to either
             * select one of the links from the
             * list or a new link
             */
            choice = selectChannel(channelCollection);
            if ( choice is existing link )
               enrichMyEndSignals(choice,m);
            else if ( choice is new link )
               addNewLink(chice,p,f);
               enrichMyEndSignals(choice,m);
            endif;
         endif;
      endif;
   endfor;
```

The SDL architecture enrichment algorithms run sequentially, from Algorithm 1 followed by Algorithm 2 to Algorithm 3. Our architectural enrichment tool implements them completely. We will show some examples in Chapter 6.

Chapter 5

Merging SDL Specifications

5.1 Introduction

Old and new specifications can be merged to get an enriched specification[41]. Enrichment or extension of an existing system has always been an area of prime interest of researchers [42. 41. 43]. With the availability of CASE tools [31. 32]. which can even generate code automatically. the need and significance of automated specification merging algorithms and tools. to get automated enrichment. have increased.

To discuss the behavioral merger. consider a FSM specification of a simple vending machine. *vendor1.* which accepts a *coin* and delivers a *coke* after the user presses the *coke_button.* This machine is shown in figure 5.1. *s1* is the initial state of *vendor1.*

The machine in figure 5.1 allows the user to get a coke after entering a coin. Now. consider that we want to have a machine which accepts a coinand then the user can choose to get a coke or a pepsi. Figure 5.2 models a machine which after accepting a coin gives a pepsi when user presses the *pepsi_button.* Figure 5.2 can be either be viewed as a separate machine or as a new functionality to the existing machine in figure 5.1. Considering figure 5.2 as a new functional requirement to the existing machine. an *enriched* machine can be obtained by *merging* the machines in

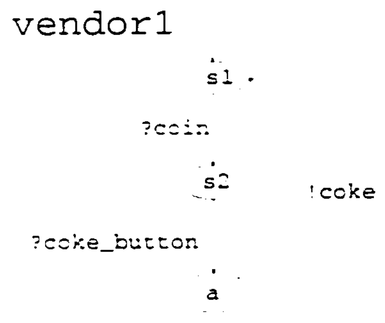


Figure 5.1: A simple vending machine

figure 5.1 and figure 5.2. The enriched machine should preserve all the behavior in the old machine(s). That is to say, it should deliver a coke when the user inserts a coin and presses the *coke_button*. Also, the enriched machine should not have any new non-determinism introduced in addition to the one already present in the old machine. This is because a new non-determinism might *invalidate* the old behavior. For this example, the enriched machine, after merging figure 5.1 and figure 5.2 is shown in figure 5.3.

The aim of this chapter is to formally define behavior preservation in SDL and introduce extension relations, which define extension of SDL specifications ensuring behavioral preservation and forbidding an introduction of new non-determinism in an enriched SDL specification. Using these, we would define rules for merging SDL specifications on *transition by transition* basis. Informally, we define a transition from a *current state* to a *next state* upon accepting an input signal (trigger), and delivering a sequence of output signals. So according to this definition, if in two specifications S1 and S2, two transitions have the same *current state*, same *input* trigger, the same sequence of output signals and the same *next* state, then we say that the transitions are equivalent. Merging equivalent transitions result in a

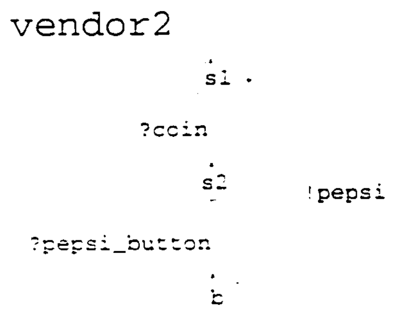


Figure 5.2: Another simple vending machine with different functionality

similar transition. But there are many other scenarios as we will see later. The rules established in this chapter are guidelines for our post-phase tool and for SDL behavioral merger in general.

5.2 Behavior preservation in SDL

Behavior preservation in SDL has been defined in [17]. Informally, an SDL specification S2 *preserves* or *extends* the behavior of another SDL specification S1, if and only if S2 exhibits all the behaviors of S1 without any new non-determinism. In other words, S2 shows all the traces (sequence of interactions with the environment) of S1 without any new "surprise" to the environment relatively to these traces. S2 can be used where S1 can be used. S2 may exhibit more behaviors than S1. For instance, S2 may take into account new input signals and define the behavior of these signals. S2 may also react with more output signals for already defined input signals.

The conformance and equivalence relations defined in [44, 45] are defined

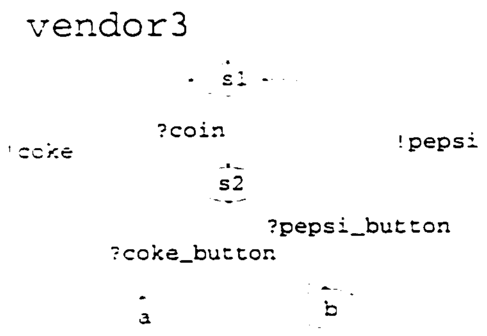


Figure 5.3: Enriched vending machine

upon blocking semantics. In other words, they are based on a system where a system or a process blocks for a non-defined interaction with its environment. These relations cannot be applied for SDL, which has non-blocking semantics. An SDL process communicates with its environment in an asynchronous manner. The environment sends signals through channels and these signals are *queued* in an input queue of the process. At a given state, an unexpected input signal, i.e., a signal not explicitly specified as input, is just removed from the input queue of the process. Furthermore, SDL processes may *save* signals for future consumption.

In order to define *behavior preservation*, or the *extension relation*, in the context of SDL, a few notions are introduced in [17], without referring to any underlying semantic model for SDL. These definitions can be applied under any extended input/output communicating finite state machine model for SDL. For the sake of convenience, we repeat these definitions here.

- For each process P, a state s, $Input(s)$ is defined as a set of input signals for which an explicit transition is defined from s. In this definition we view the SDL keyword *none* as an input signal.

- For a process P. a state s. $Save(s)$ is defined as the set of input signals saved in the state s.

- For a process P. a state s. an input i. $Output(s.i)$ is defined as the set of sequence of output signals in the transitions initiated by the input signal i from the state S. The "." is used for concatenating sequences.

- For a process P. a state S. an input i. and a sequence seq in $Output(s.i)$. $NextState(s.i.seq)$ represents the set of states that can be reached from s with input i and output sequence seq. Notice that in SDL. with a given input signal and a sequence of output signals we can reach different states. This kind of non-determinism can be modeled easily with SDL keyword *any*.

- A sequence *seq2* of (output) signals *extends* another sequence *seq1* of (output) signals. if and only if *seq1* is a prefix of *seq2*. For instance. sequence *0.1.1.0 extends* sequence *0.1*

Informally. state s_2 extends state s_1. when s_2 has all the explicit inputs of s_1 (and more) and for each of them. for each state $s_1\prime$ we can reach from s_1 and an output sequence $seq1$. there is a state $s_2\prime$ we can reach from s_2 and an output sequence $seq2$ with $s_2\prime$ *extends* $s_1\prime$ and $seq2$ *extends* $seq1$. and for each state $s_2\prime$ we can reach from s_2 and an output sequence $seq2$. there is a state $s_1\prime$ we can reach from s_1 and an output sequence $seq1$ with $s_2\prime$ *extends* $s_1\prime$ and $seq2$ *extends* $seq1$. and a signal saved in s_1 is also saved in s_2. The extension between processes is defined as the extension between their initial states[17].

Definition 1(State Extension)

State s_2 extends state s_1. if and only if

- $Input(s_1) \subseteq Input(s_2)$.

- $Save(s_1) \subseteq Save(s_2)$.

- $\forall i \in Input(s_1). \forall seq1 \in Output(s_1, i). \exists seq2 \in Output(s_2, i)$. such that
  - $seq2$ *extends* $seq1$. and
  - $\forall s_2\prime \in NextState(s_2, i, seq2). \exists s_1\prime \in NextState(s_1, i, seq1)$ such that $s_2\prime$ *extends* $s_1\prime$. and
- $\forall i \in Input(s_1). \forall seq2 \in Output(s_2, i). \exists seq1 \in Output(s_1, i)$. such that
  - $seq2$ *extends* $seq1$. and
  - $\forall s_2\prime \in NextState(s_2, i, seq2). \exists s_1\prime \in NextSate(s_1, i, seq1)$. such that $s_2\prime$ *extends* $s_1\prime$.

Definition 2(Process Extension)

A process P_2 *extends* a process P_1. if and only if
$Initial(P_2)$ *extends* $Initial(P_1)$. where as
$Initial(P_i)$ represents the initial state of P_i. with $i = 1, 2, ...$

To illustrate the extension relation between processes. consider the example given in figure 5.4. Both processes P_1 and P_2 are non-deterministic. As we can see. state s_4 in P_2 extends state s_1 in P_1. state s_5 in P_2 extends s_2. Now. if we compare state s_0 and s_3. we see that $Input(s_0) = a$ and $Input(s_3) = a$. $Output(s_0, a) = Output(s_3, a) = 0.1, 1$. $NextState(s_0, a, 0.1) = s_1$. $NextState(s_3, a, 0.1) = s_4$. 0.1 extends itself and s_4 extends s_1. $NextState(s_0, a, 1) = s_2$. $NextState(s_3, a, 1) = s_5$. 1 extends itself and s_5 extends s_2. In state s_0. process P_1 does not save any signals. while process P_2 saves signal b. We can conclude that s_3 extends s_0. Therefore. P_2 extends P_1.

Now. if we consider the second example given in figure 5.5. we can see that the process P_4 is not an extension of P_3. The reason is that P_4 introduces a new non-determinism in state s_3. Indeed. after consuming signal a. the output can non-deterministically be 0.1 or 1 for P_4. Process P_3 always produces 0.1 after consuming a. Of course process P_4 has more behaviors than process P_3. but because of this new non-determinism. P_4 is not an extension of P_3.

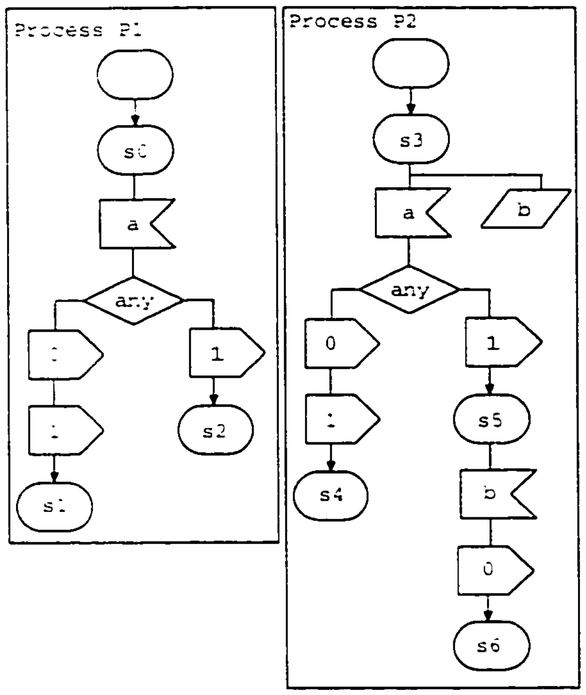


Figure 5.4: P2 is an extension of P1

We are interested in extending the SDL system specifications. Let us consider a SDL specification S and denote by $Input(S)$ the set of signals that can be sent from the environment to S and by $Output(S)$ the set of signals that S can send to the environment. From the environment point of view, i.e., hiding all the internal interactions between the processes in the system, the behavior of the SDL system specification S can be seen, similarly to the processes, as an input/output finite state machine. A state of the system is defined by the state of the processes in the system. The system interacts with the environment through signals from $Input(S)$ and $Output(S)$. The transactions are triggered by signals from $Input(S)$ or by *none*, during a transaction a sequence of output signals from $Output(S)$ is sent to the environment and the system S moves from one state to the next state in a very similar manner to a process. The set of signals to be saved in a global state

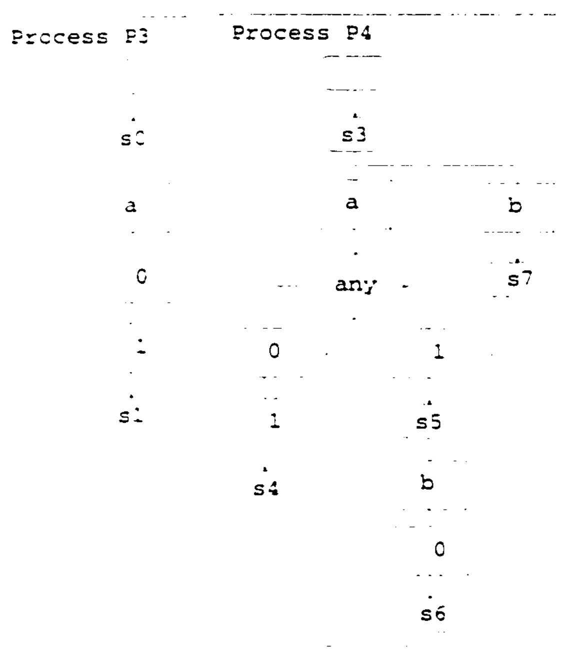


Figure 5.5: P4 is not an extension of P3

s. *Save(s)*, is defined as the union of the saves of the states of the processes that define this global state. With this abstract view of a system in mind, we define the extension relation between global states in a very similar way to Definition 1 and the extension relation between SDL systems in a very similar manner to Definition 2.

In order to extend a SDL system specification, our approach consists of extending the processes in the system. We claim that extending the processes in the system is a sufficient condition to extend the whole system, provided that the set of input signals for the processes in the system are disjoint.

The methodology discussed in Chapter 3 follows the extension relations discussed here. The *post-phase* of the enrichment process consists of merging two SDL behavioral specifications. In the next section, we discuss the heuristics involved in the behavioral merger. which ensure the preservation of old behavior and prevent the introduction of new non-determinism in the resulting specifications.

5.3 Rules for merging SDL Specifications

The merger algorithm proceeds *transition by transition.* A transition in SDL is between one state to another upon reception of an input (trigger) and and a sequence of output signals. In SDL. a transition may contain other constructs like *tasks. decisions* and *procedure calls.* but for the sake of establishment of rules. we will only consider *input* and *sequence* of output signals.

5.3.1 Assumptions and Notations

- Consider processes with four states i.e. $\mathbf{s_1}$. $\mathbf{s_2}$. $\mathbf{s_3}$ and $\mathbf{s_4}$. If $\mathbf{S_{old}}$ represents the set of states in the *old* specification then $S_{old} = \{s_1. s_2. s_3. s_4\}$.
- A transition in the *new* specification is shown in **bold** and called $\mathbf{T_{new}}$.
- A transition in the *old* specification is called $\mathbf{T_{old}}$.
- A transition in the resulting specification is called $\mathbf{T_{res}}$.
- $\mathbf{s_1}$ is the *initial state* for all the transitions.
- $\mathbf{T_{xxx}Input}$ represents the *input* signal in the *initial state*. where as *xxx* is *old. new* or *res*.
- $\mathbf{T_{xxx}Output}$ represents the sequence of *output* signals from *initial state* to *next state*. where as *xxx* is *old. new* or *res*.
- $\mathbf{T_{xxx}NextState(T_{xxx}Input. T_{xxx}Output)}$ represents the *next state* from the *initial state* with input $T_{xxx}Input$ and output $T_{xxx}Output$. where as *xxx* is *old. new* or *res*.

- $\mathbf{T_{res}} = \mathbf{T_{old}} \doteqdot \mathbf{T_{new}}$. where as $\doteqdot$ represents a behavior merger symbol.
- $\mathbf{T_{xxx}}$**Save** is a set of *save* signals in the *initial state* of a transition. where as *xxx* is *old*. *new* or *res*.

5.3.2 Rule 1 (similar transitions)

Two *similar* transitions. when merged. result in a *similar* transition. without any new behavior. Considering figure 5.6. $T_{new}Input = T_{old}Input = \{a\}$. $T_{new}Output = T_{old}Output = \{0\}$. $T_{new}.NextState(T_{new}Input. T_{new}Output) = T_{old}.NextState(T_{old}Input. T_{old}Output) = \{s_2\}$ and $s_2 \in S_{old}$ so $T_{res} = T_{old} \doteqdot T_{new}$ is possible and hence allowed.

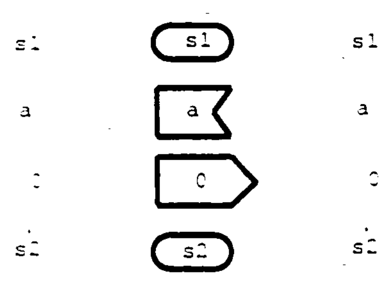
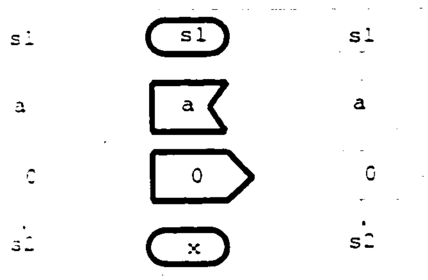


Figure 5.6: Rule 1: merging similar transitions

5.3.3 Rule 2 (same output signal sequence, different next state)

This is perhaps the most important case. as shown in figure 5.7. The new transition differs from the old transition only by the *next state*

$T_{new}.NextState(T_{new}Input.T_{new}Output) = x$. x might be the same state as that in the *old transition*. in which case it becomes *Rule 1*. If x is any of the other states in the S_{old}. then it is not allowed as it produces new *non-determinism*. In all the other cases **merge** x and $T_{old}.NextState(T_{old}Input.T_{old}Output)$ and name it similar to $T_{old}.NextState(T_{old}Input.T_{old}Output)$. In the figure 5.7 x is *merged* with s_2 and named s_2.



```
Assuming x '= s1, s2, s3 or s4

NOTE: if x is s2, this is becomes Rule1
      if x is s1, s3 or s4 it produces new non-determinism
          hence not allowed
```

Figure 5.7: Rule 2: same output signal sequence, different next state

5.3.4 Rule 3 (different output signal sequence, same next state)

This is the case in which the new transition extends the *output sequence* of the corresponding old transition. as shown in figure 5.8. $T_{new}Input = T_{old}Input = \{a\}$ and $T_{new}.NextState(T_{new}Input.T_{new}Output) = T_{old}.NextState(T_{old}Input.T_{old}Output)$ but $T_{new}Output \neq T_{old}Output$ hence $T_{res}Output$ is an *extension* of $T_{old}Output$. as shown in figure 5.8.

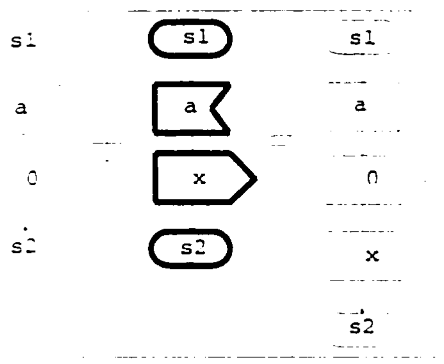


Figure 5.8: Rule 3: different output signal sequence, same next state

5.3.5 Rule 4 (different output signal sequence, different next state)

This is a scenario in which not only the $T_{new}Output$ is different from $T_{old}Output$, but also $T_{new}.NextState(T_{new}Input, T_{new}Output)$ is not the same as $T_{old}.NextState(T_{old}Input, T_{old}Output)$. Considering figure 5.9 if $y = s_2$, this becomes *Rule 3*. For all the other scenarios including $T_{new}.NextState(T_{new}Input, T_{new}Output) \in \{S_{old} - T_{old}NextState\}$ it is **not** allowed because it produces new *non-determinism*. So, considering figure 5.9, if y is s_2, it becomes *Rule 3* and for all other scenarios including y equals s_1, s_3 and s_4 it is not allowed to merge because it produces new non-determinism.

5.3.6 Rule 5 (different input signal)

This is a general rule for all the cases in which the T_{new} has a different $T_{new}Input$ as compared to $T_{old}Input$. So $T_{new}Input \neq T_{old}Input$. For such a situation the T_{old} is appended with a new transition triggered by $T_{new}Input$, followed by $T_{new}Output$ and

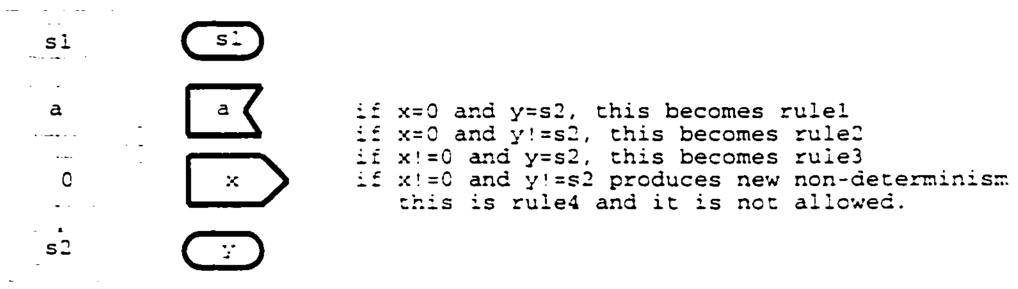

s1
a
0
s2
s1
a
x
y
if x=0 and y=s2, this becomes rule1
if x=0 and y!=s2, this becomes rule2
if x!=0 and y=s2, this becomes rule3
if x!=0 and y!=s2 produces new non-determinism
this is rule4 and it is not allowed.

Figure 5.9: Rule 4: different output signal sequence, different next state

going to $T_{new}.NextState(T_{new}.Input, T_{new}.Output)$. Figure 5.10 shows a new transition appended to the resulting specification, triggered by input signal x. As far as the algorithms are concerned, $T_{new}.Output$ and $T_{new}.NextState(T_{new}.Input, T_{new}.Output)$ become non care conditions.

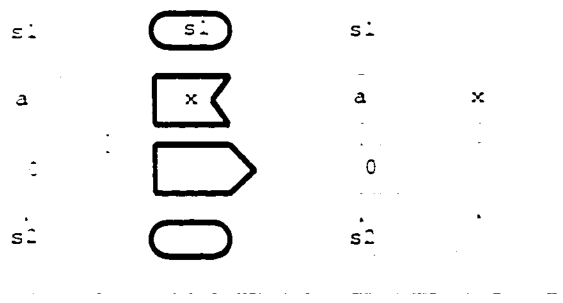


Figure 5.10: Rule 5: different input signal

5.3.7 Rule 6 (merger involving NONE)

When a new transition containing a **NONE** input signal is merged with an existing SDL transition, it produces new non-determinism. This is shown in figure 5.11 Since our intention is to prevent any new non-determinism in an enriched specification, our algorithm detects and prohibits such a case.

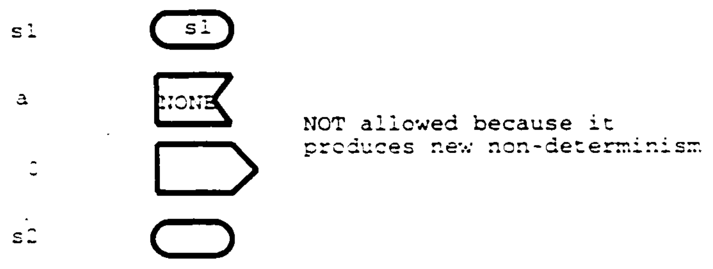


Figure 5.11: Rule 6: merger involving NONE

5.3.8 Rule 7 (merger involving SAVE)

Merger of two transitions with at least one of them having a **SAVE** signal results in a transition containing *all* the saved signals contained in the merging transitions. For all the scenarios discussed so far $T_{old}Save = T_{new}Save = \Phi$. But if $T_{old}Save \neq \Phi$ or $T_{new}Save \neq \Phi$ then $T_{res}Save = T_{old}Save \bigcup T_{new}Save$. This rule may also be combined with any other rules. For example figure 5.12 shows a scenario where *Rule 7* is combined with *Rule 5*.

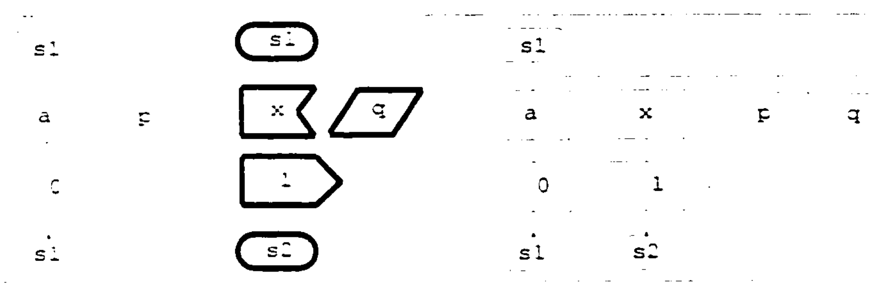


Figure 5.12: Rule 7: merger involving SAVE

5.3.9 Rule 8 (merger resulting in unreachable code)

If the starting state in the new transition is not present in the old specification. there is no way to know where to plug in this new behavior. The new behavior is thus *unreachable code* and is not allowed in our algorithm. This rule shows that the state names (equivalent to conditions in MSCs) are very significant in the automated extension and enrichment of SDL specification with MSCs. This is rather a limitation of the of the algorithms. hence implying that the user should be rather careful with the nomenclature of *conditions* in MSCs.

5.3.10 Rule 9 (involving ANY + output signal sequence exists + old transitions go to different states)

SDL keyword **ANY** is very useful in modeling non-determinism. An *old* SDL specification may have non-determinism in it modeled using **ANY** keyword. As we have already mentioned. the enrichment or extension of the SDL specification should preserve the *old* behavior. This include preserving the existing *non-determinism* in the *old* SDL specification. But still *new* transitions resulting due to the *new* MSC may be such that some or all the non-deterministic transitions in the *old* SDL specification need to be extended/enriched. We have designed six rules concerning transitions involving ANY. Rule 9 to 14, below, deal with transitions involving NONE.

Say the *new* SDL transition has the same input signal as the old non-deterministic transition. Output signal sequence *exists* in the old transition but not going to the same state. This will enrich the *output sequence*, using *Rule 3* of the old transition, which goes to the same state. This is shown in figure 5.13. Note that the new transition is merged with the *left* branch using rule 3. No merger is done with the *right* branch. Had it been done, it would use rule 2 and disallow the merger.

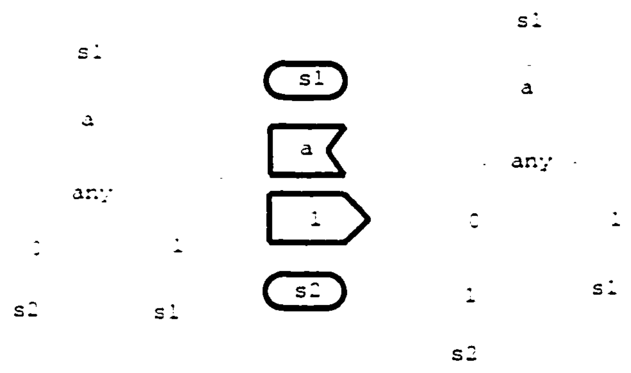


Figure 5.13: Rule 9

5.3.11 Rule 10 (involving ANY + output signal sequence does not exist + old transition go to different states)

This is the case when the output signal sequence of the new transition is *not* present in the output signal sequence of the old non-deterministic transitions. Again, the next state is the deciding factor for enriching/extending old non-deterministic transitions. This rule like *Rule 9*, uses *Rule 3* for enriching/extending the output signal sequence, as shown in figure 5.14.(b).

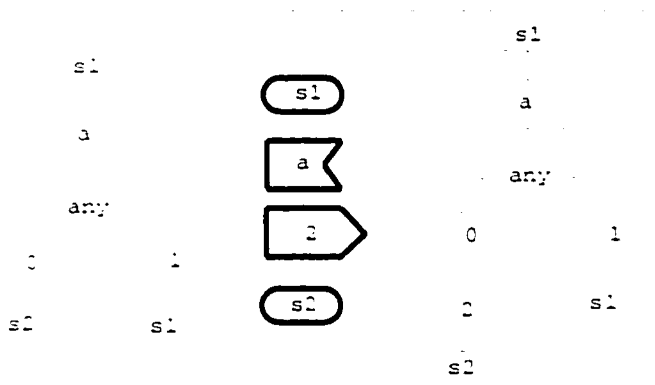


Figure 5.14: Rule 10

5.3.12 Rule 11 (involving ANY + output signal sequence exists + old transitions go to same states)

This is the case when the old non-deterministic transitions go to the same state. The output signal sequence of the new transition is present in the old non-deterministic transitions. So care must be taken for merger. As shown in the figure 5.15, the new transition merges with the old *left* non-deterministic branch according to *Rule 3*.

but it merges with the old *right* non-deterministic branch according to *Rule 1.*

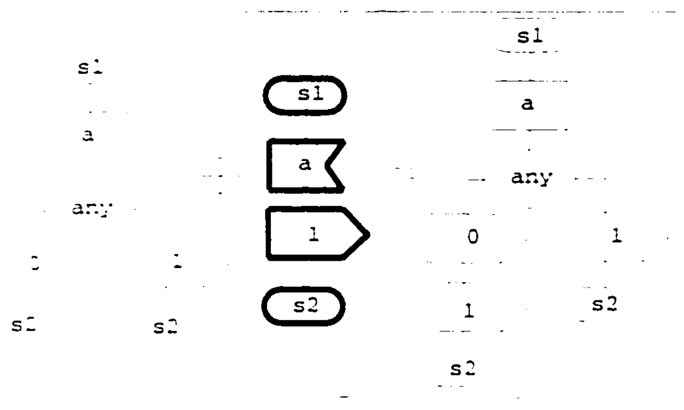


Figure 5.15: Rule 11

5.3.13 Rule 12 (involving ANY + output signal sequence does not exist + old transition go to same states)

In this case, the output signal sequence does not exist in the old SDL non-deterministic transitions. Hence it should enrich/extend both *left* and *right* non-deterministic transitions in the *old* SDL. This is shown in figure 5.16.

5.3.14 Rule 13 (involving ANY + different input signal)

This rule is in fact an extension to *Rule 5.* It is the case when the new transition has a new input signal but starting from the same state as the old non-deterministic transitions. This, in accordance to *Rule 5.* will add a new branch for the same state. See figure 5.17 for an elaboration.

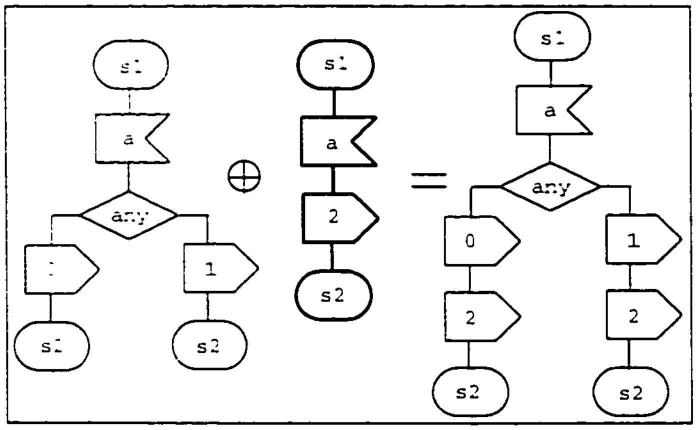


Figure 5.16: Rule 12

5.3.15 Rule 14 (involving ANY + same input signal + different output signal sequence + different next state)

This is the case when the new transition produces a new non-determinism in the enriched specification. This happens when the input signal of the new transition is the same as that of the old non-deterministic transition but the output signal sequence and the next state of the new transition is different. This is an extension to *Rule 4*. See figure 5.18 for an elaboration.

All of the 14 rules of merger discussed above have been implemented and tested in our *post phase* merger tool. We will some examples in Chapter 6.

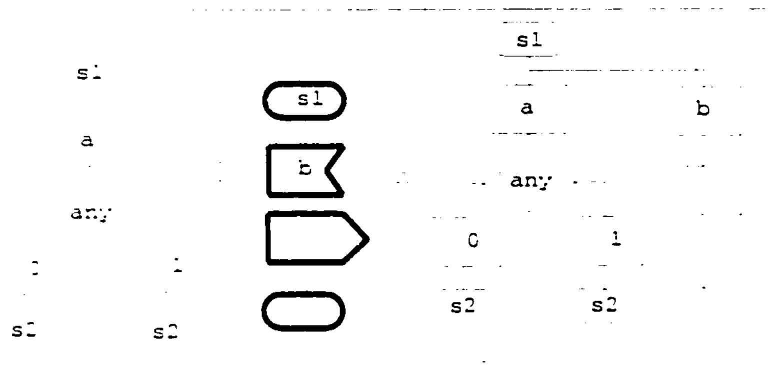


Figure 5.17: Rule 13

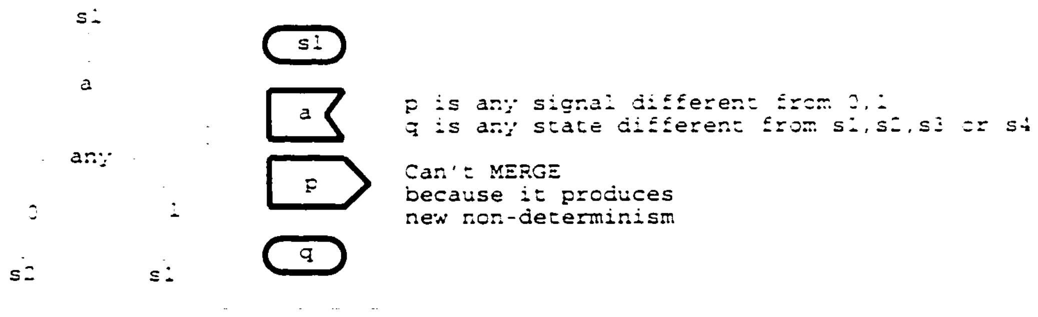

s1
a
any
0
1
s2
s1
s1
a
p
q
p is any signal different from 0,1
q is any state different from s1,s2,s3 or s4
Can't MERGE
because it produces
new non-determinism

Figure 5.18: Rule 14

Chapter 6

Tools and Examples

6.1 Introduction to Tools

SDL Enrichment Tool is part of a tool suite consisting of UML−MSC−SDL tools. The tool suite contains tools for automated translation from UML to SDL and MSC to SDL. Further, it contains a tool set for MSC Refinement. All of these tools are accessible through a simple interface, as shown in figure 6.1.

6.1.1 View menu

Related sub-tools are packaged together in a single view. The view can be changed through the **View** menu bar. As shown in figure 6.2, there are three views available i.e. **UML to SDL**, **MSC Refinement** and **SDL Enrichment**.

6.1.2 File menu

File menu, shown in figure 6.3, allows creating, opening, saving and closing projects. A project may include *.xmi files (class diagrams), *.pr files (SDL specifications) and/or *.msc files (MSC specifications). A new project is created by New Project menu of the File menu. It must be saved with a *.tpr extension.

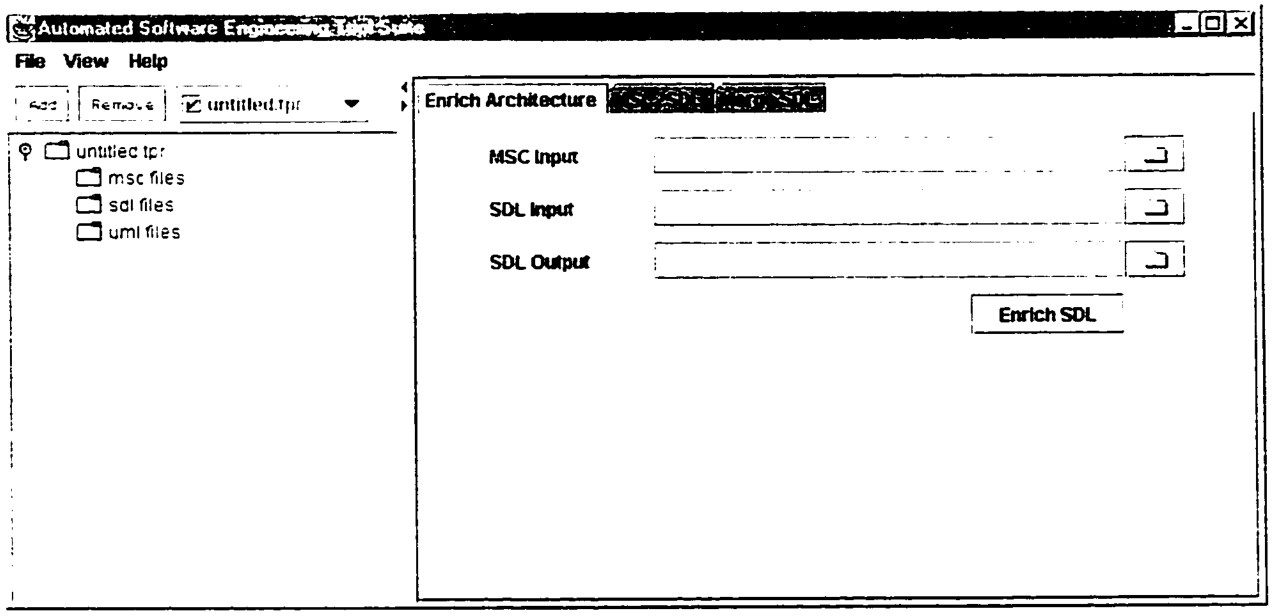


Figure 6.1: Tool Interface

6.1.3 Select active Project

Several projects may be loaded at the same time into the tool, but only one is allowed to be active. The functionality of all the tools is linked to the active project. To change the active project, click on the drop down list as shown in figure 6.1.

6.1.4 Installating and running the tool

The tool is written in Java and packaged into a JAR file. Hence, it requires no installation as long as the platform has the Java Run-time Environment installed on it. To run the tool on UNIX use the following command:

```
java -jar tool.jar
```

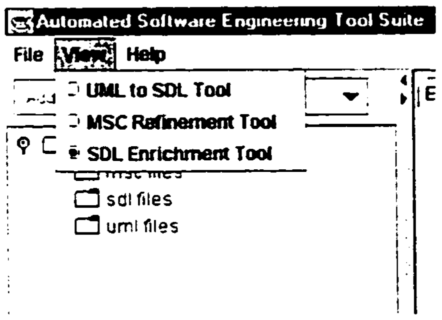


Figure 6.2: View menu of the tool

6.2 Architecture of the SDL Enrichment tool

Figure 6.4 shows high level architecture of the SDL Architectural Enrichment tool. *MSCParser* parses the new MSC file and builds the *MSCStructureTable*. as mentioned in Chapter 4. Similarly *SDLParser* parses the old SDL file and builds the *SDLStructureTable*. The *ArchitectureEnricher* runs the algorithms (discussed in Chapter 4) on *MSCStructureTable* and the *SDLStructreTable* and uses *ProcessEnricher*, *ChannelEnricher* and the *SignalEnricher* to produce an enriched SDL architecture. The *FileWriter* writes this to a **.pr* file.

For the SDL Merger Tool. as shown in figure 6.5. A single *SDLParser* is used to build *OldSDLDataStructure* and *NewSDLDataStructure* from old and new SDL specifications respectively. The *Merger* merges old and new SDL. using their respective data structures and the *RuleImplementer*. The *FileWriter* writes the resulting. enriched specification to an explicitly specified **.pr* file.

6.3 Automated Teller Machine (ATM)

We illustrate our approach by enriching an ATM system (in SDL) with a new function (in MSC). An initial ATM *system* specification is shown in figure 6.6. It consists

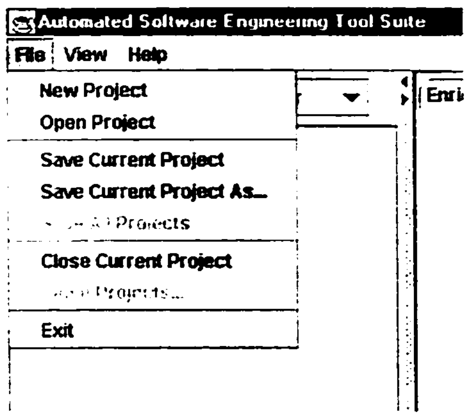


Figure 6.3: File menu

of a *block* called *ATM_BLOCK*. The *ATM_BLOCK* exchanges *signals* with the environment through a channel *to_env*.

The *ATM_BLOCK* is shown in figure 6.7. It has two processes, ATM and *Bank*. Their behaviors are shown in figures 6.8 and 6.9 respectively. The user starts the transaction by entering a *card* into the machine. The *ATM* then waits for 10 units of time for the user to enter a valid pin number. If he does not do so, the *ATM* returns the card to the user and goes back to the *idle* state. If the user is able to enter a valid pin number within 10 units of time, the *ATM* goes to *options* state. The user has two *options* in this specification. He can either *withdraw* money or *cancel* a transaction. However, the user's card is rejected if he supplies an invalid pin number. The process *Bank* is internal to the system. It does not interact with the environment. The process *ATM* interacts with the users in the environment. Our specification assumes one user at a time since we only have one *ATM* process.

Now, assume that we want to *enrich* the existing function with a new functionality, i.e., when the *card* has been accepted, the user should also be able to

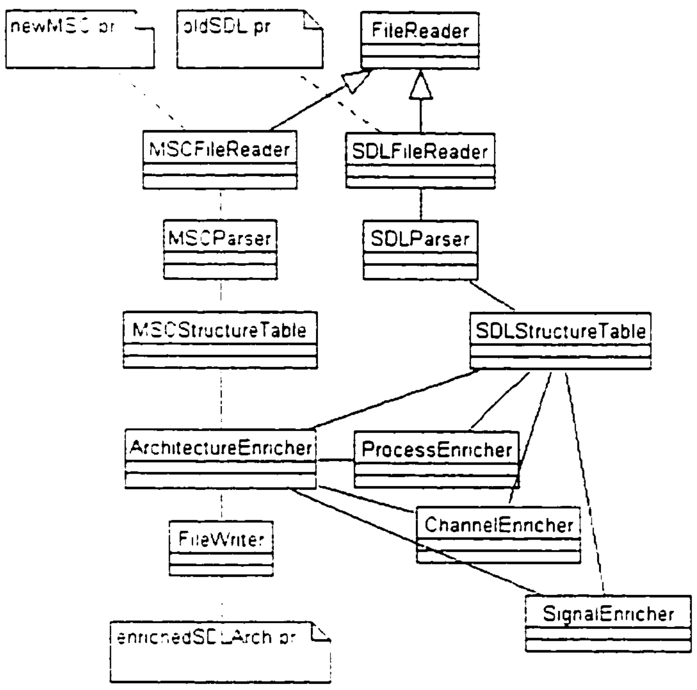


Figure 6.4: Architecture of the SDL Architecture Enrichment tool

getBalance in addition to *withdraw* and *cancel*. This new function (behavior). is described in the MSC given in figure 6.10. In MSC *getBalance*. we have used a global condition. *options*. to indicate where the new behavior is to be added in the existing SDL processes. Notice that the style of the given MSC is very important. The *conditions* are important for merging the new behaviors with the old ones. The user should know where the new behaviors have to be added. However. the conditions do not have to be global. they can even be local. While adding this new functionality into the system. we want to preserve the old behavior of the system (*getBalance* and *cancel*). Our automated approach guarantees this.

As discussed in Chapter 3. the first step in our approach is *enrichment of existing architecture*. SDL Architectural enrichment tool does this while getting

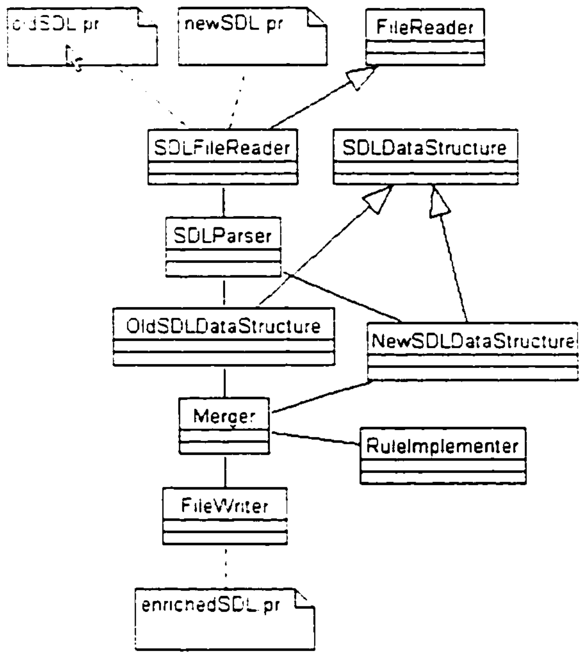


Figure 6.5: Architecture of the SDL Merger tool

feedback from the user, whenever required. For this new MSC, no new processes are to be added into the existing architecture. As shown in the figure 6.10, *get_balance*, *req_balance*, *balance* and *print_balance* are new signals to be added into the architecture. The tool allows them to be added either through existing *links* or through new *links* between *ATM* and *Bank* processes. The enriched ATM system and block are shown in figures 6.11 and 6.12, respectively.

The next step in our approach is to generate the new behavior for the *ATM* and *Bank* processes. This is done in the MSC2SDL phase of our approach (refer to Chapter 3). Newly generated behaviors for *ATM* and *Bank* are shown in the figures 6.13 and 6.14 respectively.

The final step in our approach is the merger of old and new behaviors to get the enriched system (refer to Chapter 3). This is done by the SDL merger

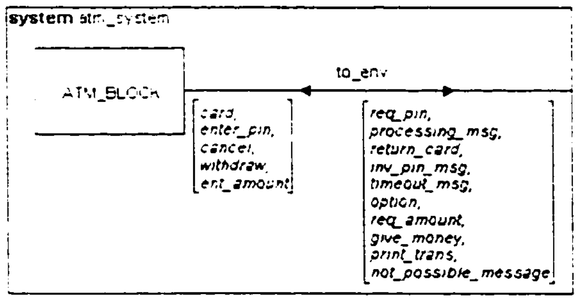


Figure 6.6: The ATM System

tool. It merges the behaviors of old and new *ATM* processes of figure 6.8 and 6.13. respectively. using the rules of merger discussed in Chapter 5. Similarly. it merges the behaviors of old and new *Bank* processes of figures 6.9 and 6.14. respectively. Finally. the enriched ATM system and block *ATM_BLOCK* are shown in figures 6.15 and 6.16. respectively. The processes *ATM* and *Bank* with enriched behavior are shown in figures 6.17 and 6.18. respectively. Notice that in the behaviors of the enriched processes. all the old behavior is preserved and and no new non-determinism is added.

Further Enrichment (second iteration)

Now. we illustrate further enrichment of the ATM specification. Currently. we have an ATM specification having *options* to *withdraw* money. *cancel* a transaction and *get_balance* of a user. as shown in figures 6.15. 6.16. 6.17 and 6.18. Say we want the ATM system to have the functionality for *depositing* money. This new function is specified in an MSC *deposit.* shown in figure 6.19. The SDL architecture enrichment tool is used to enrich the existing architecture (figures 6.15 and 6.16) in the *pre-phase.* Resulting. enriched architecture is shown in figures 6.20 and 6.21. Note that new signals *deposit. option_deposit. ok. amount_msg. amount* and *deposit_amount*

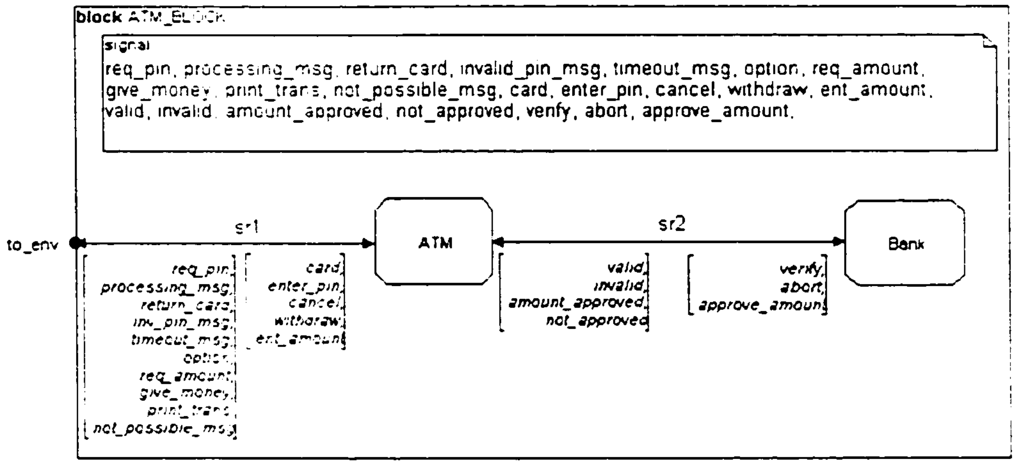

block ATM_BLOCK
signal
req_pin, processing_msg, return_card, invalid_pin_msg, timeout_msg, option, req_amount,
give_money, print_trans, not_possible_msg, card, enter_pin, cancel, withdraw, ent_amount,
valid, invalid, amount_approved, not_approved, venfy, abort, approve_amount.
to_env
sr1
ATM
sr2
Bank
req_pin, processing_msg, return_card, inv_pin_msg, timeout_msg, option, req_amount, give_money, print_trans, not_possible_msg
card, enter_pin, cancel, withdraw, ent_amount
valid, invalid, amount_approved, not_approved
verify, abort, approve_amount

Figure 6.7: The ATM Block

are added in the enriched architecture. MSC2SDL tool, in the *MSC2SDL-phase*, generates new behaviors for *ATM* and *Bank* processes using new MSC (figure 6.19) and the enriched architecture. The new behaviors of *ATM* and *Bank* processes is shown in figures 6.22 and 6.23 respectively. Finally, the SDL merger tool, in *post-phase*, merges both the old and new specifications to produce enriched ATM specifications. The enriched behaviors of *ATM* and *Bank* processes is shown in figures 6.24 and 6.25 respectively. Thus, in this second iteration of incremental development of the ATM specification, we have added another functionality into ATM system. This iterative process can be continued until all the requirements are met. Intermediate or final prototypes can be simulated, verified and validated.

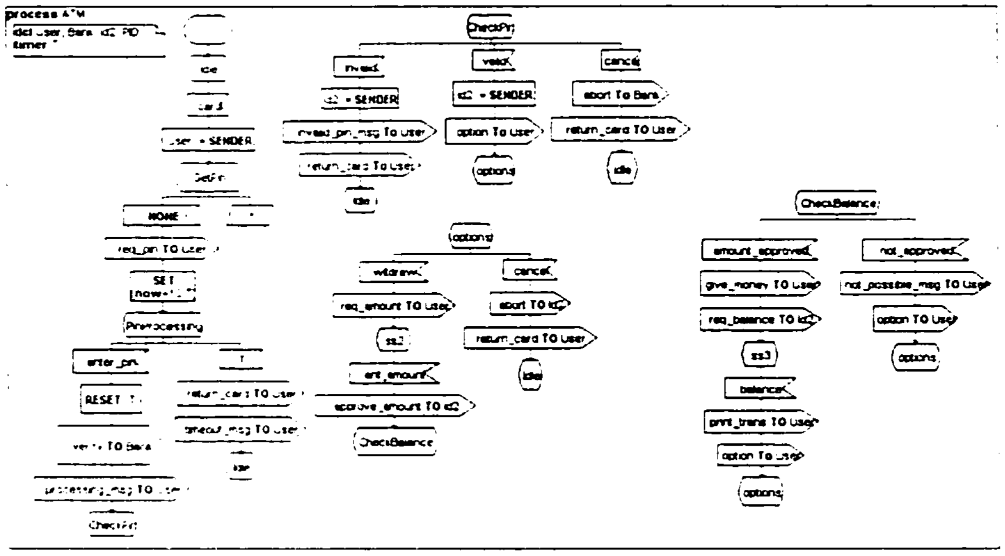


Figure 6.8: Behavior of the ATM process

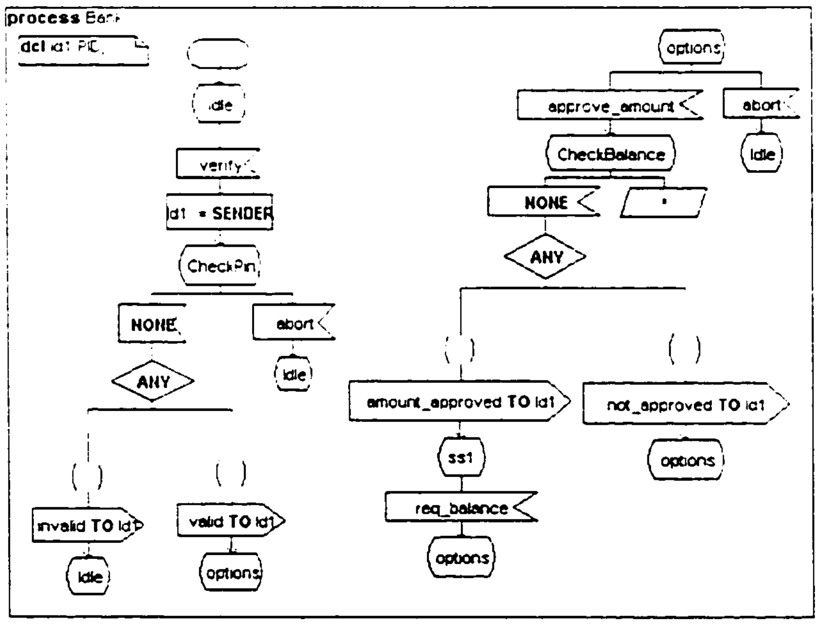


Figure 6.9: Behavior of the Bank process

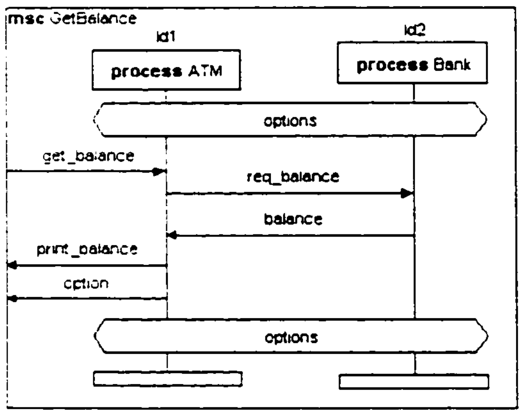


Figure 6.10: An MSC describing the new function to be added in old ATM

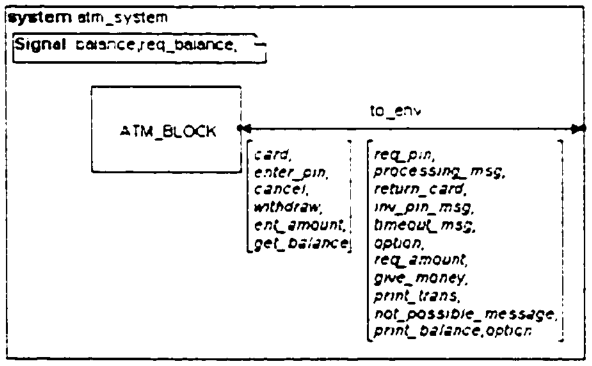


Figure 6.11: Signals get_balance and print_balance added to the ATM System

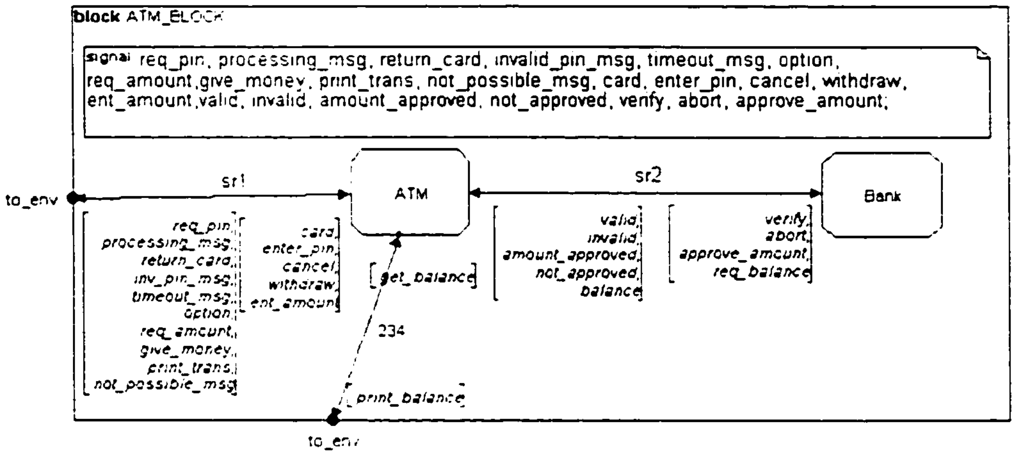

block ATM_BLOCK
signal req_pin, processing_msg, return_card, invalid_pin_msg, timeout_msg, option, req_amount,give_money, print_trans, not_possible_msg, card, enter_pin, cancel, withdraw, ent_amount,valid, invalid, amount_approved, not_approved, verify, abort, approve_amount;
to_env
sr1
ATM
sr2
Bank
req_pin, processing_msg, return_card, inv_pin_msg, timeout_msg, option, req_amount, give_money, print_trans, not_possible_msg
card, enter_pin, cancel, withdraw, ent_amount
get_balance
valid, invalid, amount_approved, not_approved, balance
verify, abort, approve_amount, req_balance
234
print_balance
to_env

Figure 6.12: Signals get_balance, print_balance, req_balance and balance added to the ATM_BLOCK block

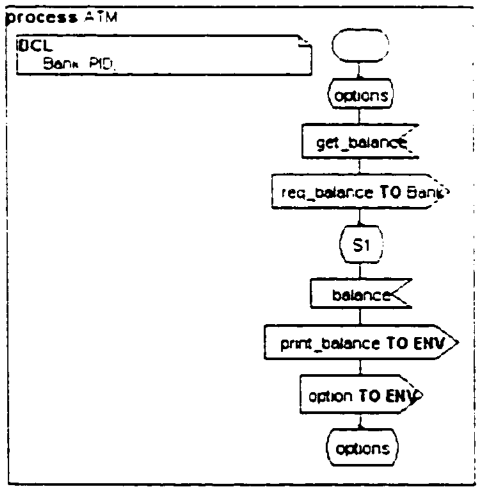


Figure 6.13: New ATM behavior generated by the MSC2SDL tool

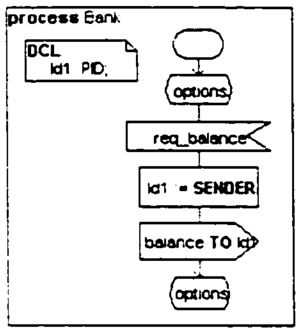


Figure 6.14: New bank behavior generated by the MSC2SDL tool

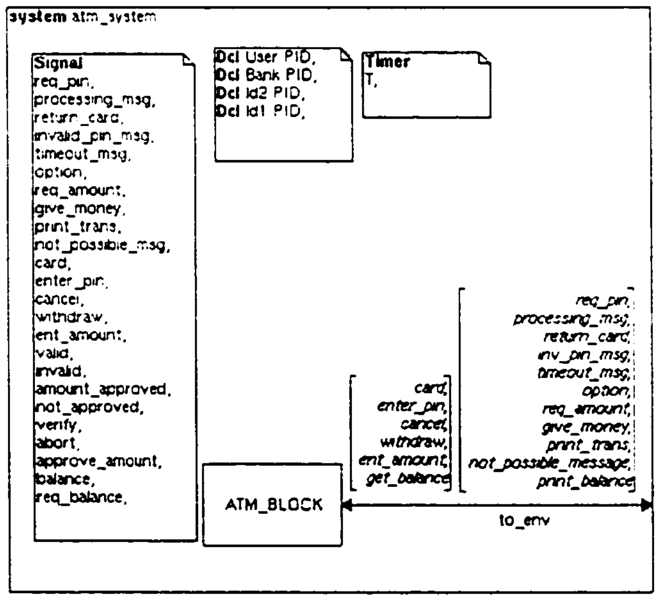


Figure 6.15: Final Architecture of the Enriched ATM System

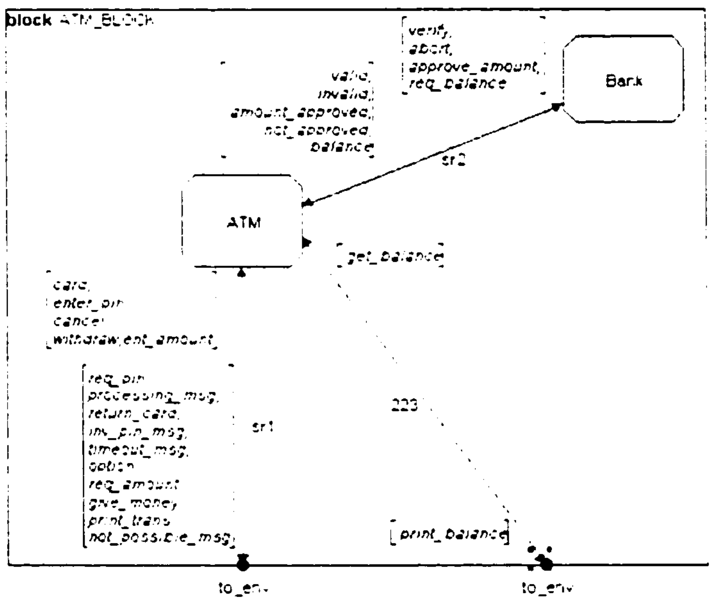


Figure 6.16: Final Architecture of the Enriched ATM_BLOCK Block

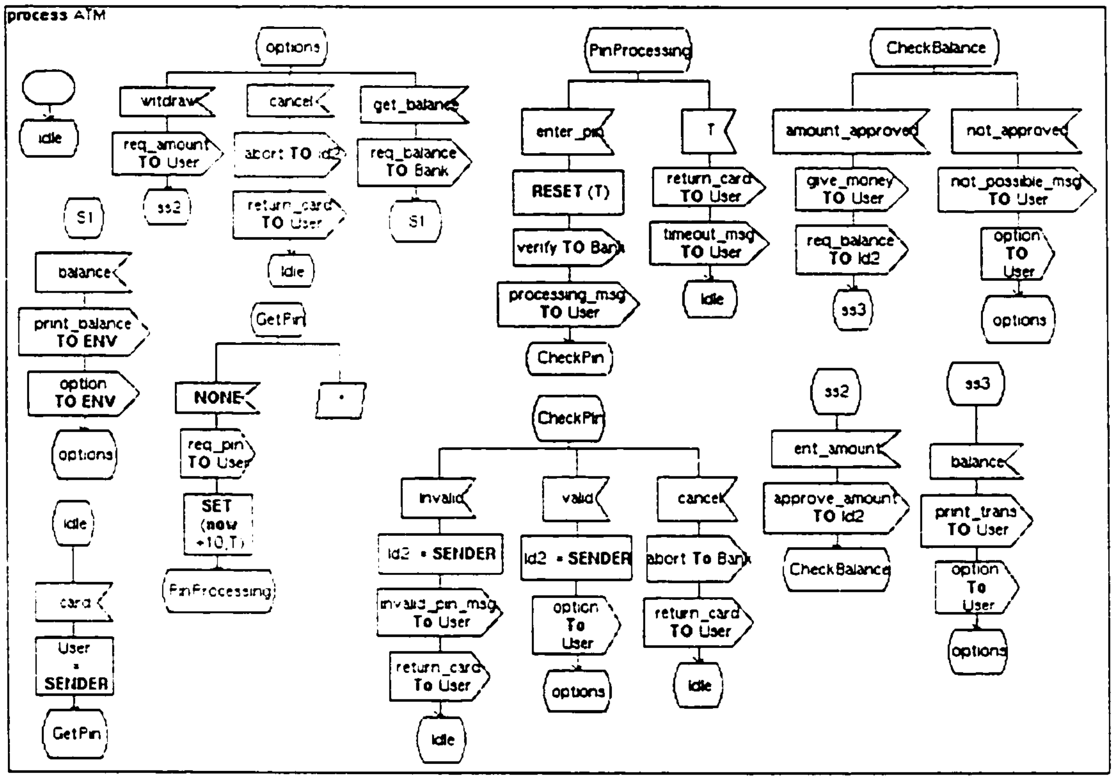


Figure 6.17: Enriched ATM process

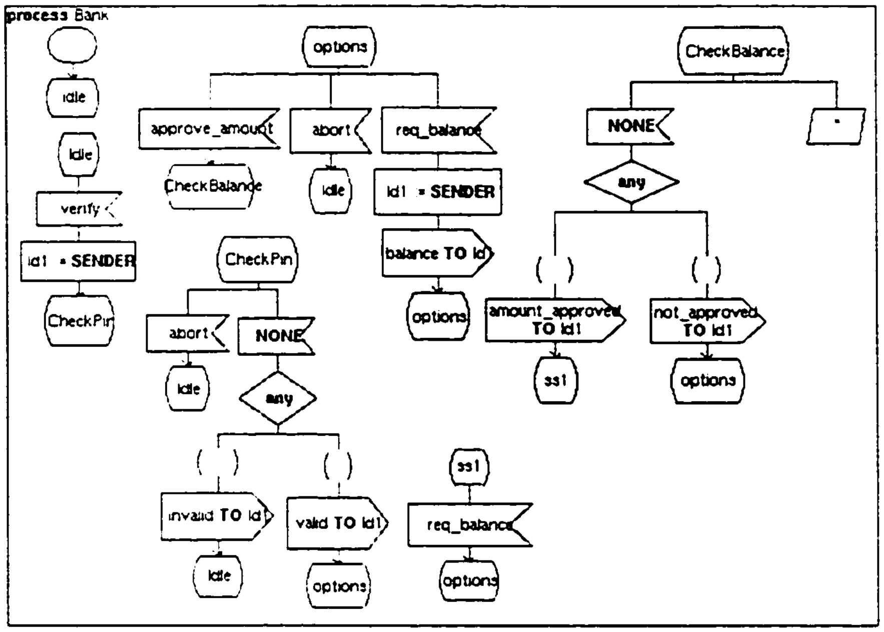


Figure 6.18: Enriched Bank Process

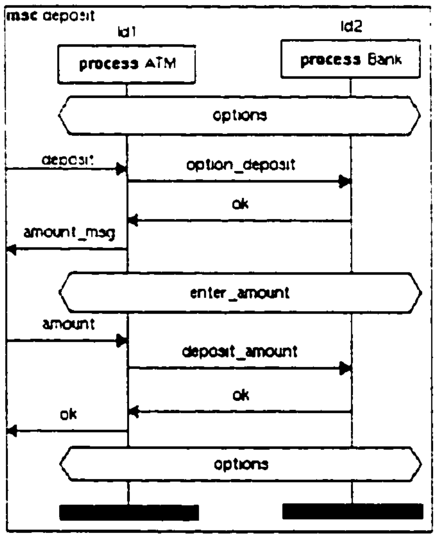


Figure 6.19: A new MSC specifying deposit functionality

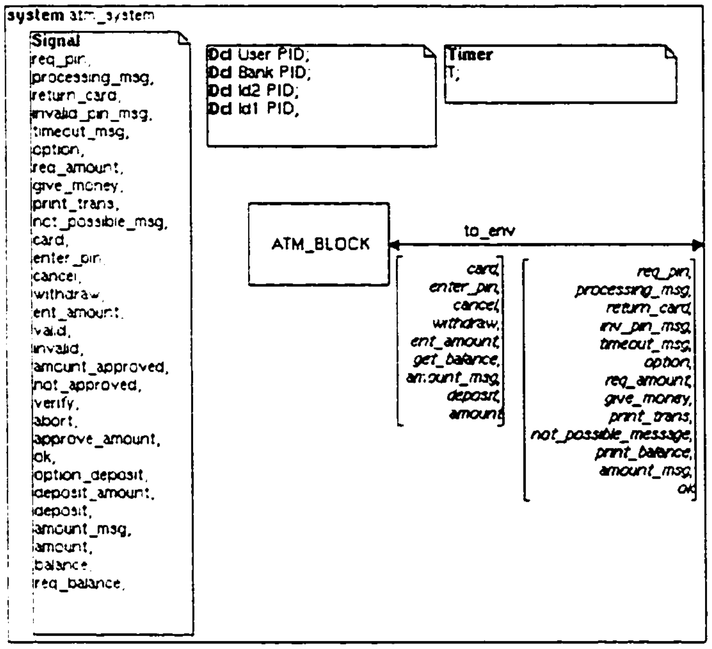

system atm_system
Signal
req_pin, processing_msg, return_card, invalid_pin_msg, timeout_msg, option, req_amount, give_money, print_trans, not_possible_msg, card, enter_pin, cancel, withdraw, ent_amount, valid, invalid, amount_approved, not_approved, verify, abort, approve_amount, ok, option_deposit, deposit_amount, deposit, amount_msg, amount, balance, req_balance,
Dcl User PID;
Dcl Bank PID;
Dcl Id2 PID;
Dcl Id1 PID,
Timer
T;
ATM_BLOCK
to_env
card, enter_pin, cancel, withdraw, ent_amount, get_balance, amount_msg, deposit, amount
req_pin, processing_msg, return_card, inv_pin_msg, timeout_msg, option, req_amount, give_money, print_trans, not_possible_message, print_balance, amount_msg, ok

Figure 6.20: Enriched ATM system architecture

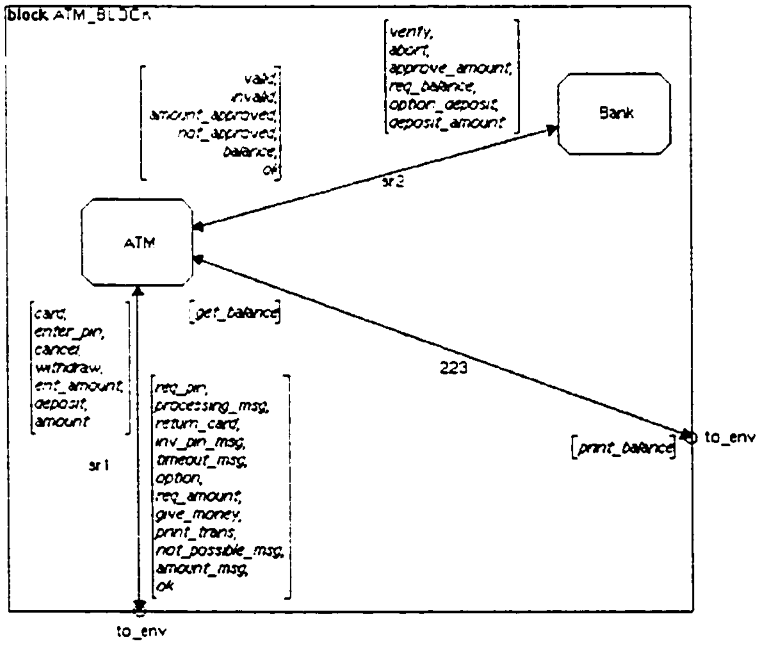

block ATM_BLOCK
verify, abort, approve_amount, req_balance, option_deposit, deposit_amount
Bank
valid, invalid, amount_approved, not_approved, balance, ok
sr2
ATM
card, enter_pin, cancel, withdraw, ent_amount, deposit, amount
get_balance
223
req_pin, processing_msg, return_card, inv_pin_msg, timeout_msg, option, req_amount, give_money, print_trans, not_possible_msg, amount_msg, ok
sr1
print_balance
to_env
to_env

Figure 6.21: Enriched ATM_BLOCK block architecture

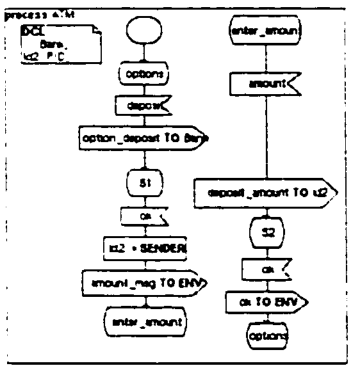


Figure 6.22: New ATM behavior generated by MSC2SDL tool

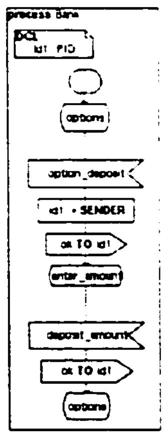


Figure 6.23: New Bank behavior generated by MSC2SDL tool

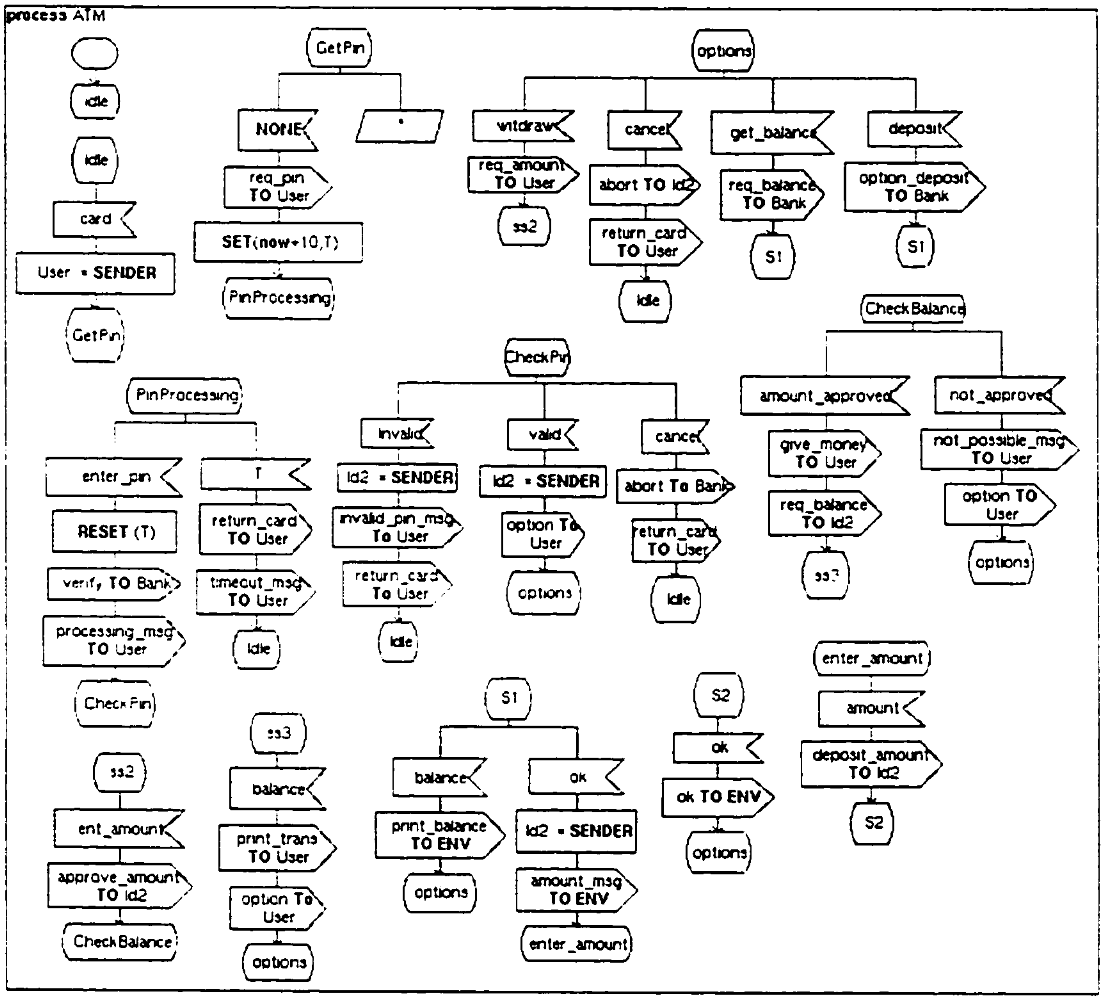


Figure 6.24: Enriched process ATM

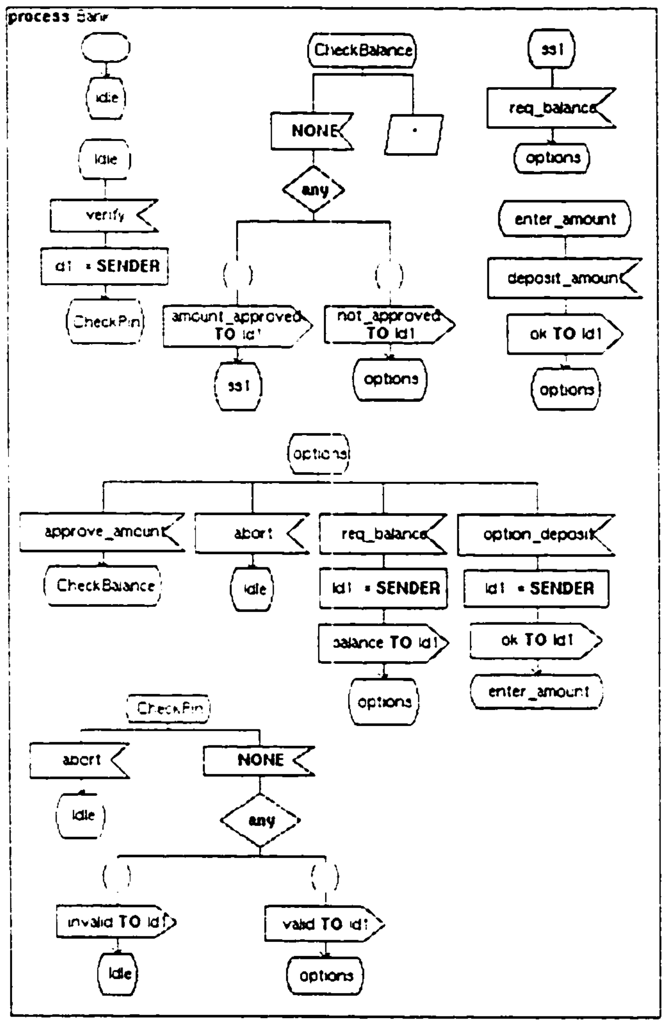


Figure 6.25: Enriched process Bank

Chapter 7

Conclusions

7.1 Contributions of this thesis

Our approach can be used for *enrichment* of SDL specifications with MSC during the maintenance phase or it can be used for incremental design of SDL specification using MSC. The approach consists of three steps, named as *pre phase*, *MSC2SDL phase* and *post phase*. The *pre phase* deals with enrichment of SDL architecture (only) using MSC. This includes addition of new processes into existing or new blocks, and hence addition of new *links* followed by addition of new *signals*. Or, it includes *enrichment* of existing *links* and *signal specifications*. We have presented algorithms for *pre phase* in Chapter 4. *MSC2SDL* phase relies on existing *MSC2SDL* tool[21, 22, 23, 24]. *Post phase* merges the *old* and *new* SDL specifications to get enriched SDL. We have given 14 rules for merging SDL specifications (Chapter 5), based upon *extension* relations defined in [17]. These extension relations, and hence the formed rules, ensure that the old behavior will be preserved in the enriched system and there will be no new non-determinism in the enriched system. In our approach the *style* of the new MSC is very important. By *style* we mean that the user should be aware of the old SDL specifications while writing a new MSC. If he wants to specify new message exchanges between existing process(es) and new process(es) or new messages between existing processes, then he must be very careful to name the MSC processes instances *exactly* as in the old SDL for *existing processes*. Further, message exchanges in MSC should be specified bounded by local or global *conditions*. The name of *conditions* on the MSC instance of an existing process is very significant. It is important because the SDL behavior generated by MSC2SDL tool transforms *conditions* into *syntactical states* in SDL. The merger algorithms proceed comparing and processing based upon these state names.

7.2 Future Work

7.2.1 Covering further SDL constructs

Presently, our algorithms and tools are applicable to enriching and working with *flat* SDL architecture. Our work can be extended to cover hierarchical SDL specifications, i.e. blocks within blocks. SDL behavior may also be specified as services within processes. Further work can cover the *enrichment or extension* of services too, based upon MSC. The merger tool can be extended to cover further SDL constructs such as *procedure calls*, *formal context parameters* etc.

7.2.2 More relations for enrichment

We have developed algorithms, rules and tools based upon an enrichment relation defined in [17]. As we have already mentioned, the characteristic of this relationship is that it guarantees to preserve old behavior and prevents inclusion of new non-determinism in the enriched SDL. A user may want another kind of enrichment, which, for example, *allows new non-determinism* to be added in the enriched specifications. This requires more formal relations of extension depending upon what type of enrichment is required. As far as the enrichment of SDL architecture is concerned, its algorithms will remain the same as discussed in Chapter 4. But for each new enrichment relation, a new set of rules of SDL merger have to be developed. Say we have a set of formal relations for enrichment $Rel_1, Rel_2, Rel_3, \ldots, Rel_n$. For each relation, we will have a set of rules to guide the algorithms i.e $RuleSet_1, RuleSet_2, RuleSet_3, \ldots, RuleSet_n$ respectively. Further work can include development of such rules and relations.

7.2.3 Extended Specialization

SDL2000[2] is a rich specification language supporting object oriented concepts. It allows enrichment of existing types using *specialization*, i.e., a super-type can be specialized into a sub-type using the specialization constructs of SDL. Specialization supports addition of new structure and behavior. This specialization is different from our approach in two ways. Firstly, it does not consider specialization based upon MSC. Secondly, it does not guarantee that there will not be any new non-determinism in the specialized specification, in addition to the already existing one in the old specification. To consider this, the specialization construct can be defined into an *extended specialization* which is based upon formally defined relations. Tools may allow specialization depending upon formally defined enrichment relations. The user may select the type of relation and use that specialization to enrich existing types. Further work may include development of such an *extended specialization.*

References

[1] Rumbaugh J., Jackobson I., and Booch G., *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[2] ITU-T. *Specification and Description Language-SDL'2000.* ITU-T, 1999.

[3] ITU-T. *Messaga Sequence Charts-MSC'2000.* ITU-T, 1999.

[4] Meyers S., *Effective C++, 2nd Ed.* Addison Wesley, 1997.

[5] Niemeyer P. and Peck J., *Exploring Java, 2nd Ed.* O'Reilly and Associates, Inc., 1997.

[6] Flanagan D., *Java in a Nutshell: a desktop quick reference.* O'Reilly and Associates, Inc., 1997.
A very good refrence book for java developpers.

[7] Winder R. and Roberts G., *Developing Java Software, 2nd Ed.* John Willey and Sons Ltd., 2000.

[8] Harvey M. Deitel and Paul J. Deitel. *Java How To Program, 2nd Ed.* Prentice Hall, 1997.

[9] Rumbaugh J., Jackobson I., and Booch G., *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[10] Meyer B.. *Object Oriented Software Construction. 2nd Edition.* Prentice Hall. 1997.

[11] Sommerville. *Software Engineering. Fifth Edition.* Addison-Wesley. 2000.

[12] Royce W. W.. "Managing the Development of Large Software Systems: Concepts and Techniques." *Proceedings of WESCON.* 1970.

[13] Gilb T.. *Principals of Software Engineering Management.* Addison-Wesley. 1988.

[14] Boehm B. W.. "The Spiral Model for Software development and Enhancement." *ACM SIGSOFT Software Engineering Notes.* 1986.

[15] Rumbaugh J.. Jackobson I.. and Booch G.. *The Unified Software Development Process.* Addison-Wesley. 1999.

[16] Andreas Mitschele-Thiel. *Systems Engineering with SDL.* John Willey and Sons. Ltd.. 2001.

[17] Khendek F. and Vincent D.. "Extending SDL Specifications with MSCs." *Proceedings of SDL and MSC workshop (SAM'2000). Grenoble. France.* June 2000.

[18] Anders EK. *SDL-Object Oriented Modeling Technique (SOMT).* Telelogic AB. 1996.

[19] Braek R.. Gorman J.. and Haugen O.. *TIMe - The Integerated Method.* SINTEF. Norway. 1999.

[20] Verilog SA. *ObjectGEODE Method Guidelines Version 1.0.* Verilog SA. 1996.

[21] Robert G.. Khendek F.. and Grogono P.. "Deriving an SDL Specification with a given architecture from a set of MSCs." *A. Cavalli and A.Sarma (eds.). SDL'97: Time for Testing-SDL. MSC and Trends. Proceedings of the eight SDL Forum. Evry. France.* Sept. 1997.

[22] Khendek F., Robert G., Butler G., and Grogono P., "Implementability of Message Sequence Charts," *Proceedings of SDL Society, International Workshop on SDL and MSC (SAM'98), Berlin, Germany.*, June 1998.

[23] Abdallah M.M., Khendek F., and Butler G., "New Results on Deriving SDL specifications from MSCs," *Proceedings of SDL Forum'99, R. Dssouli, G.V. Bochmann, Y. Lahav (eds), Montreal, Canada.*, June 1999.

[24] Abdallah M.M., "Automatic Generation of SDL Specifications from MSCs," Master's thesis, Electrical & Computer Engineering, Concordia University, 1999.

[25] Khendek F., Bourduas S., and Vincent D., "Stepwise design with message sequence charts," *FORTE 2001, Korea*, 2001.

[26] Bourduas S., "Generation of SDL Specifications from UML and MSC Use Cases," Master's thesis, Electrical & Computer Engineering, Concordia University, June 2001.

[27] ISO, *(IS9074) Estelle: A formal description technique based on an extended state transition model*, ISO, 1989.

[28] ISO, *(IS8807) LOTOS: A formal description technique based on the temporal ordering of observational behavior*, ISO, 1989.

[29] ITU-T, *ITU-T.Z.100, CCITT Specification and Description Language (SDL)*, ITU-T, 1993.

[30] Ellsberger J., Hogrefe D., and Sarma A., *SDL, Formal Object Oriented Language for Telecommunication Systems*, Prentice Hall Europe, 1997.

[31] Verilog, *ObjectGEODE*, Verilog, Toulouse, France., 1999.

[32] Telelogic, *SDT Tau*, Telelogic, Sweden., 1999.

[33] Object Management Group. "The Website of Object Management Group." http://www.omg.org.

[34] ITU-T. *(X.680)ASN.1: Specification of the Basic Notation.* ITU-T. 1998.

[35] Faergemand O. and Olsen A.. "Introduction to SDL-92." *Computer Networks and ISDN Systems 26.* 1994.

[36] ITU-T. *Recommendation Z.120-Message Sequence Charts (MSC).* ITU-T. 1992.

[37] ITU-T. *SDL methodology guidelines, Recommendation Z.100 Annex I.* ITU-T. 1993.

[38] ITU-T. *Algebraic semantics of Message Sequence Charts. Recommendation Z.120 Annex B.* ITU-T. 1995.

[39] ITU-T. *Recommendation Z.120-Message Sequence Charts (MSC).* ITU-T. 1996.

[40] Mauw S.. Reniers M.A.. and Willemse T.A.C.. "Message Sequence Charts in Software Engineering Process." *Handbook of Software Engineering and Knowledge Engineering.* 2000.

[41] Khendek F. and Bochmann G.V.. "Merging Behavioral Specifications." *Journal of Formal methods in System Design.. Vol.6. No.3.* June 1995.

[42] Faergemand O.. "Stepwise Production of an SDL description." *Proceedings of Formal Description techniques (FORTE)III. Madrid. Spain.* 1990.

[43] Ichikawa H.. Yamanaka K.. and Kato J.. "Incremental Specification in LOTOS." *Proceedings of Protocol Specification Testing and Verification (PSTV). Ottawa. Canada.* 1990.

[44] Nicola R. De. "Extension Equivalence for Transition Systems." *Acta Informatica. 24.* pp. 211–237. 1987.

[45] Brinksma E.. Scollo G.. and Steenbergen S.. "LOTOS Specifications. their implementations and their tests." *Proceedings of Protocol Specification Testing and Verification (PSTV) 86. B.Sarikaya and G.Bochmann (eds). Montreal. Canada.*. 1986.