Helsinki University of Technology Laboratory for Theoretical Computer Science
Technical Reports 21
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tekninen raportti 21
Espoo 2004 HUT-TCS-B21

AUTOMATIC TRANSLATION OF SDL INTO HIGH LEVEL PETRI NETS

Annikka Aalto

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Technical Reports 21
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tekninen raportti 21
Espoo 2004 HUT-TCS-B21

AUTOMATIC TRANSLATION OF SDL INTO HIGH LEVEL PETRI NETS

Annikka Aalto

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FIN-02015 HUT
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

ISBN 951-22-7404-3
ISSN 0783-540X

Multiprint Oy
Helsinki 2004

ABSTRACT: Designing, implementing and testing parallel and concurrent programs have traditionally been complex and error-prone tasks. Due to the concurrency and asynchronous communication within the system, the internal behavior of the system tends to be highly nondeterministic and error conditions may be impossible to reproduce. *Formal methods* address the problem by offering means for exhaustively analyzing all the alternative states of a system.

There are many formal analysis methods, but *reachability analysis* is especially well suited for automatic analysis. In this method, all the reachable states of the system are generated from the model, and then it is checked that they fulfill some desired properties.

One problem with the reachability analysis is the creation of the model. If done by hand, it is a very time consuming and error-prone task. This work describes SDL2PN, a front-end for MARIA reachability analysis tool. The front-end consists of a parser for SDL-96 language and a model generator for MARIA input language. It reads an SDL system description in textual representation and generates a text file containing the high-level Petri net model which can be read and analyzed by the MARIA tool. The combination of SDL2PN and MARIA can be used to analyze even quite large SDL systems without having to manually construct the model of the system.

KEYWORDS: SDL, high-level Petri nets, reachability analysis

CONTENTS

List of Figures

List of Tables

1 INTRODUCTION

In the modern society, computer systems control many aspects of everyday life. Telephone systems, financial systems and medical systems are examples of complex computer systems, which are often unnoticed when operating correctly, but the failure of which may lead to great financial losses, or be even life-threatening. The size and complexity of software systems has increased at a growing pace over the last decades. It is no longer uncommon for tens or even thousands of people to be involved in the construction of a software system.

In many applications, the system is divided to separate components that collaborate to perform a joint task. The parts of the system usually run in parallel, and use *asynchronous communication methods* to communicate with each other. The components of a system may physically reside in the same computer, or be scattered around the world. Asynchronous communication between the components of a system means that processes within the system may send messages to each other at any given moment. Due to the difference in the relative execution speeds of different system components, and dynamic variation in this speed [1], messages may arrive to a process in any order. It is also possible for the messages to travel different routes, in which case they may arrive at their destination in different order than they were sent.

Designing, implementing and testing concurrent programs have traditionally been complex and error-prone tasks. Due to the concurrency and asynchronous communication within the system, the internal behavior of the system tends to be highly nondeterministic and error conditions may be impossible to reproduce. *Formal methods* are notations and analysis techniques that can be used to verify that a system satisfies its specification. *Reachability analysis* is an analysis technique which suits well concurrent systems. The basic idea of reachability analysis is to compute the state space of the system and check the required properties at each state. The state space of a system can easily be represented as a graph, and it is often referred to as the *reachability graph* of the system.

The size of the state space of an industrial size system may outrule exhaustive analysis. The size of the state space can grow exponentially compared to the size of the system. This phenomenon is called *state space explosion*. There are several techniques that can be used to attack the state space explosion problem. *Partial order reduction* methods are based on the observation that some orders in which the concurrent and independently executed events are interleaved are equivalent with respect to the property to be checked. Another approach is to use *symmetry* methods, which exploit symmetries of state spaces. Most reachability analysis tools utilize at least one technique to relief the state space explosion problem.

Another problem besides the vast state space of industrial size programs is, how to construct an analyzable model of a program. Manually constructing the verification model of a large program may not be feasible, especially if the work has to be done by someone who is not very familiar with formal methods

[1]The execution speed of a process may vary considerably depending on the load of the computer system it is run on, and other external phenomena.

or with the formalism applied by the analysis tool. Since most designers are not familiar with other formalisms than the one they are directly working with, it is desirable to automatically generate the verification model out of the program specification. This requires a level of formality from the program specification method itself.

The ITU-T [2] Specification and Description Language [34], SDL for short, is a language with formally defined semantics. It is often used in describing telecommunication protocols, which form a very important class of distributed systems. Because of its formality, SDL is an interesting candidate as an input language for automatic model generation. Automatic translation from SDL to various modelling formalisms have been developed. The complexity of the translation mechanism depends on the selected modelling formalism and the set of SDL constructs allowed.

The input language of the verification tool SPIN [11], PROMELA, has been designed to be quite close to SDL which makes automatic translation from SDL to PROMELA [12] quite straightforward. Intermediate format IF is used in the translation from SDL to PROMELA described in [4]. Automatic translation to PROMELA from TNSDL, a dialect of SDL has also been defined [32].

High level Petri nets are a popular class of modeling formalisms used in reachability analysis tools. A method for translating SDL specifications to M-nets (modular multi-labelled Petri nets) is presented in [9]. Colored Petri nets are used as a modelling formalism in the automatic model generation from SDL described in [5] and [6].

Time is generally a difficult thing to model. Most programming languages have a possibility to set timers and SDL is no exception. An extension to SPIN taking timing considerations into account is presented in [3]. Petri nets as such do not have a concept of time but special classes of high level Petri nets have been developed that can be used to embed timing concepts into the model. Modeling SDL specifications with HTT nets (Hierarchical Timed Typed nets) is presented in [23].

State space explosion problem can be attacked by applying some suitable reduction method at the level of the modeling formalism, but it is also possible to apply reductions at the SDL level as described in [31] and [28]. SDLcheck verification tool [16] utilizes a partial order reduction method defined in [15].

This report presents an SDL front-end for MARIA [20] reachability analysis tool. The front-end consists of an SDL parser described in detail in [18] and a model generator capable of constructing a high-level Petri net [19] model from an SDL specification. This report focuses on the model generator part. Translation rules have been defined and implemented for most of the SDL constructs, including dynamic process creation, procedure calls and timers. To optimize the model with respect to the size of the reachability graph, some SDL level static reductions have also been defined.

[2]ITU-T is the Telecommunication Standardization sector of the International Telecommunication Union and was formerly known as CCITT (Comiteé Consultatif Internationale de Télégraphique et Téléphonique).

2 SDL

SDL (ITU-T Specification and Description Language) was developed as a specification and description language for telecommunication systems. The language was available already in 1976, but since then it has been refined several times and new features have been added. The version of SDL used in this report is SDL-96 as defined in [34] and [35].

SDL-96 has formally defined semantics [2]. A description of a system written in SDL is a *formal specification*. A formal specification is precise and consistent. It is possible to generate an implementation of the system from the specification. SDL is an *implementation-independent* language, so it is possible to create multiple implementations from an SDL specification.

Even though SDL has been designed for specifying telecommunications systems, it is suitable for describing all kinds of reactive, concurrent and distributed systems. A *reactive system* is a system whose behavior can be characterized by its responses to external stimuli. SDL concentrates especially on specifying how the system reacts to various actions in its environment.

SDL has two alternative presentation formats, a graphical presentation (SDL/GR) and a text presentation (SDL/PR). The graphical form of presentation is suitable for representing relationship between system components and examining details of the system in a form that is easy to understand. However, when large or even modest-size systems are considered, representing the whole system in graphical form is out of question. Textual representation is more compact, and does not require special tools for editing. It is also easier to exchange between different tools.

The following description of the SDL language is based on [8], [25], [27], [22] and [34].

2.1 BASIC CONCEPTS OF THE SDL LANGUAGE

An *SDL specification* specifies the structure and behavior of an *SDL system*, and the data used in the system. An SDL system is a collection of *SDL processes* and *communication paths* between them. SDL processes communicate with each other using *signals*, which travel along the communication paths. A process can only send a signal to another process, if there is a communication path between them.

Some communication paths end in *the environment*, which represents the world outside the SDL system. The system can send signals to the environment, and the environment can send signals to the system. No assumptions can generally be made about the environment, it can send signals to the system in a completely random manner.

The processes are grouped into *blocks*. A block can contain other blocks, and this way a hierarchical structure can be obtained. Communication paths that connect blocks to other blocks or to the environment are called *channels*. Channels are by default delaying, but not reordering. If two signals are sent to a channel, it may take arbitrarily long time for them travel through it, but they always come out in the same order as they were put in. It is possible to specify a channel to be non-delaying, which means that when a signal is sent to the channel, it instantly appears at the other end. This doesn't, however, mean that the receiver of the signal handles it instantly.

Figure 2.1 shows an SDL system which consists of three blocks: *Sender*, *Receiver* and *Buffer*. The *Buffer* block is connected to the *Sender* and *Receiver* blocks with bidirectional channels named *c1* and *c2*. Channel *c1* can carry signals of type *Msg* from *Sender* to *Buffer*, and *Ack* signals to the other direction. There are no channels connecting any of the blocks to the environment. This is a common way to avoid the completely nondeterministic behavior of the environment.

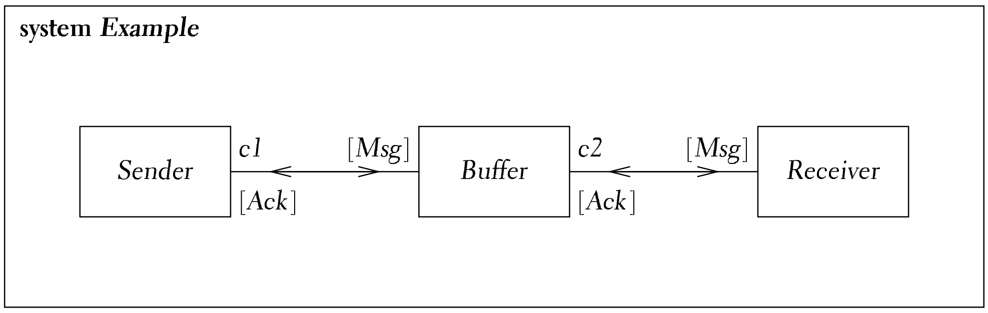


Figure 2.1: An example of an SDL system

The contents of a block are not visible outside it. Figure 2.2 shows the internal structure of the *Receiver* block. It contains two SDL processes, *Transceiver* and *User*. Communication paths which connect processes to each other or to the border of a block are called *signal routes*. They are never delaying. There are two bidirectional signal routes in the *Receiver* block, named *r1* and *r2*.

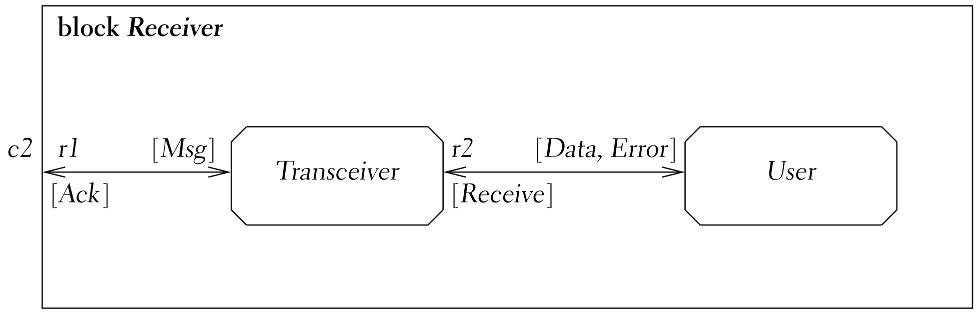


Figure 2.2: An example of an SDL block

2.2 BEHAVIORAL ASPECTS OF THE SDL LANGUAGE

The dynamic behavior of an SDL system is described by *processes*. SDL processes are communicating extended finite state machines (CEFSMs). A CEFSM has a finite set of *states* and *transitions* between the states and may also store data in *variables*. CEFSMs communicate by sending messages.

In SDL the messages used for communication between processes are called *signals*. SDL signals can carry parameters. Each SDL process has an *input queue*, to which arriving signals are stored. The queue has unlimited capacity. The queue works in FIFO (First In, First Out) manner.

The execution of an SDL process starts from an *initial state*, shown as an ellipse in the graphical representation. Each SDL process has exactly one initial state. From the initial state, an *SDL transition* leads to some other SDL state of the process. An SDL transition is a sequence of SDL statements. Examples of SDL statements are OUTPUT for sending signals to other processes, and ASSIGN for assigning a value to a variable.

Figure 2.3 shows the definition of the *User* process. The SDL transition following the initial state contains one OUTPUT statement and one NEXTSTATE statement. The OUTPUT statement is used to send a *Receive* signal. The block definition in Figure 2.2 shows that there is only one signal route capable of carrying a *Receive* signal and it ends in the *Transceiver* process, so the only possible receiver for the signal is the *Transceiver* process. The NEXTSTATE statement indicates the SDL state in which the process moves next. It always ends the SDL transition.

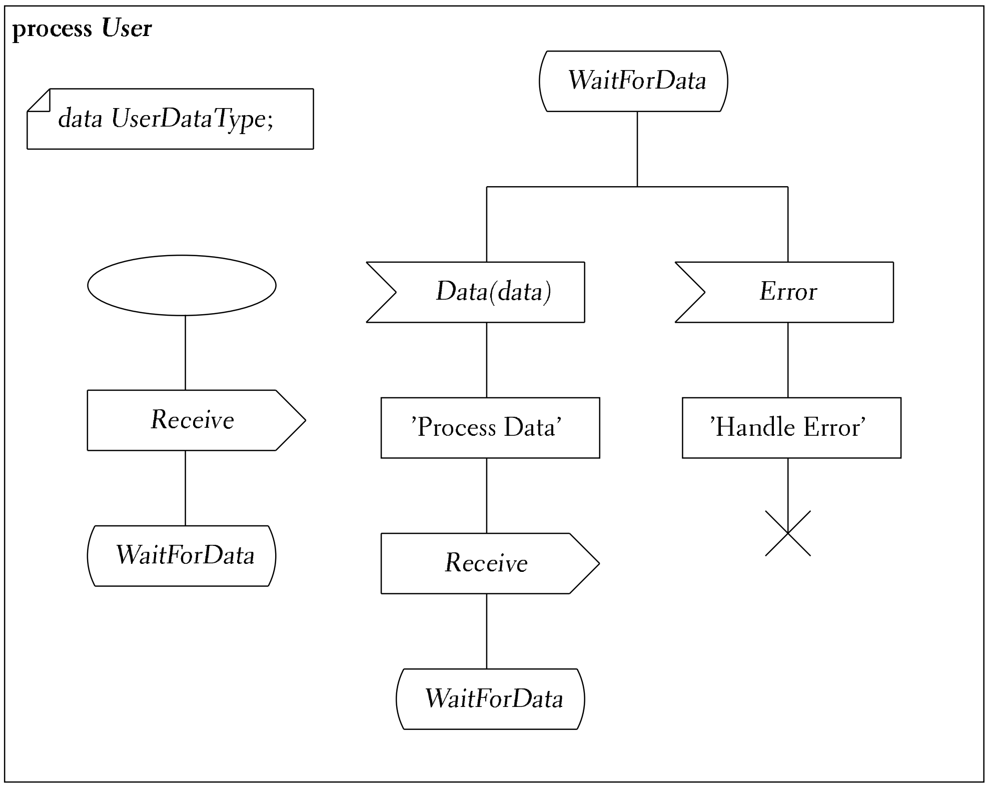


Figure 2.3: An example of an SDL process

2.2.1 Transition Triggers

For each state of an SDL process (except initial state), a set of *trigger conditions* can be specified. Most common trigger condition is the presence of a signal in the input queue.

With a trigger condition, an SDL transition can be specified. If the trigger condition holds for the state the SDL process is in, the process may start to execute the transition. After the transition has been completed, the process enters some SDL state, which may be the same state that the transition was initiated from (it can not, however, be the initial state). The example process in Figure 2.3 has only one SDL state, *WaitForData*, in addition to the initial state.

INPUT Construct

INPUT construct is the most commonly used transition trigger. The trigger condition of an INPUT construct holds, if the first signal in the input queue matches a signal specified by the INPUT construct. The signal is consumed from the input queue.

State *WaitForData* in Figure 2.3 has two INPUT constructs, for signals *Data* and *Error*. When the *User* process receives a *Data* signal, it processes the data (informal text is used to represent this action), and informs the *Transceiver* process that it is ready to receive more data by sending a *Receive* signal. If it receives an *Error* signal, it stops execution with a STOP statement.

An INPUT construct may have variables as parameters, as is the case in the INPUT for signal *Data* in Figure 2.3. The actual parameter values carried by the signal are assigned to the variables. Even if the signal specified by an INPUT construct is supposed to carry parameters, it is possible to omit parameters from the INPUT construct. In this case, the actual parameters carried by the received signal are lost when the signal is fetched from the input queue. An INPUT construct can not have more parameters than the signal it specifies. The types of the variables given as parameters for an INPUT must be compatible with the types of the signal it specifies.

Multiple INPUT constructs can be attached to the same SDL transition. The transition is executed, if any of the INPUT conditions hold.

An INPUT construct may specify more than one signal. This is a shorthand notation for a separate INPUT construct for each signal.

ASTERISK INPUT is shorthand notation for an INPUT construct, which contains all signals not specified by other INPUT or SAVE constructs of the SDL state.

SAVE Construct

SAVE construct is for saving signals for later use. The trigger condition of a SAVE construct holds, if the first signal in the input queue matches a signal specified by the SAVE. The signal is removed from the input queue and saved. When the SDL process moves to next SDL state, the signal is considered again as it was in the input queue. A SAVE does not change the SDL state the process is in.

Figure 2.4 shows an example of a SAVE construct. The *Idle* state has a SAVE construct for signal *Msg*. If the process receives a *Msg* signal while in *Idle* state, it saves the signal and continues waiting for other signals. When it finally moves to *WaitForMsg* state, it can consume the saved signal.

An SAVE construct may specify more than one signal. This is a shorthand notation for a separate SAVE construct for each signal.

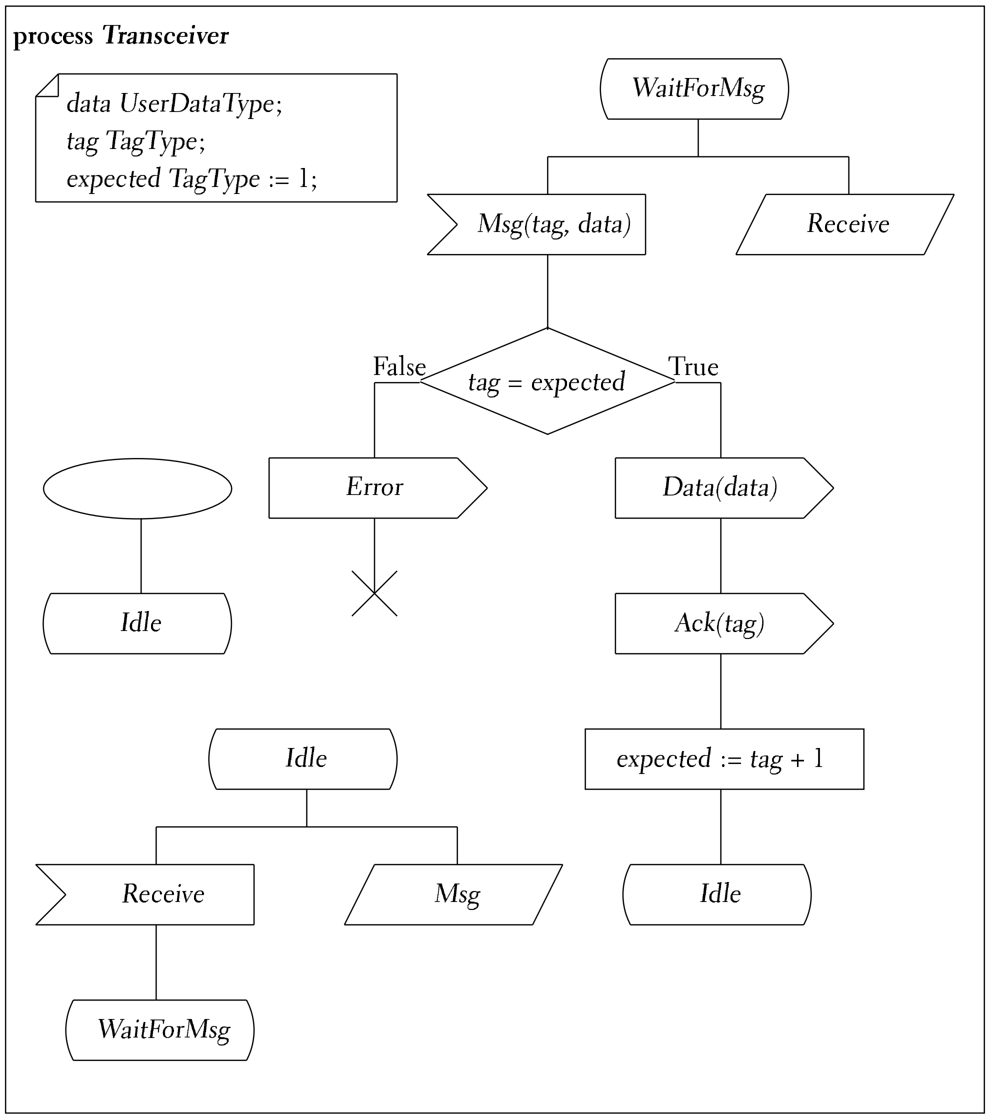


Figure 2.4: An example of an SDL process

ASTERISK SAVE is shorthand notation for an SAVE construct, which contains all signals not specified by other INPUT or SAVE constructs of the SDL state.

A signal can be mentioned in at most one INPUT or SAVE construct in a SDL state. A SDL state can have at most one ASTERISK INPUT or ASTERISK SAVE.

Spontaneous Transition
Spontaneous transition does not expect any signals in the input queue. A spontaneous transition may be triggered any time the process is in the SDL state where the spontaneous transition is declared. Whether or not a spontaneous transition is triggered is independent of the presence of signals in the input queue. There may be multiple spontaneous transitions in same state. Spontaneous transitions are often used to model unreliable parts of the system.

In graphical representation spontaneous transition is shown as a normal input symbol containing keyword NONE. For this reason, it is sometimes called NONE INPUT.

Continuous Signal
With *continuous signal* construct, a trigger condition depending on the state of the system can be created. A continuous signal definition contains a boolean expression referring to variables of the SDL process which contains the continuous signal. When the expression evaluates to true, the SDL transition following the continuous signal definition may be executed.

INPUT and SAVE actions and spontaneous transition have precedence over continuous signal. It is also possible to define priorities for continuous signals in case there are multiple continuous signals in the same SDL state.

Enabling Condition
It is possible to add extra restrictions to an INPUT construct using an *enabling condition*. An enabling condition is a boolean expression referring to variables of the process. The SDL transition associated with the INPUT construct is executed only if the enabling condition evaluates to true. Otherwise the signal is saved instead.

Implicit Signal Consumption
If the first signal in the input queue is not mentioned in any of the INPUT or SAVE constructs of the state the process is in, and the state contains neither ASTERISK INPUT nor ASTERISK SAVE construct, the signal is discarded. This feature is called *implicit signal consumption*. Implicit signal consumption does not change the SDL state the process is in.

2.2.2 Transition Terminators

An SDL transition is a sequence of SDL statements. The last statement of an SDL transition is always a *transition terminator*. Most common transition terminator is the NEXTSTATE statement. A NEXTSTATE statement moves the process to the SDL state identified in the statement.

Other transition terminators are STOP and RETURN statements, described in 2.2.4 and 2.4.3.

2.2.3 Conditional Branching

Conditional branching in an SDL transition is achieved by the DECISION statement. A DECISION statement has a *question expression* and a number of *branches*. A branch consists of an *answer expression* and an SDL transition. The answer expressions are constant ranges. Ranges must be mutually exclusive. When the question expression evaluates to a value within the range of an answer expression, the corresponding SDL transition is executed. Optionally an *else-branch* may be present, which is selected if the question expression does not evaluate to any of the answer expressions.

A DECISION statement is shown as a diamond shape in the graphical representation. Figure 2.4 shows an example of a DECISION statement. When the *Transceiver* process receives a message (*Msg* signal) from a peer entity, it checks whether the message is tagged with an expected tag value. If it is, the *Transceiver* process sends an acknowledgement (*Ack* signal) with the tag value to the sender of the message, and the data part of the message to the *User* process. Then it increments the expected tag value by one and moves to the *Idle* state to wait for the *User* process to request more data with a *Receive* signal. If the message is not tagged with the expected tag value, execution branches to error handling.

Nondeterministic Branching

An ANY DECISION is a decision statement containing neither the question nor the answer expressions. It contains a set of branches, one of which is selected nondeterministically when the statement is interpreted.

Figure 2.5 shows a lossy communication channel implemented using ANY DECISION statement. *LossyChannel* process receives two kinds of signals: *Msg* and *Ack*, and nondeterministically either sends the received signal forward or discards it.

2.2.4 Process Instances

An SDL process may have a number of *instances*. Process instances can be created either at system initialization time or dynamically during runtime. By default, one instance of each process is created at system initialization time. The number of dynamically created process instances is unlimited. It is, however, possible to override these values by specifying the initial number of process instances and/or maximum number of simultaneously existing instances for a process.

The signal routes attach to processes, not process instances. When a signal arrives at a process which has more than one instance, the receiving instance is selected at random.

Process instances can not be separated from each other by name, because instances of a given process all have the same name. Instead, each process instance has a unique ***PId*** (Process Identifier) value, which distinguishes it from any other process instance in the system. All signals carry the **PId** value

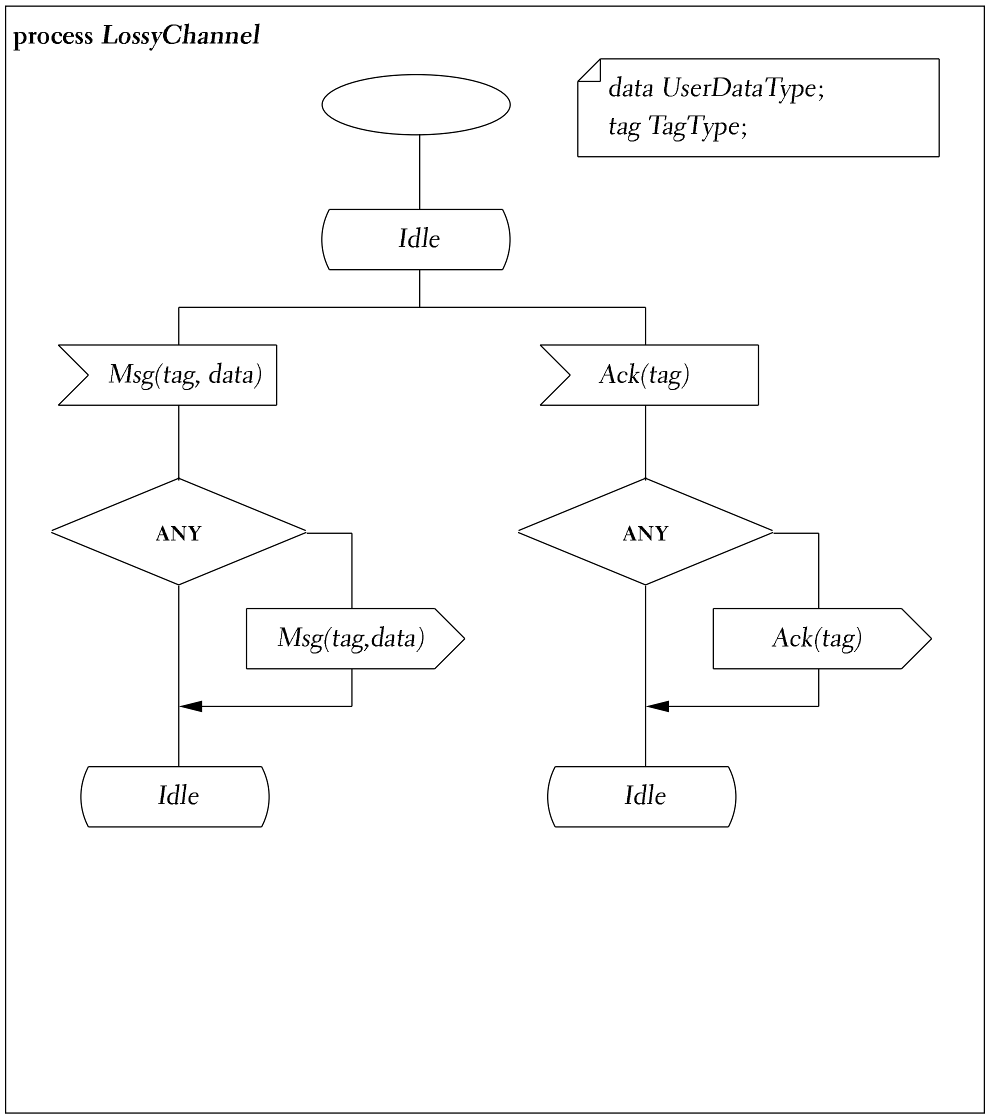


Figure 2.5: SDL definition of a lossy communication channel

of the sender. This allows the receiver to determine which process instance sent the signal and possibly respond to the same instance.

An SDL process may have parameters. Parameter values are bound at process instance creation time. If the process instance is created at system initialization time, the values of parameters are undefined. Process parameters function like read-only variables.

CREATE statement is used to create a process instance during runtime. The creating process and the process to be created must be in the same block. A process can stop its execution by a **STOP** statement. A process can not stop any other processes but itself. When a process executes a **STOP**, it ceases to exist and its **PId** value may be assigned to some newly created process.

The blocks, channels and signal routes are static elements that can not be dynamically created or destroyed. They do not have a concept of instance, either.

2.2.5 Communication Between Processes

SDL processes send signals to each other with **OUTPUT** statement. The **OUTPUT** statement specifies the signal to be sent and values of parameters for the signal.

There may be multiple process instances that can receive a signal, for instance because the signal is sent to a communication path which branches to multiple processes, or because there is more than one instance of the receiving process. In this case, the default version of the **OUTPUT** statement sends the signal to a process instance which is randomly selected among the process instances reachable from the sending process. It is, however, possible to change this behavior using special constructs **TO**, **VIA** and **VIA ALL** with the **OUTPUT** statement.

The **TO** constraint in an **OUTPUT** statement restricts the receiver of the signal to the process instance with **PId** value specified in the **OUTPUT** statement. The four predefined values that can be used are listed in Table 2.1. If there is no reachable process instance with the requested **PId** value, the signal is discarded.

Expression	**Identifies**
SELF	The process instance itself.
SENDER	The sender of signal most recently received by this process instance.
PARENT	The process instance that created this process instance.
OFFSPRING	The process instance most recently created by this process instance.

Table 2.1: Predefined **PId** expressions

The **VIA** construct can be used to restrict the set of channels, signal routes and gates the signal may travel through. Consider a system shown in Figure 2.6 a). All communication paths beginning from the process A are shown in Figure 2.6 b). Statement **OUTPUT** *Sig* **VIA** *r2* first selects the receiving process out of the set of processes reachable using the mentioned signal route. This leaves processes *C* and *D*. One of them is randomly selected, and the process instances are then considered. If the process has exactly one process instance, the signal is delivered to it. If there are more than one process instances, one of them is selected at random. If there are no process instances, the signal is discarded, even if there were instances of the other process which was not selected in the first phase of selecting the receiver for the signal.

Statement OUTPUT *Sig* VIA *c1*, *c3* works the same way, but now the reachable processes are *B* and *D*.

The list of channels, signal routes and gates in the VIA definition is called VIA *path*.

Other OUTPUT statement types send a signal to a single receiver, but the VIA ALL construct can be used to implement multicasting. An OUTPUT statement with a VIA ALL definition sends a copy of the signal to each of the channels, signal routes and gates mentioned in the VIA path.

Considering the example in Figure 2.6, statement OUTPUT *Sig* VIA ALL *c1*, *c3* sends the signal *Sig* to both processes *B* and *D*. If one or both processes have more than one instance, the receiving instance is selected randomly, as in other OUTPUT statement types.

Statement OUTPUT *Sig* VIA ALL *r2* works exactly the same way as without the ALL keyword. Because the VIA path has only one element, only one copy of the signal is sent.

Statement OUTPUT *Sig* VIA ALL *c2*, *c3* is slightly more complicated, because when a signal has been sent to channel *c2*, it still has two possible receivers: processes *C* and *D*. Now it is possible, that one copy of the signal is sent to process *C* (through channel *c2*) and one copy to process *D* (through channel *c3*), or that both copies end to process *D* using different routes, because *D* is reachable through both channels *c2* and *c3*. If both copies of the signal are sent to process *D*, they may still be received by different process instances if there is more than one instance of *D*. It is also possible that they will both be received by the same process instance.

2.2.6 Timers

An important concept of the SDL language is a *timer*. A timer is an object which can generate signals to the input queue of the owning process instance.

A timer can be *active* of *inactive*. When a timer is created, it is inactive. Activation of a timer is done using SET statement. When a timer is activated, it is given either the duration of time after which it should expire, or the actual time.

When a timer expires, it puts a signal to the input queue of the process instance. The signal has the name of the timer which generated it. The timer does not move back to the inactive state until the the signal has been consumed. ACTIVE expression can be used to check whether a timer is in active state.

A timer can be set to inactive state by RESET statement. Resetting an inactive timer has no effect. If the timer has already expired, resetting it removes the signal generated by it from the input queue of the process. If the timer has not yet expired, it is just moved to inactive state.

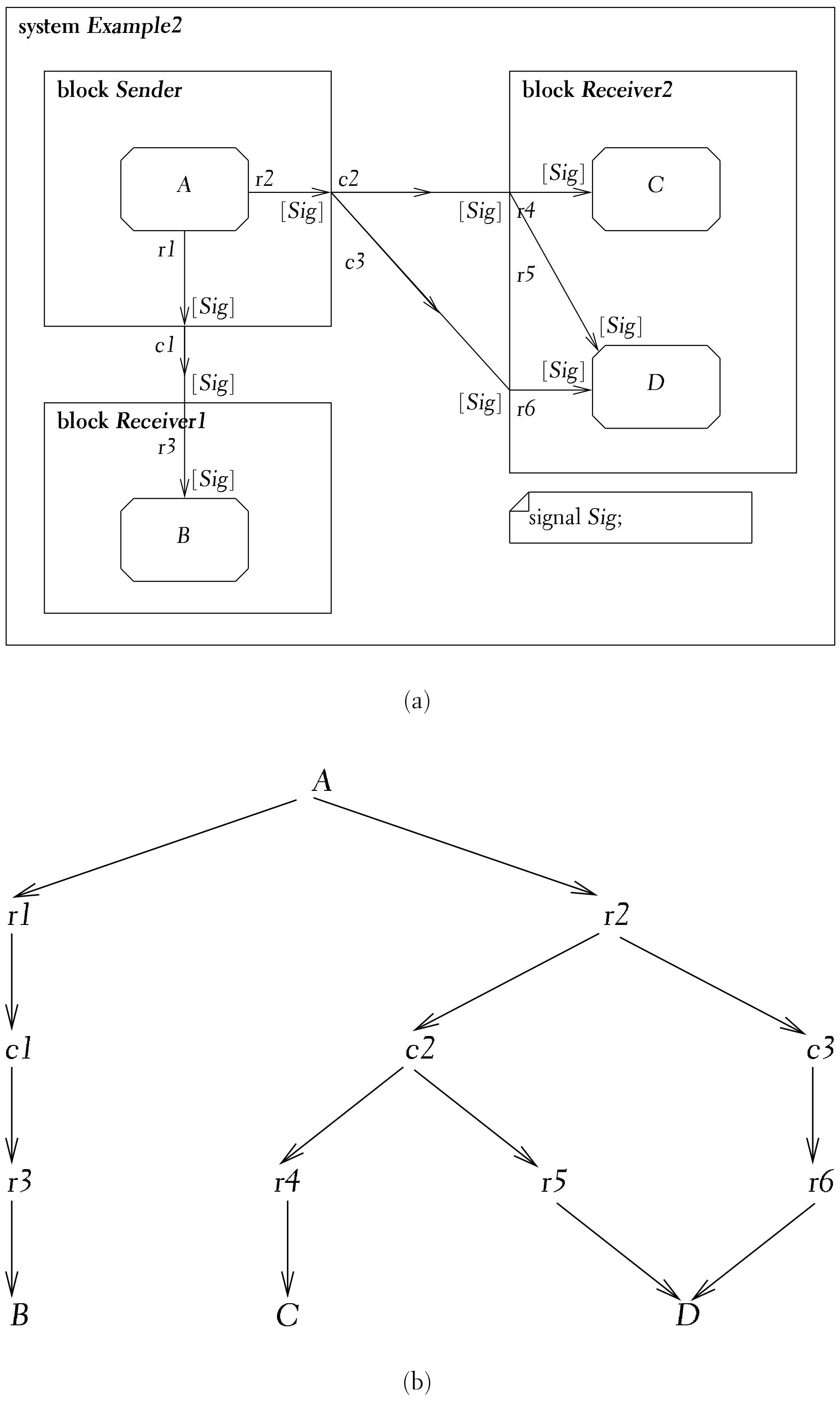


Figure 2.6: Communication paths in an SDL system

A timer can have parameters. The parameter values must be given in all **SET** and **RESET** statements and **ACTIVE** expressions. Parameter values are used in identifying the timer, in addition to name. Parameter values are also included as parameters of the signal generated by the timer.

The signal generated by a timer is just like other signals. It can be consumed by an **INPUT** construct, saved or discarded. Expiration of a timer is not considered a transition trigger, because no SDL transition is executed until the signal generated by the timer is consumed from the input queue.

2.3 DATA IN SDL

A SDL process may contain data in form of *variables*. The variables are of different data types, called *sorts* in SDL. There is a predefined set of basic sorts, such as *Boolean*, *Integer* and *Character* sorts, and new sorts may be created either by composing from the existing ones, or defining them from scratch.

Structural types can be created using the SDL *struct* construct.

SDL has some predefined *generators*, which can be used to create new sorts easily. Figure 2.7 shows the definition of a list of *Integer* values using the predefined *String* generator. It defines a new type *IntList*. The *emptylist* literal is used to represent an empty list of this sort. Other predefined generators are *Array* and *Powerset*.

```
newtype IntList
    String(Integer, emptylist);
endnewtype IntList;
```

Figure 2.7: An SDL type definition

The data type definition capabilities of the language are quite versatile, but it is also possible to combine other formalisms (such as ASN.1) to SDL to extend the data type definition capabilities.

2.4 STRUCTURE OF AN SDL SYSTEM

Systems specified in SDL are usually too large to get an overview of if all the processes are considered at once. Blocks offer a way to group processes to more manageable sets, but also other structuring concepts are available.

2.4.1 Block Substructure

Using only one level of abstraction will easily lead to unpractically large number of blocks in the system. *Block substructure* can be used to create blocks within blocks. This makes it possible to group interconnected blocks inside a superblock, this way reducing the number of system-level blocks.

If a block does not have a substructure, it can only contain processes and signal routes connecting the processes to each other or to the environment of the block. *Connections* define how the signal routes are connected to the channels outside the block. This kind of blocks are called *leaf blocks*.

A block with substructure may itself contain other blocks, which are interconnected by *subchannels*. Subchannels are defined as normal channels, and they are connected to the channels in the environment of the block by *channel connections*.

In SDL, a block may contain both processes and a block substructure. They are then two different views to the same block, and only one of them can exist in a system at any time.

2.4.2 Channel Substructure

If partitioning blocks is not enough, it is possible to create further structure to channels as well. *Channel substructure* is similar concept to block substructure, but it defines the internal structure of a channel.

A channel substructure can be used to change the behavior of a channel by creating an internal structure for it. A channel substructure may contain blocks and subchannels, just like a block substructure. The difference is in the way how subchannels are connected to the outside world.

Figure 2.8 shows an example of a channel substructure definition. It is a definition of a system with two blocks: *Sender* and *Receiver* connected with a lossy channel. Because SDL channels do not lose signals, the channel *c* has a substructure which adds the signal losing feature. The substructure contains only one block, which contains a process *LossyChannel* (Figure 2.5) implementing the actual signal losing behavior.

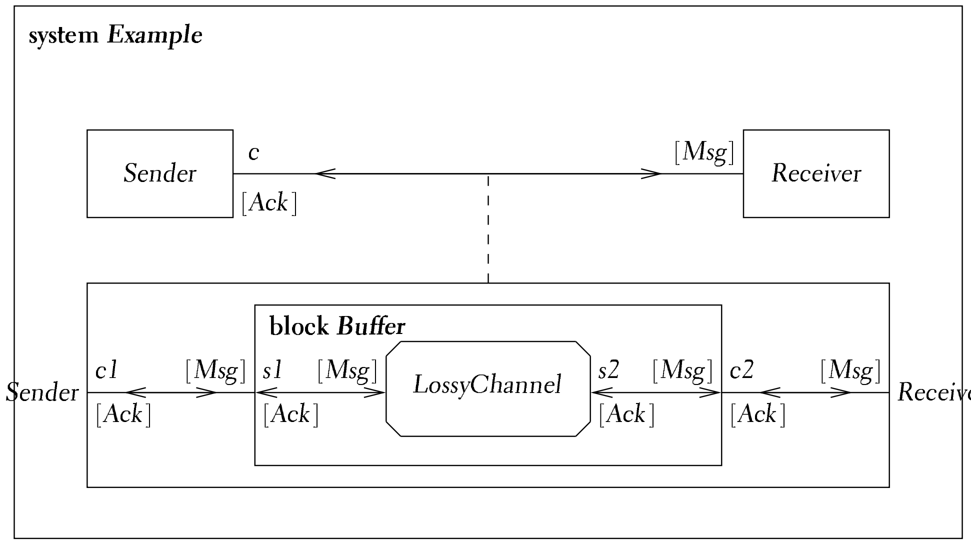


Figure 2.8: An example of SDL channel substructure

2.4.3 Procedure

There are two ways of splitting an SDL process to smaller pieces: *procedures* and *services*.

A procedure concept in SDL is similar to the one known in programming languages. A procedure is a state machine within a process, which is created when a *procedure call* (CALL statement) is interpreted, and ceases to exist when a RETURN statement is interpreted within the procedure. While the procedure is being interpreted, the calling process instance is suspended. A procedure uses the input queue of the calling process instance.

A procedure may have two kinds of parameters. IN-parameters are like local variables. Value for an IN-parameter may be given in procedure call, but it is not mandatory. If value for a paramater is not given, default value is used. IN/OUT-parameters may be used to return values from the procedure. In a procedure call, each IN/OUT parameter of the procedure must be replaced with a variable of the calling process or procedure. When the procedure returns, the values of the variables are replaced with the values of the IN/OUT parameters. A procedure may also return a value as an expression associated with the RETURN statement.

2.4.4 Service

In addition to using procedures, it is also possible to split an SDL process to smaller pieces by using *services*. An SDL service is, like an SDL process, a set of SDL states and transitions. Each service in a process must contain a distinct set of states.

Only one service may be executing a transition at a time. The active service may change only when a transition terminator is reached and an SDL state is entered.

2.5 TYPES

SDL uses the *type* concept extensively. There can be types of different entity kinds, for example process- or block types. A type defines the common properties of a category of entities, and it may be instantiated any number of times. A block type may, for example, be instantiated to two different blocks. Even if these two blocks are internally identical, they are different blocks, and they may be used in different parts of the system.

The type information is static. For example processes can not be created from a process type dynamically, like process instances can be created for a process.

When process types are used, the signal routes to which the process is attached can not be known when the type is specified. Instead, *gates* are used. In the block definitions the signal routes are then connected to the gates.

3 PETRI NETS

Petri net theory is a formalism suitable for describing systems of distributed and combinatorial nature. Petri nets are particularly well suited for discrete event systems. Petri net theory has emerged from the work of Carl Adam Petri in the early 60's. A compelling fact in this formalism is the way how the basic aspects of distributed systems are identified both conceptually and mathematically. The following description of nets is based on [30], [26], [10] and [19].

In Petri net theory, the state of the system is seen as a composition of local states of it's parts. A change of state is also local — its extent does not depend on the global state at which it occurs. The actions changing states are called *transitions* [1].

Definition 3.1 (Nets) *A net is a 3-tuple $N = \langle S, T; F \rangle$, where S is a set of* places, *T is a set of* transitions *and F is a* flow relation. *Following properties hold for the sets:*

1. $S \cup T \neq \emptyset$ *and* $S \cap T = \emptyset$
2. $F \subseteq (S \times T) \cup (T \times S)$

A net is an ordered bipartite directed graph without isolated nodes. In graphical notation the places are usually represented as circles, and transitions as rectangles. Flow relation is represented with directed arcs. The places represent local states of the system, and the flow relation determines which states a transition has an effect on. An arc coming from a place to a transition is called an *input arc*, and an arc from a transition to a place is called an *output arc*.

In Petri net theory a lot of work has been done using low level nets with "black" tokens, meaning that the tokens can not be distinguished from each other. This kind of nets are called *Place/Transition nets* (P/T nets). A P/T net is a net as defined in Definition 3.1 associated with an *initial marking*, the initial distribution of tokens to the places. The tokens are represented graphically as black dots.

Figure 3.1 shows a simple example of a P/T net. The example shows a model of a four-slot buffer, where the tokens in the slot places represent messages that the sending end may put to the buffer, and the receiving end may remove from the buffer. Each slot of the buffer has to be modeled with a separate place, because otherwise there is no way to tell which message is in which slot. The fact that tokens have no identities makes it difficult to model message contents. Because a buffer slot usually fits only one message at a time, complement places are needed to restrict the number of tokens in each slot place to one.

[1]The concepts *transition* and *sort* are used in SDL in a completely different meaning than in net theory. Unless otherwise stated, the occurrences of words *transition* and *sort* in this chapter refer to *net transitions* and *net sorts*

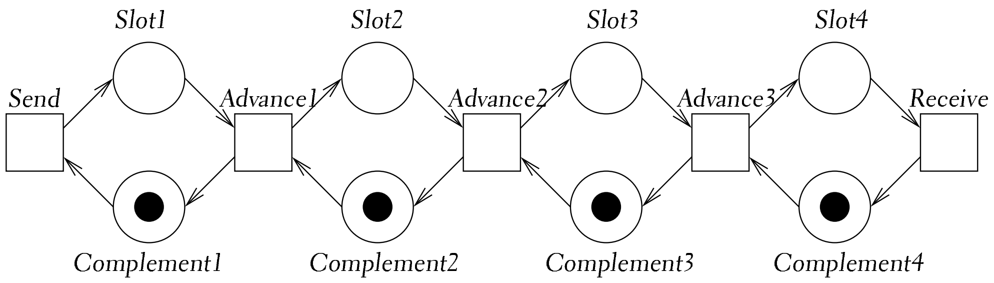


Figure 3.1: A Place/Transition net modeling a four-slot buffer

A transition is said to be *enabled* if there are tokens in all its input places (the places from which there is an input arc to the transition). When a transition is enabled, it can be *fired*. When a transition is fired, a token is removed from all its input places and a token is inserted to all its output places. (Note that this definition applies only to P/T nets without weights and capacities.)

P/T nets are very illustrative when modeling simple systems, but with even slightly more complicated examples the models become unpractically large. P/T nets lack the ability to model *individuals*. *Predicate/Transition nets* (Pr/T nets) are a class of high-level Petri nets which allow modeling dynamical systems with a set of individuals ordered by functions and relations.

While the tokens used in P/T nets were simple unnamed entities, in Pr/T nets they are tuples of terms, and thus tokens can be identified. With each arc is associated an *arc expression*, which tells how many, and what kind of tokens are removed from or inserted to the associated place when the transition is fired. Also *gate expressions* can be added to transitions. A gate expression (also called *transition selector* or *firing condition*), is a truth-valued expression. The transition can be fired only when the gate expression evaluates to true.

Although a Pr/T model of a system can be considerably more compact than a P/T model, it can not be used to model anything that could not be modeled using a P/T net. In fact, each Pr/T net can be *unfolded* to a P/T net, and a P/T net can be *folded* to a Pr/T net.

Pr/T nets are used in for example `PROD` [33] reachability analysis tool, and they can be very efficied in modelling even quite complex systems. However, when the aim is to generate models from a description in another language, the modelling formalism falls short in expressing the data types of the source language. If complex data types have to be represented by, for example, tuples of integers as done in `PROD`, unnecessary complexity is introduced in the models. They become both harder to understand, and more time-consuming to analyze.

3.1 ALGEBRAIC SYSTEM NETS

Algebraic system nets allow the use of many kinds of data types, called *sorts*. The use of algebraic techniques makes it possible to describe both the syntax and the semantics of the terms appearing in the nets. The syntax of an algebraic system is defined in a *signature*, which specifies the symbols used in terms but does not give any interpretation for them. The semantics are defined by the *algebra*, which gives a correspondence between the operation symbols given in the signature and the actual interpretation for them.

Definition 3.2 (Signatures) *A signature* $\mathbf{S} = \langle \mathcal{S}, \mathcal{F} \rangle$ *consists of*

1. *A non-empty set* $\mathcal{S}$ *of sort names (*sorts*).*
2. *A pairwise disjoint family* $\mathcal{F} = \bigcup_{\sigma \in \mathcal{S}^*, s \in \mathcal{S}} \mathcal{F}_{\sigma,s}$ *of* operation symbols.

An operation symbol $f \in \mathcal{F}_{s_1,\dots,s_n,s}$ denotes an operation from sorts $s_1, \dots, s_n$ to sort s . The set $\mathcal{F}_{\epsilon,s}$, where ϵ denotes the empty sequence, is called the set of $\mathbf{S}$ *-constant symbols of sort* s [2].

Definition 3.3 (Variables) *A pairwise disjoint family* $\mathcal{V} = \bigcup_{s \in \mathcal{S}} \mathcal{V}_s$ *of symbols such that* $\mathcal{V} \cap \mathcal{F} = \emptyset$ *is called a family of* $\mathbf{S}$ *-variables.*

Using operation symbols, constant symbols and variables, we can create $\mathbf{S}$ -terms as follows. Algebraic terms can be seen as sequences of symbols, the interpretation for them is defined separately.

Definition 3.4 (Terms) *The set* $\mathbf{T}^{\mathbf{S}}_s(\mathcal{V})$ *of* $\mathbf{S}$ -terms of sort s over variables $\mathcal{V}$ *is the minimal set defined inductively by the following rules:*

1. $\mathcal{V}_s \subseteq \mathbf{T}^{\mathbf{S}}_s(\mathcal{V})$
2. *For* $n \geq 0$, *if* $f \in \mathcal{F}_{s_1,\dots,s_n,s}$ *and* $T_i \in \mathbf{T}^{\mathbf{S}}_s(\mathcal{V})$ *for* $1 \leq i \leq n$, *then* $f(T_1, \dots, T_n) \in \mathbf{T}^{\mathbf{S}}_s(\mathcal{V})$.

The set $\mathbf{T}^{\mathcal{S}}_s(\emptyset)$ *is the set of* $\mathcal{S}$-ground terms of sort s.

An algebra corresponding to a signature gives an interpretation for the sorts and operations. It assigns each sort a domain and each operation symbol a function. For example, a boolean sort would be assigned a domain $\mathcal{D}^{\mathcal{A}}_b = \{\bot, \top\}$.

Definition 3.5 (Algebras) *Let* $\mathbf{S} = \langle \mathcal{S}, \mathcal{F} \rangle$ *be a signature. A* $\mathbf{S}$-algebra $\mathcal{A} = \langle \mathcal{D}^{\mathcal{A}}, \mathcal{F}^{\mathcal{A}} \rangle$ *consists of:*

1. *A pairwise disjoint family* $\mathcal{D}^{\mathcal{A}} = \bigcup_{s \in \mathcal{S}} \mathcal{D}^{\mathcal{A}}_s$ *of non-empty* domain sets *for sorts.*
2. *A pairwise disjoint family* $\mathcal{F}^{\mathcal{A}} = \bigcup_{f \in \mathcal{F}} f^{\mathcal{A}}$ *of operations. For all* $f \in \mathcal{F}_{s_1,\dots,s_n,s}$ *with* $n \geq 0$, *the operation* $f^{\mathcal{A}}$ *is a mapping* $f^{\mathcal{A}} : D^{\mathcal{A}}_{s_1} \times \cdots \times D^{\mathcal{A}}_{s_n} \rightarrow D^{\mathcal{A}}_s$.

Definition 3.6 (Assignments) *Let* $\mathcal{A}$ *be an algebra. The set*

$$V^{\mathcal{A}}(\mathcal{V}) = \left\{ v \| v : \bigcup_{s \in \mathcal{S}} (\mathcal{V}_s \rightarrow \mathcal{D}^{\mathcal{A}}_s) \right\}$$

is the set of all assignments *to the variables of the family* $\mathcal{V}$.

[2]The convention used in this document is that sequence indices are written in ascending order. When the index of the last element of a sequence is smaller than the index of the first element, the sequence is empty.

An assignment basically fixes a value with each variable. Given an assignment, $\mathbf{S}$ -terms can be evaluated as follows:

Definition 3.7 (Evaluation of Terms) *An assignment $v \in V^{\mathcal{A}}(\mathcal{V})$ to variables is extended to the corresponding* evaluation of terms

$$e_v^{\mathcal{A}} : \bigcup_{s \in \mathcal{S}} \left(\mathbf{T}_s^S(\mathcal{V}) \to \mathcal{D}_s^{\mathcal{A}} \right)$$

by the following inductive definition for each $T \in \mathbf{T}_s^S(\mathcal{V})$:

1. *If $T \in \mathcal{V}_s$, then*
$$e_v^{\mathcal{A}}(T) = v(T)$$
2. *If $T = f(T_1, ..., T_n)$, $f \in \mathcal{F}_{s_1,...,s_n,s}$ and $T_i \in \mathbf{T}_s^S(\mathcal{V})$ for $1 \le i \le n \ge 0$, then*
$$e_v^{\mathcal{A}}(T) = f^{\mathcal{A}}(e_v^{\mathcal{A}}(T_1), ..., e_v^{\mathcal{A}}(T_n))$$

For ground terms all evaluations yield the same value, because variables are not involved.

Multi-set signatures and multi-set algebras are special cases of signatures and algebras. The set of sorts in a multi-set signature is divided to *basic sorts* and *multi-set sorts*, in such a way that for each multi-set sort there exists a corresponding basic sort. The domain of a multi-set sort is the set of all multisets over the domain of the corresponding basic sort. More formal definition is as follows.

Definition 3.8 (Multi-set Signatures) *Let $\mathbf{S} = \langle \mathcal{S}, \mathcal{F} \rangle$ be a signature with a finite set of sorts $\mathcal{S}$. Let $\mathcal{S}_\beta, \mathcal{S}_\mu \subseteq \mathcal{S}$ such that $\mathcal{S}_\beta \cup \mathcal{S}_\mu = \mathcal{S}$ and $\mathcal{S}_\beta \cap \mathcal{S}_\mu = \emptyset$, and let $\mu : \mathcal{S}_\beta \to \mathcal{S}_\mu$ be a bijective mapping from* basic sorts *$\mathcal{S}_\beta$ to* multi-set sorts *$\mathcal{S}_\mu$. Then*

$$\mathbf{S}_\mu = \langle \mathcal{S}, \mathcal{F}, \mu \rangle$$

is a multi-set signature.

A multi-set algebra is a straightforward extension of an algebra. Based on a multi-set signature, it requires that the domain set of each of its multi-set sorts is the set of multi-sets over the domain set of the corresponding basic sort.

Definition 3.9 (Multi-set Algebras) *Let $\mathbf{S}$ be a signature and $\mathbf{S}_\mu$ be a corresponding multi-set signature. An $\mathbf{S}$-algebra $\mathcal{A} = \langle \mathcal{D}^{\mathcal{A}}, \mathcal{F}^{\mathcal{A}} \rangle$ is an* $\mathbf{S}_\mu$-algebra *if for each $s \in \mathcal{S}_\beta$, $\mathcal{D}_{\mu(s)}^{\mathcal{A}} = \mathcal{M}(\mathcal{D}_s^{\mathcal{A}})$, where $\mathcal{M}(D_s^{\mathcal{A}})$ denotes the set of all multi-sets over $D_s^{\mathcal{A}}$.*

Using the concept of multi-set algebras it is possible to define the structure of an algebraic system net.

Definition 3.10 (Algebraic System Nets) *An* algebraic system net

$$\Sigma = \langle \mathcal{N}, \mathcal{A}, \mathcal{V}, i \rangle$$

over $\mathcal{A}$ consists of

1. *A net $\mathcal{N} = \langle \mathcal{P}, \mathcal{T}; \mathcal{F} \rangle$, where*

   (a) *$\mathcal{P}$, a finite pairwise disjoint family of sort-indexed places $\mathcal{P} = \bigcup_{s\in\mathcal{S}_\mu}\mathcal{P}_s$, is a set of $\mathbf{S}_\mu$-variables whose assignments are multi-set valued.*
   (b) *$\mathcal{T}$, a finite set of transitions, is disjoint from the family of places $\mathcal{T}\cap\mathcal{P}=\emptyset$.*
   (c) *$\mathcal{F}\subseteq(\mathcal{T}\times\mathcal{P})\cup(\mathcal{P}\times\mathcal{T})$ is a* flow relation.

2. *An $\mathbf{S}_\mu$-algebra $\mathcal{A}$ for a multi-set signature $\mathbf{S}_\mu = \langle\mathcal{S}_\beta\cup\mathcal{S}_\mu,\mathcal{F},\mu\rangle$; one basic sort $b\in\mathcal{S}_\beta$ is the Boolean sort of truth values with $\mathcal{D}_b^{\mathcal{A}}=\{\bot,\top\}$.*

3. *A sorted $\mathbf{S}_\mu$-variable set $\mathcal{V}=\bigcup_{s\in\mathcal{S}}\mathcal{V}_s$ such that $\mathcal{P}_s\cap\mathcal{V}_s=\emptyset$ for all $s\in\mathcal{S}_\mu$.*

4. *A* net inscription *$i:(\mathcal{P}\cup\mathcal{T}\cup\mathcal{F})\rightarrow\bigcup_{s\in\mathcal{S}}\mathbf{T}_s^{\mathbf{S}_\mu}(\mathcal{V})$ such that*
   (a) *$i(p)\in\mathbf{T}_s^{\mathbf{S}_\mu}(\emptyset)$ for each $s\in\mathbf{S}_\mu$ and $p\in\mathcal{P}$ are the* initialization expressions.
   (b) *$i(t)\in\mathbf{T}_b^{\mathbf{S}_\mu}(\mathcal{V})$ for each $t\in\mathcal{T}$, where b is the Boolean sort are the* transition guards.
   (c) *$i(f)\in\mathbf{T}_s^{\mathbf{S}_\mu}(\mathcal{V})$ for each $f\in\mathcal{F}$ such that $f=\langle p,t\rangle$ or $f=\langle t,p\rangle$ where $s\in\mathcal{S}_\mu$, $p\in\mathcal{P}_s$ and $t\in\mathcal{T}$ are the* arc expressions.

Figure 3.2 shows the four-slot buffer from Figure 3.1 modeled as an algebraic system net. In this example, three sorts are used:

Natural_t Natural numbers, usual mathematical comparison operations.

Message_t *Message_t* data type has two literals, $\{0,1\}$. It is used to represent the contents of messages moving in the system.

MessageBuffer_t *MessageBuffer_t* is a variable-length buffer of items of sort *Message_t*. It has three operators:

add *MessageBuffer_t, Message_t* $\rightarrow$ *MessageBuffer_t* adds the message to the end of the buffer.

remove *MessageBuffer_t* $\rightarrow$ *MessageBuffer_t* removes a message from the beginning of the buffer.

count *MessageBuffer_t* $\rightarrow$ *Natural_t* returns the number of messages in the buffer.

The four-slot buffer, the modelling of which required four places and three transitions using low level nets, can now be modeled with a single variable of type *MessageBuffer_t*. It is stored in a place named *Slots*. *Messages* place is used to store an instance of all possible messages that can be sent. The *Send* transition selects one of the messages, and adds it to the buffer. The *Send* transition is augmented with a gate expression for ensuring that the buffer is not full. Similarly, the *Receive* transition removes a message from the buffer. It has a gate expression checking that there are messages in the buffer.

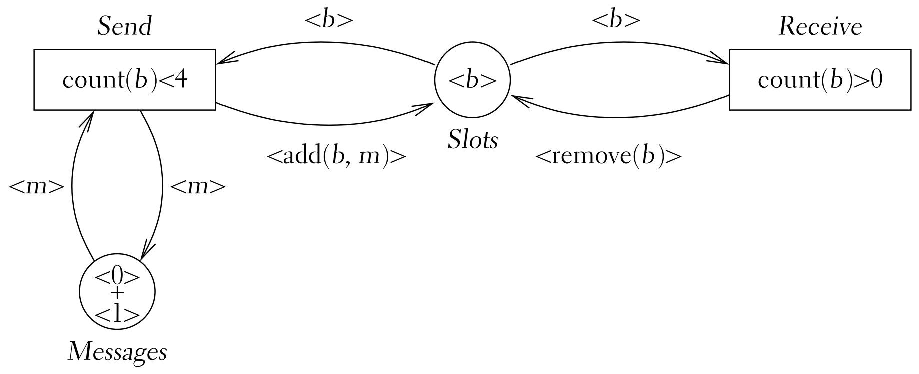


Figure 3.2: An algebraic system net modelling a four-slot buffer

The same structure can be used to represent arbitrarily long buffers, and arbitrarily complex messages.

A model of a system is not very useful if it can not be used to model the dynamic behavior of the system. In Petri nets, the global state of the system is modeled by *marking*, the contents of the places in the net.

Definition 3.11 (Marking, Local Marking and Tokens) *A* marking *of an algebraic system net Σ is a mapping*

$$M : \bigcup_{s \in \mathcal{S}_\mu} \left(\mathcal{P}_s \to \mathcal{D}_s^{\mathcal{A}} \right)$$

The set of all markings M is denoted by M_Σ. For each $s \in \mathcal{S}_\beta$ and $p \in \mathcal{P}_{\mu(s)}$, we call $M(p)$ the local marking *of place p and the items $d \in \mathcal{D}_s^{\mathcal{A}}$ tokens of sort s. If $M(p)(d) = n$, we say that the place p contains n tokens carrying the value d.*

Initial marking defines the initial state of the system. It fixes an evaluation of ground terms with each place, yielding the initial distribution of tokens in the model.

Definition 3.12 (Initial Marking) *Let $v_\emptyset \in V^{\mathcal{A}}(\emptyset)$ be an empty assignment. The marking $M_0 : \mathcal{P} \to \mathcal{D}^{\mathcal{A}}$ with*

$$M_0 : \bigcup_{s \in \mathcal{S}_\mu} \left\{ \left\langle p, e_{v_\emptyset}^{\mathcal{A}}(i(p)) \right\rangle \| p \in \mathcal{P}_s \right\}$$

is called the initial marking *of Σ.*

The global state of the system (i.e. the marking of the net) changes when a transition is *fired*. A transition can be fired only when it is *enabled*. A transition is enabled when the transition guard holds, and there are enough tokens in each of its input places. To define the enabling rule more formally, the notion of *input effect* is needed.

Definition 3.13 (Input and Output Effect) *Let Σ be an algebraic system net, $t \in \mathcal{T}$ a transition and $v \in V^{\mathcal{A}}(\mathcal{V})$ an assignment. The two substitutions $t_v^-, t_v^+ : \bigcup_{s \in \mathcal{S}_\beta} (\mathcal{P}_{\mu(s)} \to \mathcal{D}_{\mu(s)}^{\mathcal{A}})$ are called the* input effect *and the* output effect, *respectively, and they are defined by:*

$$
\begin{aligned}
t_v^-(p) &= \begin{cases} e_v^{\mathcal{A}}(i(\langle p,t\rangle)) & \text{if } \langle p,t\rangle \in \mathcal{F} \\ \mathcal{D}_s^{\mathcal{A}} \to \{0\} & \text{otherwise} \end{cases} \\
t_v^+(p) &= \begin{cases} e_v^{\mathcal{A}}(i(\langle t,p\rangle)) & \text{if } \langle t,p\rangle \in \mathcal{F} \\ \mathcal{D}_s^{\mathcal{A}} \to \{0\} & \text{otherwise} \end{cases}
\end{aligned}
$$

An assignment to variables of a transition t is called a *mode for* t.

Definition 3.14 (Enabling Rule) *Let Σ be an algebraic system net, M its marking, $t \in \mathcal{T}$ a transition of Σ and $v \in V^{\mathcal{A}}(\mathcal{V})$ an assignment. Transition t is* enabled *in mode v at marking M of Σ if the following conditions hold for each $s \in \mathcal{S}_\mu$ and $p \in \mathcal{P}_s$:*

1. *$e_v^{\mathcal{A}}(i(t)) = \top$; i.e. the transition guard holds*
2. *$t_v^- \leq M(p)$; i.e. each input place contains enough tokens*

Definition 3.15 (Firing Rule) *Let Σ, M, $t \in \mathcal{T}$ and $v \in V^{A}(\mathcal{V})$ be such that t is enabled in mode v at marking M of Σ. The firing of transition t in mode v at marking M produces a marking*

$$M' = \bigcup_{s \in \mathcal{S}_\mu} \left\{ \langle p, M(p) - t_v^-(p) + t_v^+(p)\rangle \| p \in \mathcal{P}_s \right\}$$

The fact that marking M'' is the result of firing t in mode v at marking M can be written $M[t_v\rangle M''$.

Using the firing rule it is possible to determine the set of markings that are reachable in the model starting from the initial marking.

3.2 ANALYSIS OF ALGEBRAIC SYSTEM NETS

The aim of the reachability analysis of an algebraic system net is to compute the graph of all the system states that can be reached from its initial state.

Definition 3.16 (Reachable States) *Let Σ be an algebraic system net and M_0 the initial marking of Σ. The set of reachable states of Σ is the smallest set $R \subseteq \mathcal{M}_\Sigma$ fulfilling the following conditions:*

1. *$M_0 \in R$*
2. *$\{M' \| M[t_v\rangle M'\} \subseteq R$ for all $M \in R$, $t \in \mathcal{T}$ and $v \in V^{\mathcal{A}}(\mathcal{V})$ such that t is enabled in mode v at marking M.*

Definition 3.17 (Reachable Actions) *Let Σ be an algebraic system net, M_0 the initial marking of Σ and R the set of reachable states of Σ. The set of* reachable actions *of Σ is defined to be the smallest set $E \subseteq R \times (\mathcal{T} \times V^{\mathcal{A}}(\mathcal{V})) \times R$ for which*

$$\{\langle M, \langle t, v\rangle, M'\rangle \| M[t_v\rangle M'\} \subseteq E$$

for all $M \in R$, $t \in \mathcal{T}$ and $v \in V^{A}(\mathcal{V})$ such that t is enabled in mode v at M.

Definition 3.18 (Reachability Graph) *Let Σ be an algebraic system net and M_0 the initial marking of Σ. Let R be the set of reachable states and E the set of reachable actions of Σ. The* reachability graph *of Σ is the directed graph $G = \langle R, E \rangle$.*

After the reachability graph for a model has been constructed, it is possible to analyze it to see whether it fulfills some desired properties. These properties can be stated, for example, using temporal logics.

Exhaustive reachability analysis, which considers each possible state in the reachability graph, is the most complete analysis method. The main drawback with exhaustive reachability analysis is the *state space explosion* problem, meaning that the number of reachable states is too large to be analyzed in a reasonable amount of time. This stems from the fact that the size of the reachability graph grows very fast (exponentially) with the size of the system. The state space explosion problem can be alleviated by using some suitable method of selecting only subset of states for analysis, where it is guaranteed that the states left out have the properties to be checked if and only if the states that have been selected have them. *Partial order reduction* methods are this kind of methods.

Some properties can be checked while generating the reachability graph, and thus the graph generation can be stopped when a state violating the specific property is found. Now the whole reachability graph does not need to be generated if the property does not hold.

The system can also be *simulated* using the model. In simulation the whole reachability graph does not need to be generated, because only a small subset of possible actions will be carried out. Simulations typically generate a trace of execution, always performing an action on the state generated by the previously performed action. The next action can be selected either randomly, chosen by the user, or selected using some suitable heuristic.

3.3 EXTENSIONS TO ALGEBRAIC SYSTEM NETS

Algebraic system nets can be extended in various ways to bear more similarity to traditional programming languages. This may be beneficial if they are used in modelling systems described using programming language -like methods. Now the construction of a net model from the initial system model is more straightforward and can be more easily automated. Two extensions to algebraic system nets are presented here: error checking and short-circuit evaluation. [19]

3.3.1 Error Checking

An *error algebra* is an algebra $\mathcal{A}$ augmented with the *undefined symbol* $\epsilon \notin \mathcal{D}^{\mathcal{A}}$. For all the operations, whenever an argument equals ϵ, so does the result. Also assignments are allowed the undefined value, making it possible for a term to be non-evaluable (evaluate to ϵ).

A transition can only be fired when all its arc expressions evaluate to a defined value. The undefined value can be used to detect an illegal evaluation of terms, such as a division by zero, and show an error message or abort the reachability graph generation.

3.3.2 Short-Circuit Evaluation

Short-circuit evaluation of algebraic terms is a way to make algebraic system nets resemble more closely optimized computer implementations of expression evaluators. A signature of a *short-circuit algebra* contains, in addition to sort names and operation symbols, a pairwise disjoint family $\mathcal{G} = \bigcup_{s',s \in \mathcal{S}} \mathcal{G}_{s',s^n}$ of *short-circuit operation symbols*, which is disjoint from the set of operation symbols. Here s^n stands for $\underbrace{s, \ldots, s}_{\text{n times}}$.

A short-circuit operation symbol $g \in \mathcal{G}_{s',s^n}$ stands for an operation from sort s' to an operation from s^n to s, where s' is the *selection sort* and s is the *range sort* of g.

The selection sort must be ordered, in other words, the algebra must define a bijective mapping from the domain set of a selection sort to an ordered set of items, for example natural numbers. Now a short-circuit term $T = g(T', T_1, \ldots, T_n)$ can be evaluated by first evaluating the selection term T', finding it's ranking among the domain set if its sort, and evaluating the term T_k, where the index k matches the ranking. Note that the number of terms T_i must match the size of the domain set of the selection sort.

4 THE MARIA PROJECT

MARIA (Modular Reachability Analyzer) [20], [19] is a research project at the Laboratory for Theoretical Computer Science of Helsinki University of Technology. The aim of the project is to produce a pack of software tools that perform reachability analysis and check safety and liveness properties of distributed system models. The models can be constructed either by hand or automatically from some description language, such as SDL.

The MARIA analyzer uses a formalism based on Algebraic System Nets. A predecessor for the MARIA analyzer was `PROD` [33], developed at the Digital Systems Laboratory of the Helsinki University of Technology [1]. `PROD` is capable of exhaustively analyzing systems with millions of reachable states and performing model checking while generating the set of reachable states. It has been later extended with advanced algorithms, such as a model checker for branching time temporal logic and the stubborn set method.

The current MARIA analyzer was coded primarily by Marko Mäkelä in 1999-2003. Major improvement over `PROD` is the type system: while `PROD` allowed only tuples of integers to be used, the type system of the MARIA analyzer has been designed to meet the practical needs of specification and programming languages and contains also high-level data types like queues and stacks.

The MARIA analyzer supports the design of several different *task-specific front-ends* for various input languages like SDL, Petri nets, transition systems etc. The user of the analyzer does not need to be an expert in the specific formal method, but may use a familiar user interface.

A primitive front-end for standard SDL, supporting the basic elements of the language, has been coded by Marko Mäkelä, André Schulz and Teemu Tynjälä in 1998-2000 at the Laboratory for Theoretical Computer Science of Helsinki University of Technology.

4.1 THE MARIA ANALYZER

The input language of the MARIA analyzer is entirely textual, and the syntax of the expressions resembles the C programming language. Although Petri nets are often represented graphically as directed bipartite graphs, the advantage of the textual representation is not having to implement a graphical user interface, which would mean creating an optimal graphical layout for an automatically generated Petri net model. Graphical notations are also very difficult to use for large systems and must be equipped with facilities for dividing the graph into several (hierarchical) parts.

The MARIA analyzer is based on algebraic system nets, using a short-circuit error algebra defined in [19].

4.1.1 Data Types of the MARIA Analyzer

The basic data types of the MARIA analyzer are:

[1]now Laboratory for Theoretical Computer Science

- *Boolean*
- *Integer*
- *Character*
- *Enumerated*
- *Identifier*

The *Enumerated* type consists of named integer constants having distinct values. The *Identifier* type is similar to a pointer or resource handle notion of programming languages in the extent that it does not have any literals, and the values can only be compared for equality. Internally the *Identifier* type is represented as integer values. Symmetry reductions are very effective on models making use of the *Identifier* type. The MARIA analyzer does not have a data type that would behave exactly like pointers or references in programming languages, because it would break the locality principle of Petri nets.

The MARIA analyzer supports the following composite types:

- *Structure*
- *Union*
- *Array*
- *Queue*
- *Stack*

All the MARIA data types have a *limited domain*, which can be further restricted by specifying ranges of allowed values. This is very important in reachability analysis where the value ranges have to be restricted as much as possible to avoid state space explosion.

All the data types are *totally ordered*, including the composite types.

The *Array* type specifies a non-empty set of objects of a *member type*, indexed by an *index type*. Usually arrays in programming languages are integer-indexed but MARIA allows using any data type as the index type, because there is a total order on every type. The array size is determined by the number of constants in the index type.

The *Queue* and *Stack* data types are different cases of a *Buffer* type. The *Buffer* type resembles the *Array* type, but is not indexed. A buffer has always a maximum length.

4.1.2 Verifying Properties of Models

Rather simple error conditions of a system, such as deadlocks, can be detected directly in the model by reachability analysis. More complex properties must be described with, for example, formulae of temporal or modal logic. These properties are often divided into *safety* and *liveness* properties. A safety property ("nothing bad happens") is violated if a chain of events is found which leads from the initial state of the system to a “bad” state. A violation of a liveness property is an infinite loop of actions which never leads to a “good” state. Liveness properties may be refined with fairness assumptions i.e. excluding obviously impossible execution paths.

MARIA supports LTL logic with fairness constraints. An external tool is used for translating LTL formulae to generalized Büchi automata. Replacing the translator is possible due to the modularity of the MARIA analyzer.

4.1.3 Reachability Analysis

The generated reachability graphs tend to be very large, and storing them efficiently is not a trivial problem. The MARIA analyzer uses a technique where each state is stored separately. This makes it possible to navigate in the reachability graph and to perform all sorts of queries on it afterwards.

One way to reduce the space requirements of the reachability graph is to generate it only partially. Using some reduction method, like partial order reduction methods or symmetries, it is possible to reduce the size of the reachability graph considerably. Because of the modular structure of the analyzer it is possible to extend it by implementing different reduction methods.

Sometimes even the reduction methods are insufficient. When industrial systems are verified, it is desirable to allow for partial verification either using smart automatic selection of paths in the reachability graph or interactive selection by the user. Now the whole reachability graph does not have to be generated, only the interesting part of it.

4.2 SDL FRONT-END FOR THE MARIA ANALYZER

The SDL front-end for MARIA, SDL2PN, uses the ideas of the Emma [13], [14] project. Emma is a tool which translates TNSDL[2] [17] language to the net description language of `PROD`. Naturally, the data type system of MARIA makes translating data types and expressions much easier. However, SDL contains constructs not in TNSDL, and some of them have quite complex translations.

[2]TNSDL (TeleNokia SDL) is a dialect of SDL-88

5 TRANSLATION RULES FOR SDL

To translate an SDL specification into a Petri net is basically to generate a corresponding net transition for each SDL statement. Some statements may require more than one net transition. The translation is carried out in the following steps:

- static analysis of the SDL system
- translate types
- create places
- add initial marking to places
- generate net transitions for the environment
- generate net transitions corresponding to SDL states and transition triggers

Optimizing the model for reachability analysis may require a number of further translations, e.g. merging consecutive independent net transitions.

5.1 STATIC ANALYSIS OF AN SDL SYSTEM

The control flow of an SDL program is controlled using a *program counter*. It is a number identifying the SDL statement which is to be executed next. Before starting to translate the system, each SDL statement is given a unique number, which will be used as the program counter value.

Some structural analysis of the SDL system has to be done to find all the possible receivers for signals mentioned in output statements. During this step, a structure of signal paths between the processes in the system is resolved as described in Section 5.8. This step is quite complicated, because the TO and VIA constructs of OUTPUT statements are taken into account.

5.2 TRANSLATION OF TYPES

The translation of types consists of the following steps:

- create simple common types
- translate user-defined types
- create types for signals
- create complex common types

Creating a type means generating a type definition for it in MARIA input language.

The simple common types, like *pid_t* (for process identifier), *tid_t* (for timer identifier), *rec_t* (for recursion depth) and *pc_t* (for program counter), are created first because they are used as parts of other types. The reason for translating user-defined types before signal types is that signals may have parameters of user-defined types.

SDL specifies that a special expression **SENDER** should return the process identifier of the process instance that sent the last signal consumed from the input queue. To implement this behavior, the signals must carry the process identifier of the sender, which the receiver then saves. For each signal, a structure type is defined which contains the parameters of the signal. The example in Figure 5.1 shows two signals, one with parameters and another without, defined in MARIA input language. The actual *signal type signal_t*, which is used to represent all signals in the system, is a structure containing a *pid_t* and a union of all the structure types defined for different signals. This way, a value of signal type contains a process identifier and any signal defined in the system. The process identifier value 0 is reserved for signals coming from the environment of the system.

```
typedef struct {
  int param1;
  bool param2;
} signal1_t;

typedef struct {
} signal2_t;

typedef union {
  signal1_t signal1;
  signal2_t signal2;
} signal_union_t;

typedef struct {
  pid_t pid;
  signal_union_t signal;
} signal_t;
```

Figure 5.1: Signal type definition

Complex common types include a union type for all signals in the system (*signal_union_t*) and a type for process queues (*queue_t*).

Names for the created MARIA types are of form
<*type prefix*><*sort name*><*type suffix*>. This is to make names of types clearly distinguishable from other names in the net description. By default, *type prefix* is empty and *type suffix* is "*_t*".

5.3 GENERATING PLACES AND INITIAL MARKINGS

The places that are generated to the net can be divided into three categories: process-, procedure-, and system-related places.

5.3.1 Places for Processes

Two places are created for each process in the system: a *process control place* and a *process queue place*.

The purpose of a process control place is to keep track of the execution of the process. A process control place contains a token for each instance of the process. The tokens contain three values: process identifier (value of type *pid_t*), recursion depth (value of type *rec_t*) and program counter (value of type *pc_t*).

The name of a process control place is the name of the process, prefixed by names of enclosing blocks and the name of the system. This is to prevent name clashes when processes in different blocks have the same name. For example, the name of the process control place for process *P1* in block *B1* in system *Test* is *Test_B1_P1*, and the name of the process control place for process with the same name in block *B2* in the same system is *Test_B2_P1*.

The names created using this strategy are usually quite long, which may be annoying if the resulting reachability graph is shown graphically. If the names of entities (processes, blocks, procedures etc.) in an SDL specification are distinct, it is possible to use only the name of a process as the name of the process control place. Another way to reduce the length of the names is to use some compact labeling for names. The processes could be, for instance, numbered. This reduces the readability of the net, because the labels do not contain information about the entities which they represent. The problem can be overcome by keeping a separate *name translation table* which maps the labels to more intuitive names. SDL2PN allows the user to select whether prefixed names or simple names are used. It does not employ name translation tables.

A process queue place represents the input queue of a process. It contains a token for each instance of the process. The tokens contain the process identifier of the process instance (value of type *pid_t*) and a buffer of signals (variable of MARIA *Queue* type). Name for a process queue place is constructed the same way as the name for the process control place, but it has a suffix "Queue".

Some processes may have a restriction on the maximum number of instances that may exist at any given time. For these processes, a *process count place* is created. The process count place contains a single token, which holds one integer value. The value specifies the number of process instances that may be created. When creating a new process instance, the value is checked and if it equals to zero the creation fails.

In the rest of this Section, the process control place of a process named "X" is denoted as *Control X*, and the process queue place of the same process is denoted as *Queue X*. The process count place of the process, if it has one, will be denoted as *Count X*.

A separate *process variable place* is created for each variable of a process.

The tokens in a process variable place contain the process identifier of the owning process instance (value of type *pid_t*), the recursion depth (value of type *rec_t*) and the value of the variable (type depends on the sort of the variable).

In SDL, a process may also have *parameters*. The values of process parameters are bound when a process instance is created. For each process parameter, a *process parameter place* is created. The structure of tokens contained in a process parameter place is similar to the case of process variables. The initial marking contains no tokens in process parameter places, because no process parameters can be bound at system initialization time.

5.3.2 Places for Procedures

A *procedure control place* is generated for each procedure in the system. The tokens in a procedure control place contain the same components as the tokens in process control places, and additionally a *wait number* (value of type *pc_t*), which is used to separate different procedure calls from each other.

For each procedure variable and parameter, a *procedure variable place* or *procedure parameter place* is created, respectively. The tokens in these places contain the process identifier of the calling process instance (value of type *pid_t*), the recursion depth (value of type *rec_t*), the value of the variable or parameter (type depends on the sort of the variable/parameter) and the wait number as in a procedure control place.

The initial marking does not contain tokens in any procedure-related place, because there are no procedure calls active at the system initialization time.

5.3.3 System Places

Several places are created for the use of all processes in the system. The *PId pool place*, *free PId place* and *PId expression place* are places controlling the use of process identifiers. The *timer control place*, *timer lock place* and *TId map place* are used for handling timers. The *resource place* is for controlling the granularity of actions as described in Section 6.1.

The *PId pool* place works as a pool of process identifiers to be given for dynamically created process instances. The tokens in the *PId pool* place contain a single value of type *pid_t*. The initial marking has tokens in *PId pool* place for all process identifiers not assigned to any process instance. When a process instance is created, a process identifier is removed from the *PId pool* place. Destroying a process instance returns its process identifier to the *PId pool* place.

To implement correct behavior of the SDL CREATE statement, the dynamically changing number of available process identifiers must be stored. For this reason, there is the *free PId* place, which always contains a single token. The token in the *free PId* place contains an integer value specifying the number of tokens in the *PId pool* place.

The purpose of *PId expression* place is to implement the special SDL expressions SENDER, PARENT and OFFSPRING. The initial marking has a token in *PId expression* place for each process instance in the system. The tokens consist of four *pid_t* values: one for each special expression and one for the process identifier of the process instance itself. The tokens are updated when signals are consumed from the input queue or new process instances are created.

The *timer control* place contains a token for each timer in the system. In SDL, timers can have parameters and the parameter values are used for identifying the timer. A timer which is activated two times with different parameter values is considered to be two different timers. Because of this, a *timer identifier* (**TId**) is assigned to all possible timer name – parameter value combinations. This can be done, because all data types in MARIA language have a limited domain. The domain may be quite large, however, and if there are timers with parameters which have vary large value range, a lot of unnecessary tokens will be generated to the net. The tokens in *timer control* place contain a timer identifier (of type *tid_t*), a process identifier (of type *pid_t*) identifying the process instance which owns the timer, and a flag (of MARIA *Boolean* type) which indicates whether the timer is active.

The timer identifiers must somehow be mapped to the parameter values that the signal, which the timer sends when it expires, should carry. This is possible using the *TId map* place, which contains a token for each timer identifier value in the system. In addition to the timer identifier, the tokens contain a value of type *signal_t*, which contains the parameter values for the timer.

To keep the size of the reachability graph in reasonable limits, the timers can not be allowed to expire at any time. Instead, special commands (described in Section 5.7) are used to specify an *expiration window*, the SDL statements between which the timer can expire. To prevent a timer from expiring outside its expiration window, the *timer lock* place is introduced. It contains a token for each timer identifier in the system. Each token contains also a flag (MARIA *Boolean* type), which indicates whether it is allowed for the timer to expire.

To further reduce the size of the reachability graph, it is possible to treat sequences of SDL statements as atomic actions (see Section 6.1). To implement this, a *resource* place is created. It contains a single token, containing a MARIA *Boolean* value.

Table 5.1 summarizes the system-related places and types of tokens they contain.

Place name	**token type**
PId pool	<*pid_t*>
free PId	<*Integer*>
PId expression	<*pid_t, pid_t, pid_t, pid_t*>
timer control	<*pid_t, tid_t, Boolean*>
TId map	<*tid_t, signal_t*>
timer lock	<*tid_t, Boolean*>
resource	<*Boolean*>

Table 5.1: System-related places

5.4 THE ENVIRONMENT

The behavior of the environment of the system is modeled by generating an *environment transition* for each signal – parameter – receiver combination, where the receiver is a process to which there exists a communication path from the environment of the system capable of carrying the specific signal. An environment transition adds the signal to the queue of the receiving process. Figure 5.2 shows the structure of an environment transition. The token taken from the process queue place consists of two *variables*, *pid* and *buffer*. The *buffer* variable is of MARIA *Queue* type, which has a "+" operator for adding elements at the end of the queue. It is used to add a new signal to *buffer*. The signal name and parameters are fixed when the transition is generated (a separate transition will be generated for all signals and parameter values). The name for the transition consists of the word "Env", the name of the receiving process, the name of the signal and a running number. The purpose of the number is to separate similar transitions with differing parameter values of the signal.

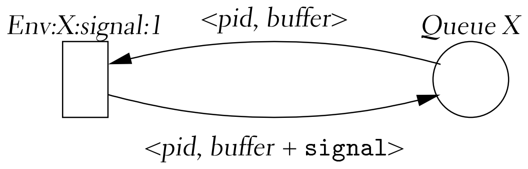


Figure 5.2: An environment transition

In SDL, there is a difference between the environment of the system, and the environment of a block. The environment of a block is the enclosing block, or in the case of a block defined in the system level, the SDL system itself. The environment of the system is represents the world outside the system, and it is not defined in the specification.

In SDL, each endpoint of a signal route which is not attached to some process leads to or from the environment of the block in which the signal route is defined. On the border of the block, the signal route may be connected to a channel. The channel itself may lead either to some other block or to the environment of the block or system where the channel is defined. An environment transition is generated for only such communication paths which start from the environment of the system. Figure 5.3 shows an SDL system consisting of two blocks, *BA* and *BB*. Figure 5.4 shows the channels, signal routes and connections defined in the system. Although the definitions of signal routes *r1* and *r3* look identical, an environment transition will only be generated for process *PA*, because it is connected to the environment of the system (through channel *c1*).

If there is no need to model the behavior the system with all the possible alternative input sequences, the system can be completely detached from the environment, and the environment be simulated *within* the system. There may, for example, be one process which generates all the input sequences of interest, and then stops execution.

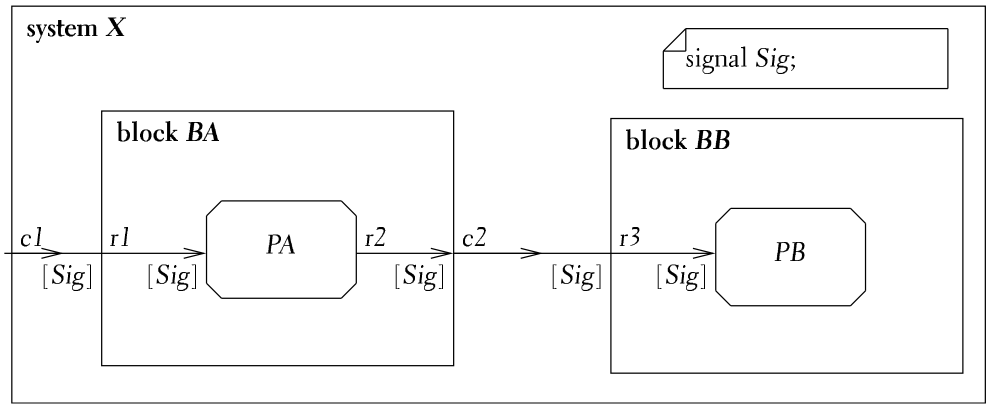


Figure 5.3: An example of an SDL system with a signal path from the environment

```
system X;
  channel c1 from env to PA with Sig;
  endchannel c1;

  channel c2 from PA to PB with Sig;
  endchannel c2;

  block BA;
    signalroute r1 from env to PA with Sig;
    signalroute r2 from PA to env with Sig;

    connect c1 and r1;
    connect c2 and r2;
  endblock BA;

  block BB;
    signalroute r3 from env to PB with Sig;

    connect c2 and r3;
  endblock BB;
endsystem X;
```

Figure 5.4: The channel, signal route and connection definitions for the system in Figure 5.3

5.5 TRANSLATING TRANSITION TRIGGERS

For each state of an SDL process (except initial state), a set of *trigger conditions* can be specified. The set of trigger conditions for a state may consist of a number of **INPUT** constructs, **SAVE** constructs, spontaneous transitions and continuous signals. Additionally, one **ASTERISK INPUT** or **ASTERISK SAVE** may be present. In the following, translations for **INPUT** construct and spontaneous transition are presented.

Net transitions modeling transition triggers combine the last statement of the previous SDL transition and the (nondeterministic) selection of the next SDL transition. For this reason, a set of net transitions is generated to model one transition terminator: one net transition for each combination of previous SDL transition and next SDL transition.

Figure 5.5 shows a simple SDL process named *Device* consisting of two states, *Idle* and *Active*. Both states have two transition triggers, **INPUT** constructs with signals *Activate* and *Deactivate*. There are three **NEXTSTATE** statements leading to the *Idle* state and two leading to the *Active* state. The statements are numbered for clarity.

Table 5.2 lists all net transitions that would be generated to model the transition triggers in the *Device* process. Two things are needed to separate net transitions from each other: the signal consumed in the transition and program counter of the previous statement.

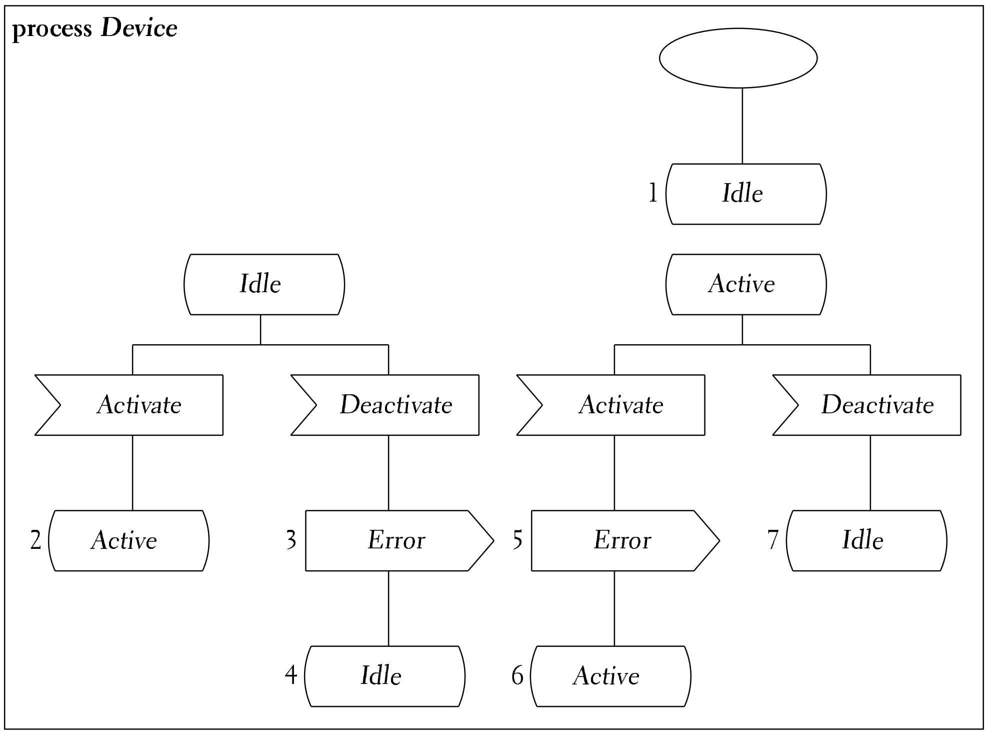


Figure 5.5: An SDL process

State	**Signal**	**Program Counter**	
		Before	**After**
Idle	*Activate*	1	2
Idle	*Activate*	4	2
Idle	*Activate*	7	2
Idle	*Deactivate*	1	3
Idle	*Deactivate*	4	3
Idle	*Deactivate*	7	3
Active	*Activate*	2	5
Active	*Activate*	6	5
Active	*Deactivate*	2	7
Active	*Deactivate*	6	7

Table 5.2: Net transitions needed for transition triggers of the SDL process in Figure 5.5

5.5.1 INPUT Construct

Figure 5.6 shows the structure of an INPUT net transition. Some values are fixed at net generation time. Those values are listed in Table 5.3. The first three values are present in all INPUT net transitions. They refer to *Program Counter Before*, *Program Counter After* and *Signal* columns of Table 5.2, respectively.

Other names shown in tokens on input arcs are *variables*. Variables in this sense should not be mixed with SDL process variables that are represented as places in the net. Variables used in Figure 5.6 are briefly explained in Table 5.4. More thorough explanation of an INPUT net transition and the variables used follows.

The *buffer* variable models the process input queue. It is of MARIA *Queue* type containing elements of type *signal_t* (see Figure 5.1). MARIA *Queue* type has a "/" -operator, which returns the number of elements in the queue that it is applied to. It also has a "*" -operator for viewing the element at the beginning of the queue. These operators are used on the *buffer* variable in the gate expression to check that there is at least one element in the queue, and it is a signal of the type mentioned in the INPUT part that the net transition is generated for (`sig`).

The MARIA *Queue* "*" -operator does not remove elements from the queue. The first element in the queue is removed with "-" -operator before the *buffer* variable is returned to the *Queue X* place.

An INPUT net transition must also update the *PId expression* place to contain the process identifier of the SDL process that sent the signal. This is achieved by replacing the old value of *sender* variable with the `pid` component of the *buffer* element.

An INPUT construct may also contain SDL process variables, if the signal specified in it has parameters. In this case, the old values of the variables are replaced by the parameter values carried by the signal. In the example in Figure 5.6, the value of variable *Y* is replaced by value of `param0` of the

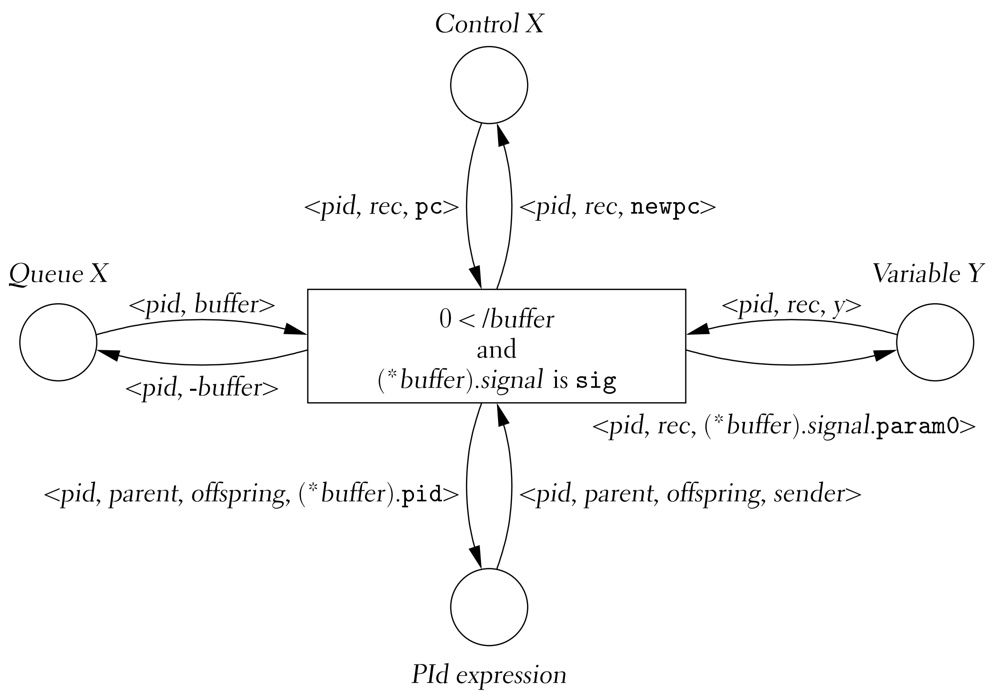


Figure 5.6: The structure of an **INPUT** net transition

Constant	**Description**
`pc`	Program counter before the net transition is fired - number of the previous SDL statement
`newpc`	Program counter after the net transition is fired - number of the first SDL statement in the SDL transition associated with the **INPUT** construct.
`sig`	Name of the signal consumed by the **INPUT** construct.
`param0`	Name of a signal parameter. Arcs are created for all parameters of the signal in question.

Table 5.3: Description of constants used in Figure 5.6

Variable	**Meaning**	**Use in transition**
pid	Process identifier	Ties input tokens together (all tokens contain the same *pid*)
rec	Recursion depth	Not used
buffer	Input queue	First element is removed
parent	PId expression	Not used
offspring	PId expression	Not used
sender	PId expression	Updated
y	Process variable	Updated

Table 5.4: Description of variables used in Figure 5.6

signal.

If a signal has multiple parameters, arcs to and from the variable places are generated for each SDL process variable mentioned in the **INPUT** construct. As a special case, same variable can occur multiple times in an **INPUT** construct. In this case, only one pair of arcs is generated to/from the variable place. The old value of the variable will be replaced with the value of the signal parameter matching the last occurrence of the variable in the **INPUT** construct.

According to SDL specification [34], it should be possible to omit some signal parameters from an **INPUT** construct. This kind of behavior has not been implemented in the SDL2PN front-end. If the signal has parameters, also the **INPUT** construct has to define corresponding variables to receive their values.

If an **INPUT** construct of a state contains multiple signals, a separate net transition is generated for each of them. All the transitions update the program counter to the same value.

5.5.2 ASTERISK INPUT Construct

ASTERISK INPUT construct is a shorthand notation for an **INPUT** construct, which accepts any signal not defined in the other **INPUT** or **SAVE** constructs of the state.

An **ASTERISK INPUT** net transition is similar to a normal **INPUT** net transition, but the gate expression excludes all the signals specified in other **INPUT** or **SAVE** constructs of the state. To find the signals to exclude in the gate expression, all the **INPUT** and **SAVE** constructs of the state must be checked. At the same time it is easy to check that there is at most one **ASTERISK INPUT** or **ASTERISK SAVE** per state, and no signal has been defined twice in **INPUT** or **SAVE** constructs of the state.

Figure 5.7 shows a gate expression excluding two signals, *sig1* and *sig2* in MARIA language.

```
( 0 < /buffer ),
!( ( *buffer ).signal is sig1 ||
   ( *buffer ).signal is sig2
 )
```

Figure 5.7: A gate expression of an **ASTERISK INPUT** net transition

5.5.3 SAVE Construct

SAVE construct allows saving signals for later use. A signal mentioned in a **SAVE** construct of an SDL state is not removed from the input queue, but it is retained in the queue and available for input in a consecutive state.

Lets take as an example an SDL state *S* which has input constructs for signals *b* and *c*, and a save construct for signal *d*. Assume that when the process enters the state *S*, there are signals <*d*, *b*, *c*> in the input queue, where *d* is the oldest (first) signal in the queue. Since *d* is saved in the state *S*, the process consumes the signal *b* and executes the SDL transition following input *b* and moves to the next SDL state. In the next state the signal *d* is first signal in the input queue.

This example suggests, that a straightforward translation of the **SAVE** construct would be to swap first two signals in the input queue, and this is how SDL2PN currently works. There is, however, a fundamental flaw in this approach which deadlocks the system if the first two signals in the input queue are both mentioned in **SAVE** constructs of the state. A better solution would be to utilize an auxiliary queue for saved signals.

5.5.4 Spontaneous Transition

Spontaneous transition does not expect any signals in the input queue. A spontaneous transition may be triggered any time the process is in the SDL state where the spontaneous transition is declared. After a spontaneous transition has been interpreted, the **SENDER** expression returns the process identifier of the process itself.

Figure 5.8 shows the structure of a net transition modeling SDL spontaneous transition.

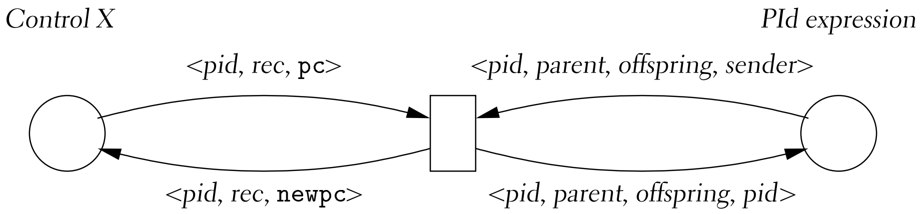

Control X
PId expression
<pid, rec, pc>
<pid, parent, offspring, sender>
<pid, rec, newpc>
<pid, parent, offspring, pid>

Figure 5.8: The structure of net transition generated to model SDL spontaneous transition

5.5.5 Enabling Condition

An *enabling condition* can be associated with an **INPUT** construct or spontaneous transition. An enabling condition is a boolean expression which is evaluated before the signal specified by the **INPUT** construct is consumed from the input queue. If it evaluates to false, the signal is saved instead. In the case of spontaneous transition the spontaneous transition is not triggered if the enabling condition evaluates to false.

A spontaneous transition with an enabling condition is simple to translate by adding the condition as a gate expression to the net transition. An **INPUT** or **ASTERISK INPUT** construct with an enabling condition needs two net transitions to be generated: one for the case when the condition is true and another for the case when it is false. The first net transition is a normal **INPUT** net transition where the gate expression has been augmented with the condition expression. The other net transition is a **SAVE** net transition, the gate expression augmented with the negation of the condition.

If a variable is used both in the enabling condition of an input, and as a parameter for the same input, the value of the variable when the enabling condition is evaluated is the value *before* the assignment of signal parameters.

Procedure calls are not allowed in enabling condition expressions, because they can not be evaluated in a single net transition.

5.5.6 Translation for Implicit Signal Consumption

Implicit signal consumption in SDL means discarding signals which are not mentioned in any of the INPUT or SAVE constructs of the state the process is in, and the state contains neither ASTERISK INPUT nor ASTERISK SAVE construct.

An additional net transition is needed for each SDL state to implement the discarding of a signal. A state which contains an ASTERISK INPUT or ASTERISK SAVE does not need an implicit signal consumption net transition because it already handles all the possible signals the input queue may contain.

An implicit signal consumption net transition does not change the program counter value, it just removes the first signal from the input queue.

An implicit signal consumption net transition should not remove signals that are mentioned in an INPUT or SAVE construct of the SDL state. For this reason, a gate expression is needed which contains a negation of all the signals that are mentioned in any INPUT or SAVE construct of the state. Resolving these signals can be done at the same time as signal sets for ASTERISK INPUT and ASTERISK SAVE net transitions are collected. Actually, the signal sets to be excluded in ASTERISK INPUT, ASTERISK SAVE and implicit signal consumption net transition are the same.

If a state does not contain an ASTERISK INPUT or ASTERISK SAVE, it has an implicit signal consumption net transition and vice versa.

5.6 TRANSLATING THE SDL STATEMENTS

As a general rule, one net transition is required to model one SDL statement. There are some exceptions, however, for example multi-signal OUTPUT statement requires one net transition for each signal. The following sections describe translations for DECISION, CREATE, STOP, OUTPUT and TASK SDL statements.

5.6.1 DECISION Statement

DECISION statement is used to implement conditional branching in SDL. One net transition is generated to model the selection of the SDL transition to execute. The net transition sets the program counter value using a MARIA if-then-else expression.

Figure 5.10 shows an example of a net transition generated to model the DECISION statement shown in Figure 5.9. In the example, the SDL transition to be executed is selected based on the value of variable V.

The SDL transitions in the DECISION branches are translated as normal SDL transitions. The only difference is that an SDL transition which is in a DECISION branch does not have to end in a transition terminator. In this case, the control flows to the SDL statement following the DECISION state-

```
decision (V);
   case (1):
      /* Do something */
   else:
      /* Do something else */
enddecision;
```

Figure 5.9: An SDL DECISION statement

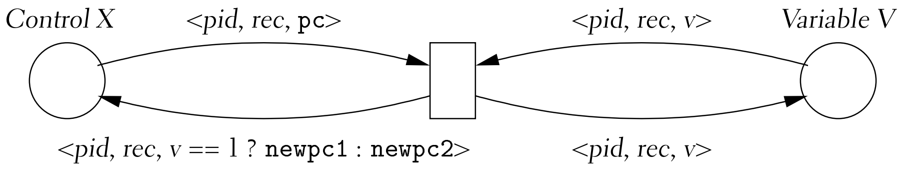


Figure 5.10: A DECISION net transition

ment. In the generated model the value of the program counter is changed to the statement number of the first statement after the DECISION statement.

ANY DECISION Statement

An **ANY** DECISION statement contains a set of branches, one of which is selected nondeterministically when the statement is interpreted. A separate net transition is generated for each branch. Each net transition replaces the program counter value with the statement number of the first statement of the branch.

5.6.2 CREATE Statement

CREATE statement is used to create a process instance during runtime. The main function of CREATE net transition is to update the program counter of the calling process and add new tokens to the process control, queue and variable places of the process to be created. A new **PId** is fetched from the *PId pool* place.

In SDL processes may have constraints restricting the number of instances of the process that may exist simultaneously. If a process tries to create a new instance of a process whose instance count is full, the process creation should fail and the creating processes OFFSPRING value should equal `NULL` after the CREATE statement has been interpreted. For this reason, a process count place is generated for each process which has a restricted maximum number of instances. The CREATE net transition checks that the allowed instance count is more than zero before creating the new instance, and decrements the allowed instance count at the same time.

According to [34], the number of **PId** values is unlimited, there are always new identifiers to assign to newly created processes. In real life implementation this is not possible, but the *pid_t* type has a maximum value. If there are no more identifier values to use when executing a CREATE statement, similar behavior is assumed than in the case in which a maximum number of process instances is reached. The *free PId* place contains the number of **PId** values available. The CREATE net transition uses this value in the similar way as the allowed process instance count.

Figure 5.11 shows the two net transitions corresponding to an SDL CREATE statement. The gate expressions of the two transitions are negations of each other, so exactly one of the net transitions is fired. Both CREATE net transitions update the program counter of the calling process, but only the succeeding CREATE transition adds new tokens to the net and changes the values of available **PId** values and allowed number of process instances.

An SDL process may have parameters. Parameter values are bound when the CREATE statement is interpreted. The process parameters are modeled as normal process variables and places are generated to the net for them. In the CREATE net transition new tokens containing the actual parameter values are added to the parameter places.

In process type definitions, the CREATE statement may contain a keyword THIS. It is interpreted to mean the process executing the CREATE statement, derived from the process type. The translation is similar to a normal CREATE statement.

5.6.3 Stop Statement

The STOP net transition is the opposite of the CREATE net transition. It removes the tokens from the process control, queue and variable places of the process, removes the pid expressions token of the process, returns the process identifier value to the *PId pool* place and updates the number of free pids and possibly the number of allowed instances in the process.

It is not allowed to execute a STOP statement in a procedure.

5.6.4 Output Statement

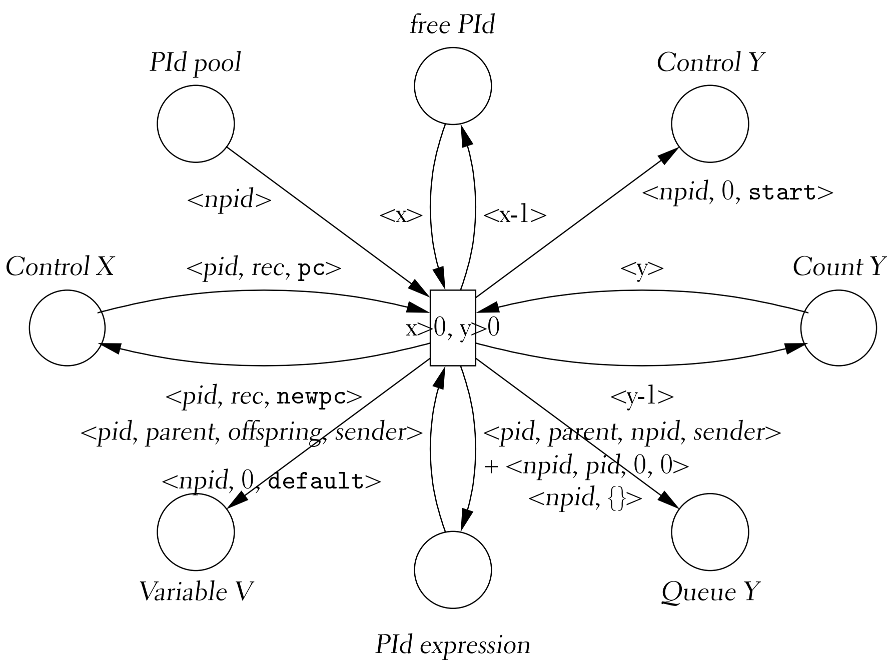
SDL processes send signals to each other with OUTPUT statement. The OUTPUT statement specifies the signal to be sent and values of parameters for the signal. Figure 5.12 shows the structure of a net transition generated for SDL statement OUTPUT *Sig*. The transition updates the program counter of the calling process to new value, and adds the signal to the queue of the receiving process.

Unlike other net transitions, in OUTPUT transition two different **PId**s are considered: the **PId** of the sending process and the **PId** of the receiving process. The **PId** of the receiving process is represented by *rpid* net variable.

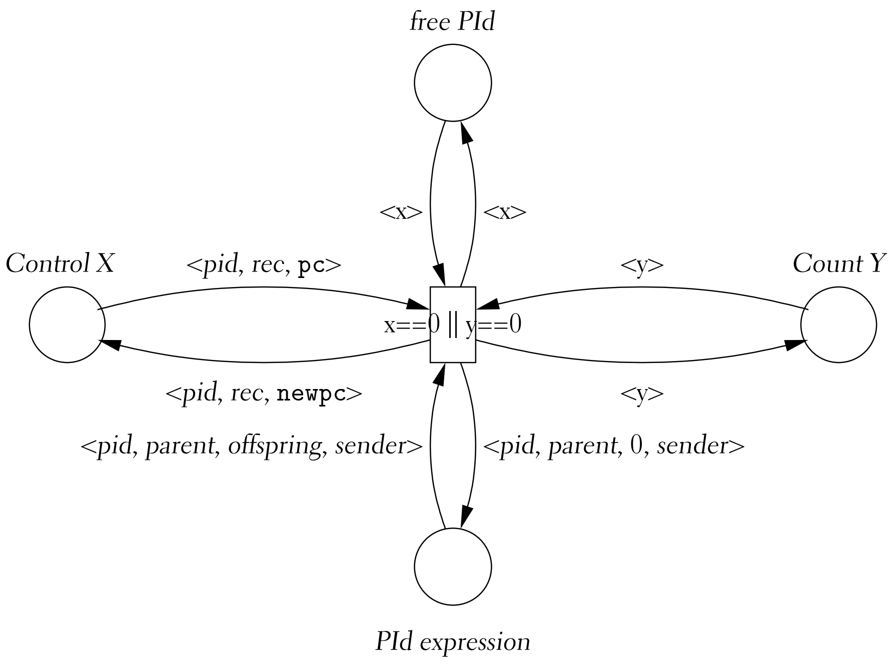
The receiver process input queue after adding the new signal is shown in the figure as `newbuffer` to make it easier to read. In reality, the value returned to *Queue B* place is constructed by adding to *buffer* variable a structure value of type *signal_t*. The structure value consists of the value of *pid* variable and a union value of type *signal_union_t*. The union value contains a structure value of type *Sig_t*. In Maria input language,

```
buffer + is signal_t { pid, Sig_t = {} }.
```

In case there are multiple possible receiver processes for a signal, a separate OUTPUT net transition is generated for each of them. Only difference between transitions is the receiver process place. For example, consider



(a) Succeeding CREATE net transition



(b) Failing CREATE net transition

Figure 5.11: CREATE net transitions

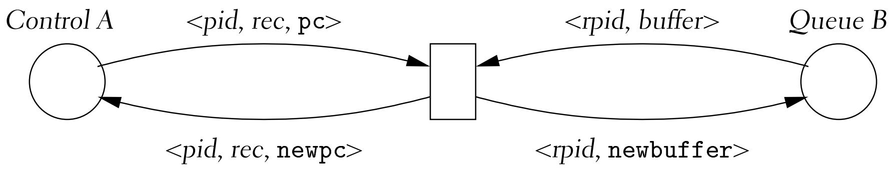


Figure 5.12: The structure of an OUTPUT net transition

the SDL system in Figure 2.6. Three OUTPUT net transitions are needed to model OUTPUT *Sig* statement in process *A*. A separate net transition is needed for processes *B*, *C* and *D* because each of them is reachable from process *A*.

By creating a separate net transition for each possible receiver of the signal, the selection of the actual receiver process is effectively made nondeterministic. If the receiver process has more than one process instance, the selection of the receiving process instance is naturally nondeterministic as well because the receiver **PId** is not fixed at net generation time.

An OUTPUT statement may contain multiple signals. In this case, a separate OUTPUT net transition is generated for each signal. For this reason, a continuous block of statement numbers (to be used as program counter values) is reserved for each multi-signal output statement, one statement number for each signal. Now the signals are sent one after another, preserving their order.

TO Constraint

The TO constraint in an OUTPUT statement restricts the receiver of the signal to the process instance with **PId** value specified in the OUTPUT statement. The translation is the basic OUTPUT net transition, shown in Figure 5.12, augmented with a gate expression which allows the signal to be put only in the input queue of a process with the **PId** value specified by the TO construct.

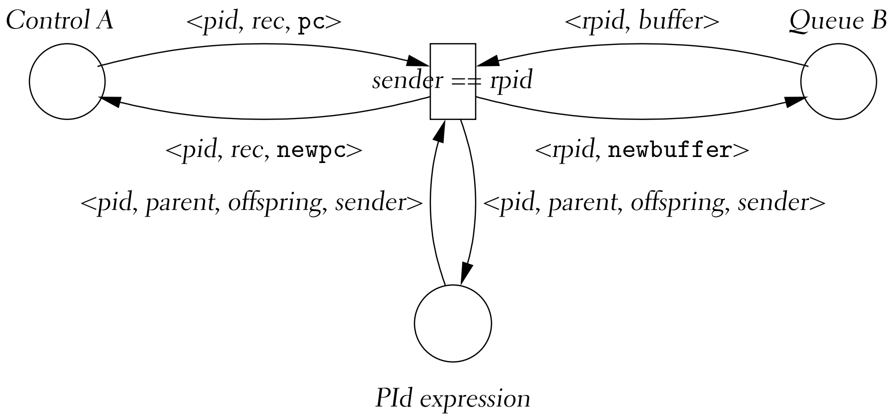


Figure 5.13: A net transition modeling an OUTPUT statement with a TO construct

Figure 5.13 shows a net transition created to model OUTPUT *Sig* TO SENDER. As in the case of basic OUTPUT statement without a TO constraint, a similar net transition is generated for each possible receiver process.

The new contents of the receiver process input queue are as in the case of a basic OUTPUT net transition.

The *PId expression* place is connected to the net transition because the gate expression uses SENDER expression. The value of the SENDER expression is relative to process *A*, because process *A* contains the OUTPUT statement. This is why the token taken from *PId expression* place has *pid* as the first element instead of *rpid*.

VIA Constraint

VIA constraint can be used to restrict the set of channels, signal routes and gates the signal may travel through. *Signal path elements* (channels, signal routes and gates) are static and known at net generation time. An OUTPUT statement with a VIA constraint is translated as a basic OUTPUT statement, but a net transition is created for only such SDL processes that are reachable through a signal route element mentioned in the VIA path. Information required to find such processes is gathered during the static analysis phase.

VIA ALL Constraint

VIA ALL constraint implements multicast through all signal path elements mentioned in the VIA ALL path. This means a separate net transition has to be generated for each distinct set of processes reachable through the signal path elements.

Figure 5.14 shows the net transition created to model OUTPUT *Sig* VIA ALL *c1*, *c3* statement in process *A* in the system shown in Figure 2.6

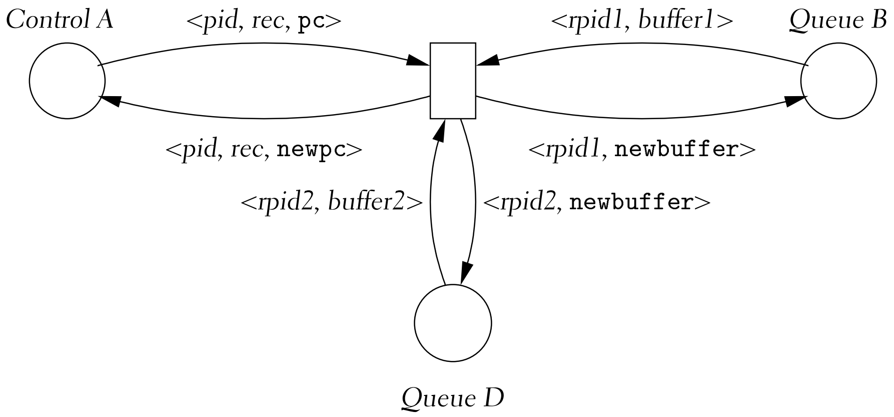


Figure 5.14: A net transition modeling an OUTPUT statement with a VIA ALL constraint

A special case when multiple separate VIA ALL -paths lead to the same process has not been implemented in the SDL2PN front-end.

Signal Parameters

If the signal contains parameters, the values of the parameters are evaluated first and after that the new signal is constructed and put to the queue of the receiver. Figure 5.15 shows an example of a net transition created to model OUTPUT *Sig*(*V*) statement. *V* is a variable of the enclosing process. Value of *V* is assigned to the signal parameter when the signal is sent.

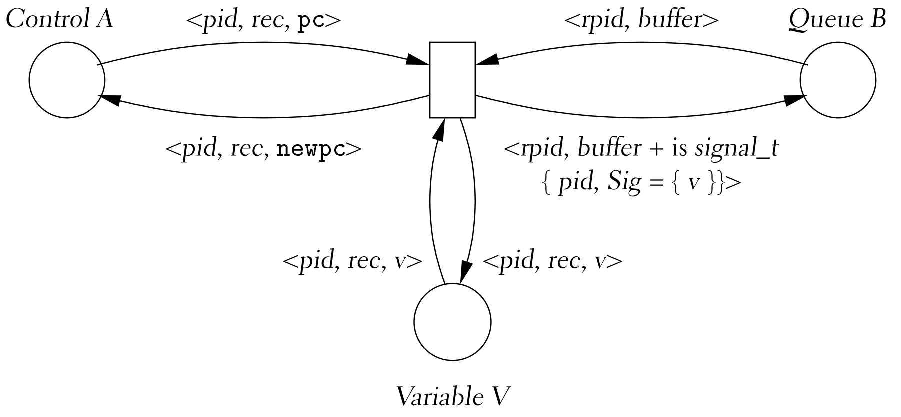


Figure 5.15: A net transition modeling an **OUTPUT** statement with signal parameters

5.6.5 TASK Statement

The **TASK** statement is used for two purposes in SDL: it may contain *informal text* or a *variable assignment*. Translating **TASK** statements containing informal text is not implemented in the SDL2PN front-end. The rest of this section concerns **TASK** statements which contain a variable assignment.

The structure of a **TASK** net transition is basically as shown in Figure 5.16. The transition updates the program counter to the number of the following statement and changes the contents of the token representing the variable on the left-hand side of the assignment.

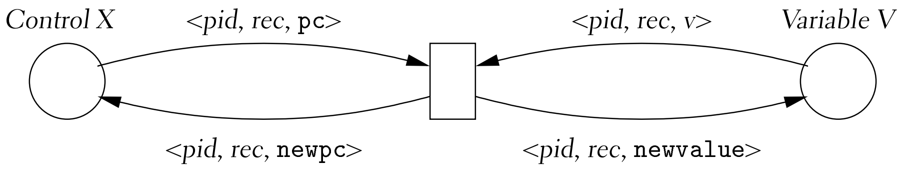


Figure 5.16: The structure of a **TASK** net transition

If the expression in the right-hand side of the assignment operator contains variables, the net places of these variables must also be connected to the generated net transition. The values of these variables are not changed by the assignment, so all tokens taken from these places are put back unchanged. Variable *W* in Figure 5.17 is an example of a variable used in the right-hand side of an assignment statement.

Even if the expression in the right-hand side of an assignment statement references the same variable twice, there may still be only one arc to and from the variable place. While generating the net, an arc is added from a net transition to a variable place only if no arc from the same net transition to the same variable place already exist. The same applies to incoming arcs. It is essential to create the assigning arc (the arc with the new value of the variable) first, because otherwise it will not be created at all if the same variable has also been used on the right-hand side of the expression.

In SDL, the left-hand side of an assignment may be a simple variable or a component of a structure or array variable. Figure 5.17 shows the net transition generated from SDL statement **TASK** *V*(*W*) := *W*; where *V* is an array of integer values. Note that although variable *W* appears on both sides of the assignment, only one pair of arcs is generated from/to its net place.

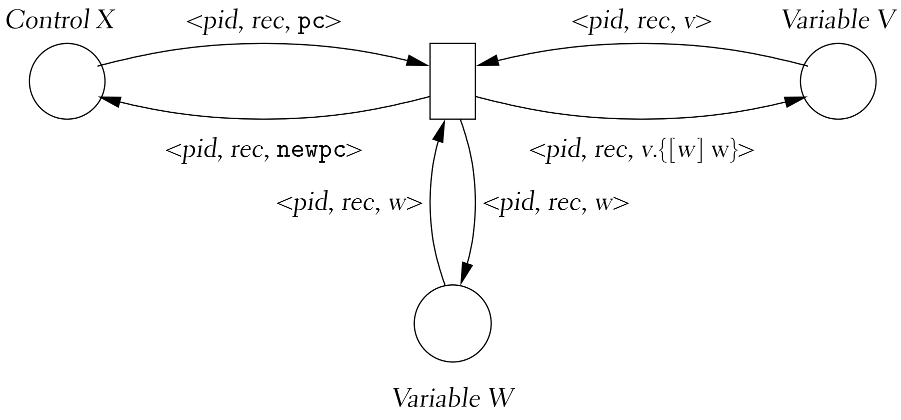


Figure 5.17: Assignment to an array element

Canonizing Statements

The expression on the right hand side of an assignment can contain procedure calls. Because each procedure call requires multiple net transitions, the procedure call has to be executed before the assignment. If the assignment contains many, possibly nested, procedure calls, the translation becomes overwhelmingly complex.

The solution used is to transform the procedure calls in an assignment statement to assignments to temporary variables, preceding the original assignment. The original assignment just uses the values of the temporary variables. This process, *canonizing* the statements, is done before starting to translate them. After canonization, procedure calls appear only as single statements, or as the right side of an assignment. Figure 5.18 shows an assignment statement containing procedure calls before canonization and in canonized form. The canonized form consists of three assignment statements, one for each procedure call in the original expression.

```
V := call C ( call A, 5 + call B ( 4 ) ) );

T1 := call B ( 4 );
T2 := call A;
V := call C ( T2, 5 + T1 );
```

Figure 5.18: A **TASK** statement before and after canonization

5.7 TIMERS

Three net places are used to control all timers in the system: *timer control* place, *timer lock* place and *TId map* place. These are explained in Section 5.3.3. All the three places contain a token for each timer in the system. No tokens are added or removed to or from these places in any net transition. The tokens in *timer control* place contain flags controlling whether a timer is active or inactive. The tokens in *timer lock* place contain flags controlling whether a timer may expire. The tokens in *TId map* place contain the signals that timers send when they expire.

There are two SDL statements for using timers: SET and RESET. In addition to these, two new statements are introduced to control the expiration window. These statements are called EXPIREPOINT START and EXPIREPOINT STOP.

5.7.1 SET Statement

Figure 5.19 shows the structure of a SET statement. Besides updating the program counter of the process, it changes the flag in the *timer control* place to `true`, indicating that the timer is active. In the SET statement, timer name and parameters are given, but the corresponding **TId** value must somehow be found to get correct token from the *timer control* place. This is done constructing the timer signal from the name and parameter values, and fetching the corresponding **TId** value from *TId map* place based on the signal.

The timer identifier is not actually necessary, because the timer signal contains all the information needed to identify the timer. If the **TId** were removed, some net transitions would be simpler. However, if the model contains timers with space-consuming parameters, removal of **TId** is not desirable because the timer signals will then appear in tokens in both *timer control* place and *timer lock* place.

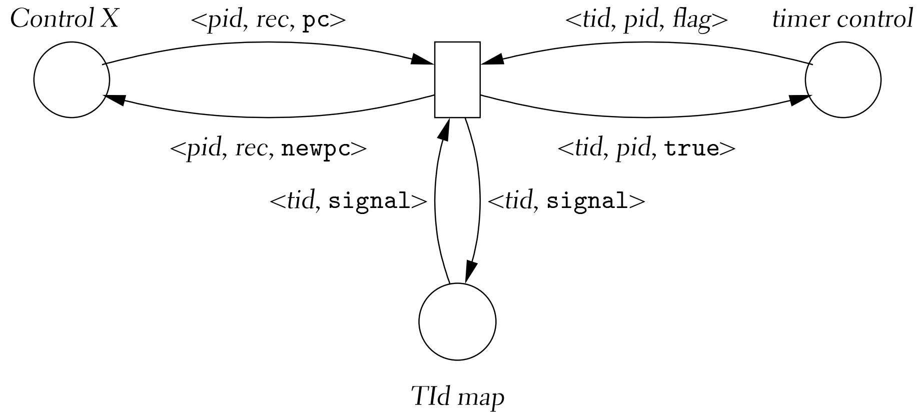


Figure 5.19: SET timer net transition

5.7.2 RESET Statement

Resetting a timer is quite complex, because if the timer has already expired and the timer signal is still in the input queue of the process, it must be removed from the queue. Luckily, MARIA queue type allows removing items from the middle of the queue in one step. Still, a separate net transition has to be generated for each possible position of the timer signal in the queue.

Figure 5.20 shows a **RESET** net transition. Similar net transition is created for all values of `y` less than maximum queue length. If the $\mathtt{y}^{th}$ signal in the queue is the one to remove, it is removed when the transition is fired. At the same time, the flag controlling timer activity in the *timer control* place is changed to `false`.

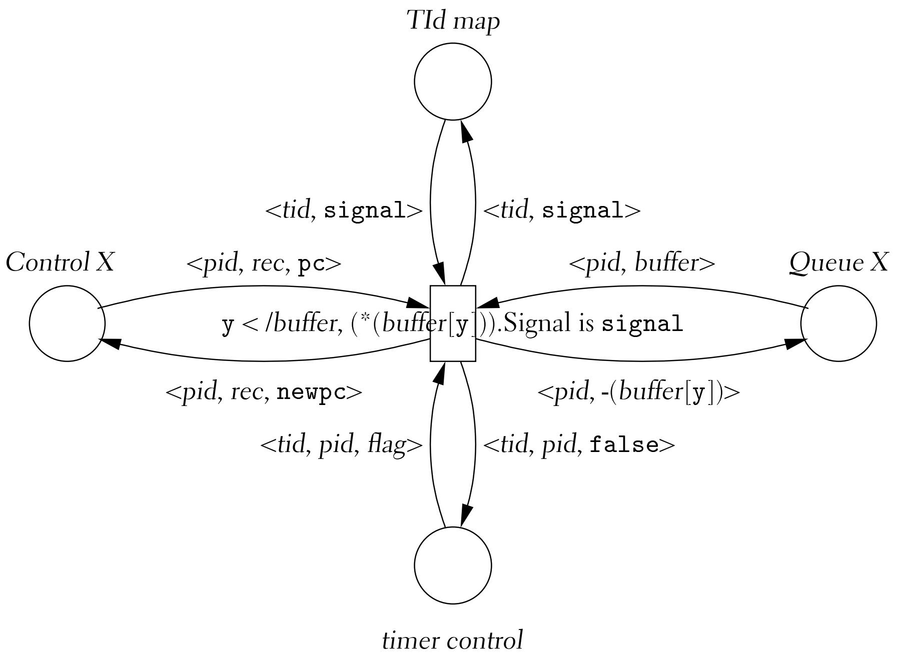


Figure 5.20: **RESET** timer net transition

An additional net transition is added for the case when the timer has not yet expired and the signal is not in the queue. The gate expression of the transition excludes all cases handled by the other **RESET** net transitions. The gate expression is thus a disjunction of expressions of form

```
( y == /buffer, !( *( buffer[x] ).Signal is signal )),
```

in MARIA language where $0 < \mathtt{y} <$ maximum length of process input queue and $0 < \mathtt{x} < \mathtt{y}$. This transition does not affect the queue, it just resets the timer activity flag.

5.7.3 Controlling the Expiration Window

To open and close the expiration window of a timer, it is only required to change the value of the flag in the *timer lock* place. Both **EXPIREPOINT**

START and EXPIREPOINT STOP statements are translated to one net transition, respectively. Each net transition updates the program counter of the calling process and changes the value of the flag in the *timer lock* place. EXPIREPOINT START net transition changes the value of the flag to `true`, and EXPIREPOINT STOP changes it to `false`.

An EXPIREPOINT START net transition is shown in Figure 5.21. The EXPIREPOINT STOP net transition is similar, but it contains additional arcs to and from the *timer control* place to check that the timer is inactive, to force the timer to expire (or be reseted) before the expiration window is closed.

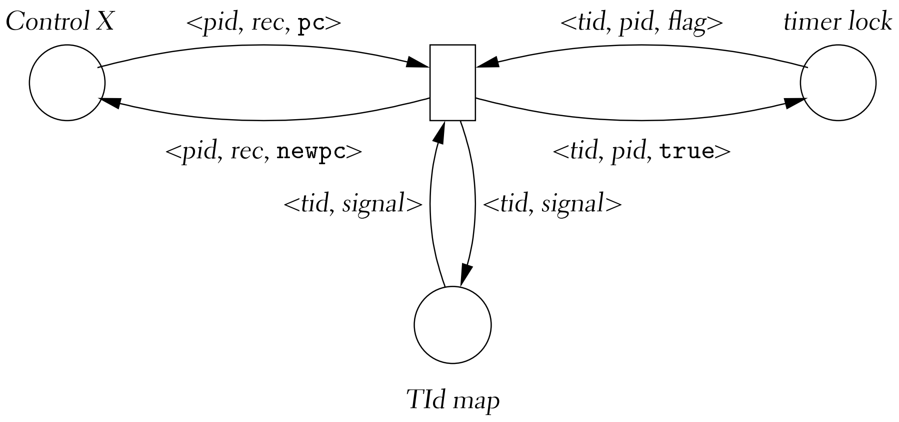


Figure 5.21: EXPIREPOINT START net transition

It must be noted that if the timer parameters contain variables and their values change between the SET and EXPIREPOINT START statements, the wrong timer expiration window will be opened because the timer is identified by the values of its parameters at the time the statement is interpreted.

5.7.4 Modeling Timer Expiration

Figure 5.22 shows a timer expiration net transition. A timer expiration net transition is generated for each process in the system which owns timers. There is no need to create a separate timer expiration net transition for each timer in the system.

The timer expiration transition may fire only if a timer is active and its expiration window is open. The timer expiration transition sets the timer to inactive state and puts the timer signal to the process input queue. In SDL, a timer is in active state until the timer signal has been consumed from the input queue, so there is a slight difference between SDL semantics and the generated model. The end result is same, however: the timer may still be reseted also after it has expired, and the signal be removed from the queue.

5.8 STRUCTURAL CONCEPTS

SDL has a variety of constructs for structuring a specification to smaller parts. Because Petri Nets do not have any structural concepts, the structure of an SDL system can mainly be ignored. There is, however, one issue it has im-

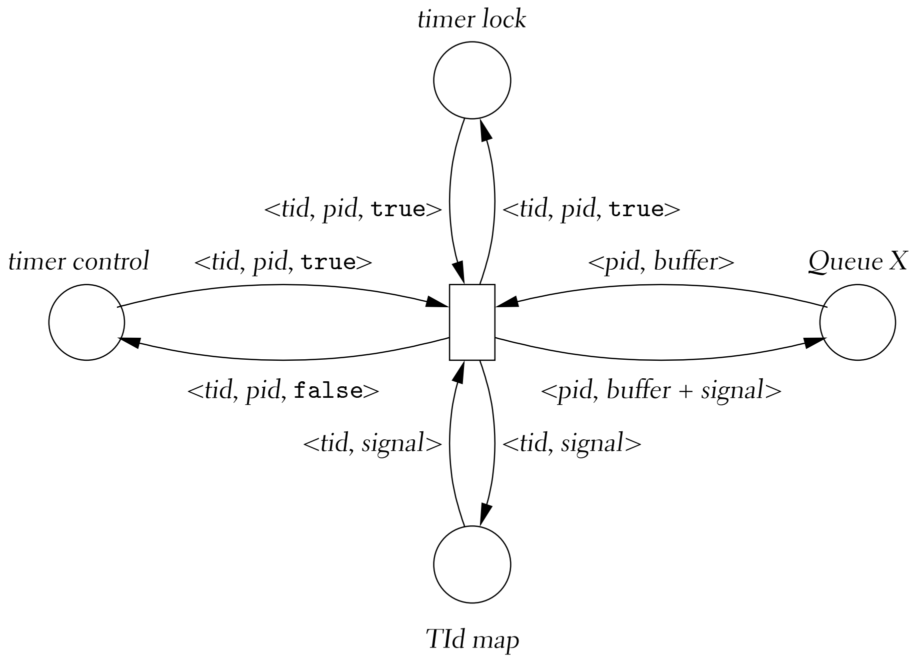


Figure 5.22: Net transition modeling timer expiration

pact on, and that is the OUTPUT statements. The structure of the SDL system determines to which processes a signal can go.

When only the basic components: block, process, channel and signal route, are used, it is quite straightforward to determine the processes to which there exists a path from a specific process. All possible paths consist of either a single signal route, or two signal routes and one channel.

Use of block or channel substructures makes it more difficult to find all possible receivers for a signal, because now the signal can go through a number of channels before reaching the receiver process. If this signal path goes through gates, the case is even more complicated. One way to find all the *signal path endpoints* - processes which may receive the signal - is to traverse the signal path from its origin to all the processes where it ends.

Because OUTPUT statements can have restrictions about the path via which the signal may travel, it is also necessary to keep information about which signal path elements are along the path in question. The most natural way to implement this seemed to be to gather the information about signal paths before starting to generate the net, and then just use the information when needed.

The information is stored so that each signal path element - was it process, channel or something else - has a set of *successors*, other signal path elements to which a signal may go from this specific element. If we select the process interpreting the OUTPUT statement as root, the other signal path elements form a tree structure, leaves being the processes which are reachable from the sending process. If multiple paths lead to same process set, the constructed structure is not a pure tree, but a directed graph with no loops.

When it is time to generate the OUTPUT statement, it is easy to traverse the tree and create a net transition for all possible receivers. Each signal path element also contains information about which signals it is able to carry, so such paths that do not contain the signal being sent can be pruned. VIA path restrictions can be taken into account simply by generating OUTPUT net transitions for only such signal path endpoints to which the path contains one of the VIA path elements.

Gathering information about the successors of a signal path element is a problematic task itself. The following sections describe the problems that arise when dealing with different kinds of structuring constructs.

5.8.1 Block Substructure

In SDL, a block may contain both processes and a block substructure. They are then two different views to the same block, and only one of them can exist in a system at any time. Because there is no way for the model generator to know which version of the block to use when generating the net, only pure leaf blocks or sub-structured blocks are allowed by SDL2pn.

Successors for subchannels and the channels in the surrounding block can be resolved by looking only at the channel connections. If a connection contains a subchannel and a channel which are both going out of the block, the channel is a successor for the subchannel. Same applies in the opposite direction.

There are some restrictions that SDL places on block substructures, which possibly could be checked by the front-end, but currently are not. One restriction is that for each channel-subchannel pair going to the same direction in a channel connection, their signal sets should be identical. Also each channel in the enclosing block which is connected to the block containing the substructure should be present in exactly one channel connection.

5.8.2 Channel Substructure

Resolving the successors of a subchannel in a channel substructure is more complicated than in the case of block substructure, because it is not enough to consider the channel endpoint connections but we must also look into the connections in the block being the endpoint.

The channel itself must also exist on the signal path, otherwise output statements with VIA constructs mentioning it will not be translated correctly. It need not appear on a signal path leading out of the channel substructure because processes inside the substructure can not use the channel in VIA constructs. Outgoing subchannels can thus be handled as normal channels.

For incoming subchannels, the superschannel is added as a signal route element between the channel in the enclosing scope and the subchannel. Now a signal sent with a VIA construct mentioning the superchannel will be directed to the substructure. This will, of course, not work for a two-way channel, so each direction of a channel is considered as a separate signal path element.

A channel substructure specification has the side-effect of changing the origin of the signal sent through the channel. This follows from the way how the channel substructure concept is defined.

5.8.3 Types and Gates

A type - was it block, process or service type - can be instantiated in multiple places. Each of these instances must be separately translated. This is implemented by copying the contents of a type to the instances of the type. For example, a process instantiated from a process type has a copy of all the variables and SDL transitions of the original type. A straight consequence of this is that processes instantiated from the same type have equivalent statement numbers. This should not be a problem, because all net places and net transitions generated include the full name of the process, and will thus not be mixed.

If a process type uses VIA constructs in OUTPUT statements, it can not use names of channels or signals routes as VIA path elements, but it must use gates instead. Due to this, gates must be part of signal paths as well. This is implemented simply by adding a gate to the signal path whenever a channel or signal route passes through it.

5.9 PROCEDURES

A procedure body is translated the same way as a process body by translating each transition trigger and SDL statement separately according to rules introduced in 5.5 and 5.6.

Figure 5.23 shows the basic principle of translating a CALL statement. Each CALL statement in the SDL system has a unique wait number which is separable from any statement number in the system. A CALL net transition replaces the program counter value of the calling process or procedure with the wait number. Now the process can not proceed until the wait number is changed back to a normal statement number. A CALL statement also adds new tokens to procedure control place, procedure variable places and procedure parameter places.

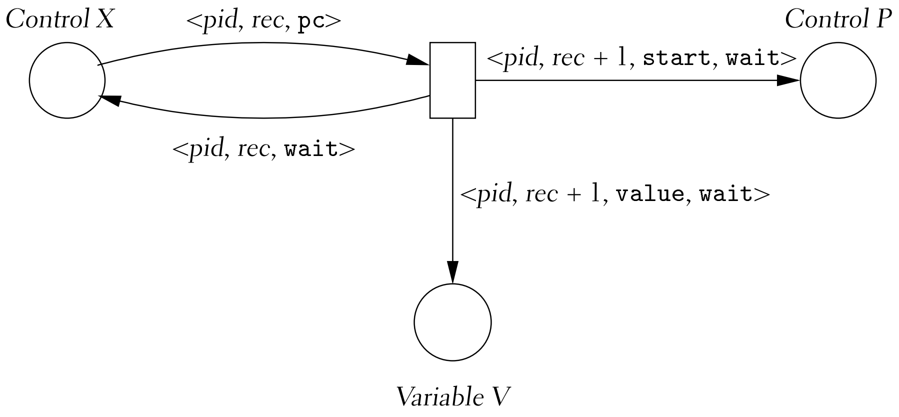


Figure 5.23: CALL net transition

In the case of a recursive procedure, there may be multiple tokens with the same **PId** in the procedure control place and the procedure variable places. The recursion depth (*rec*) is used to separate these from each other. On a given recursion level, only the tokens with the same recursion depth value are used. A **CALL** net transition creates new tokens with the recursion depth incremented by one.

As a procedure may be called multiple times in a process, there must indeed be a separate **RETURN** net transition matching all these calls. Further, there may be more than one **RETURN** statement in a procedure, so a separate **RETURN** net transition has to be generated for each **CALL-RETURN** pair.

The structure of a **RETURN** net transition is shown in Figure 5.24. The correct **CALL** statement is identified by the wait number, and the program counter of the calling process or procedure is changed to the statement number of the statement following the **CALL**. Also all tokens added in the **CALL** net transition are removed in the **RETURN** net transition.

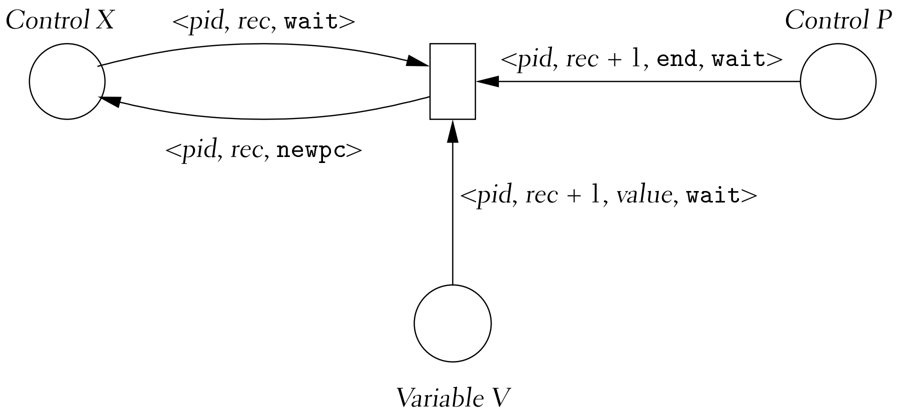


Figure 5.24: **RETURN** net transition

Some of the procedure parameters may be **IN/OUT** parameters, which means they can return a value. For these parameters, the **RETURN** net transition must move their values to the places of actual parameters, corresponding variables in the calling process or procedure. The actual parameters are naturally call-specific, so when translating a **RETURN** statement, all calls for the procedure containing it must be known.

A procedure may be value-returning, in which case the **RETURN** transition must also update the value of the variable to which the return value is being assigned. Because the SDL statements were simplified so that procedure calls appear only as the only one expression in a statement (see 5.6.5), the assignment of the return value can be done in the same transition which removes the tokens from the procedure places. If procedure calls were allowed to exist as parts of more complex expressions, for example as parameters for another procedure call, the translation would have been far more complex.

If a procedure is defined outside the calling process or procedure, the call should be transformed into a call of a local, implicitly created subtype of the procedure [34]. This has been implemented by making a copy of the procedure to each process which calls it, either directly or indirectly (by procedures called by the process). As a result of this transformation, there is at most one calling process for each procedure. Now it is easy to determine which queue to use when translating **INPUT** constructs of a state in a procedure.

6 OPTIMIZING THE MODEL

Using the previously presented translation rules it is possible to create a formal model of an SDL program utilising almost any SDL language constructs. However, the resulting model may be quite *inefficient* in a sense that the set of reachable global states of the system can be excessively large, and thus analysing the state space may require immoderate amounts of space and time. This is called *state space explosion*.

When the intention is to analyse industrial-size programs, the state space explosion is a severe problem. Although there are various methods to reduce the size of the reachability graph generated from a Petri Net model, it is also possible to optimise the model itself. These kind of optimisations usually transform the net to disallow some execution sequences or leaving out superfluous data.

Optimisations done on the Petri Net model do not usually reduce the size of the model, actually they may introduce additional structure to the net. Instead, their aim is to reduce the size of the reachability graph generated from the net. Figure 6.1 shows the relationship between the initial SDL model of a program, its Petri Net model and the reachability graph. The tools presented in the figure, SDL2PN and MARIA, are the tools of interest in the scope of this thesis. Naturally other tools can be used or the net model may be constructed by hand.

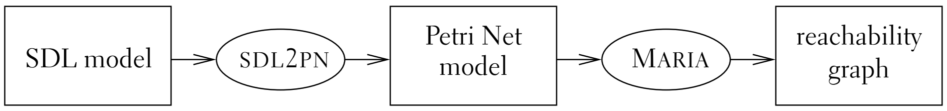


Figure 6.1: Different views of a system

The time to generate the Petri Net model from an SDL program, and also the space consumed by the model are proportional to the size of the SDL model, because except few exceptions, each SDL statement is modeled by one net transition. The time it takes to create the net model is really short (if done automatically) compared to the time required to generate the reachability graph, where the number of states may grow exponentially compared to the size of the model. Creating a slightly more complex net model is thus justified if it results in a smaller reachability graph.

Nevertheless, optimizations do not come without a price. When disallowing some execution sequences of a system (reducing concurrency), some perfectly legal system states are removed from the model. If an error state is optimized away, the error will not be found in the analysis. Similar concerns arise when abstracting data. Extreme care must be taken in the model optimization, so as not to exclude any integral behaviour of the system from the model. In most cases, some optimizations have to be done, otherwise the system could not be analyzed at all due to the size of its state space. All in all, the user should be permitted to define whether to use specific optimization method or not.

6.1 ATOMIC SECTIONS

The basis for the state space explosion is the fact that when a pair of processes is executing independenly of each other, all the possible interleavings of their atomic actions must be considered. This creates a large amount of intermediate states between synchronization points of the processes. Figure 6.2 (a) shows a simple example of the state space generated by two independent processes, both of which have four local states numbered from 0 to 3.

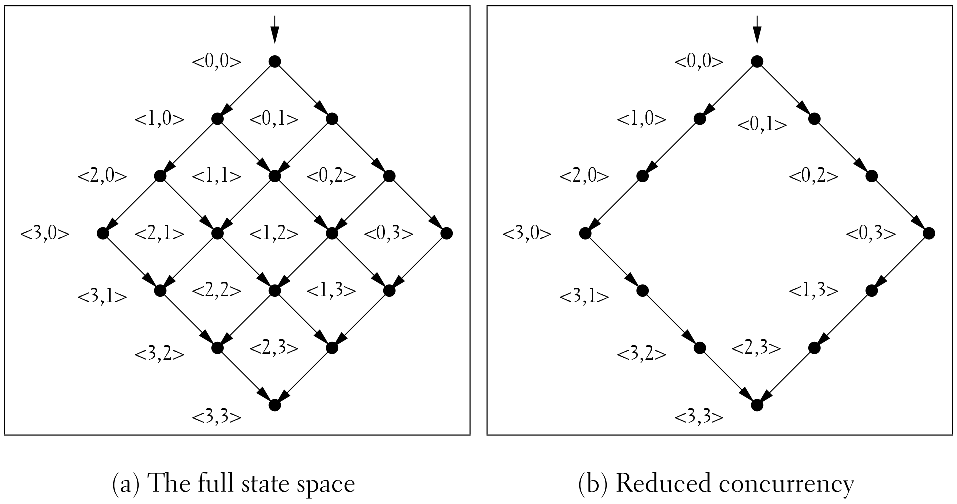


Figure 6.2: An example of state spaces of two processes

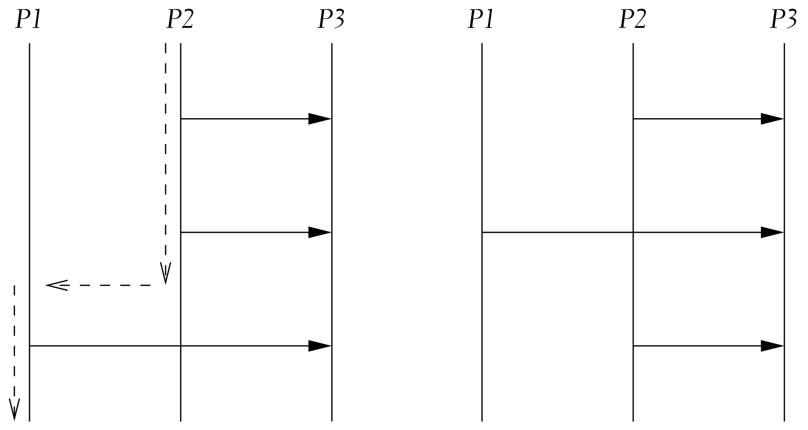
If there are more processes, the number of states naturally grows even faster. The intermediate states between the points where the processes communicate with each other are usually not essential to the analysis of the model, because they represent the sequential parts of the program, which can be treated quite well using other methods suitable for sequential programs.

The intermediate states can be abstracted away by allowing only one process at a time to execute, as seen in Figure 6.2 (b). In a point where processes communicate with each other, it must be allowed to swith the executing process. The parts of program where a process executes independently of other processes are called *atomic sections*. If the atomic sections are long, as usually is the case with SDL programs, disabling interleavings within them may lead to considerable decrease in the number of global system states.

Handling atomic sections is based on the approach described in [21]. A *resource* place is added to the net. It contains one token with a boolean value. A process may begin executing an atomic section only if the *resource* place contains value `true`. At the same time, the value is changed to `false`. At the end of the atomic section, the resource token value is set back to `true`.

A simple implementation would claim the resource token in an INPUT net transition, and release it in a STOP, RETURN and NEXTSTATE net transitions. The first problem with this approach is, that there is no net transition for the NEXTSTATE statement because it was combined with the transition trigger net transitions as described in 5.5. The resource token must thus be released in the net transition *preceding* a transition trigger net transition.

Second, and more severe, problem is that the atomic sections implemented this way are too coarse. Figure 6.3 shows a situation that may arise when using this kind of atomic sections. There are three processes in the example, named *P1*, *P2* and *P3*. *P1* sends one signal to *P3*, and *P2* sends two signals to *P3* in the same SDL transition. In the Figure 6.3 (a) atomic sections are used, and the dashed line shows how the resource token is passed from one process to another. Because the resource locking mechanism forces the whole SDL transition to be executed before other processes get their turn, the two signals process *P2* sends arrive to the queue of *P3* always immediately following each other: no other signal can arrive between them. On the other hand, if atomic sections were not used, the situation might be as in Figure 6.3 (b), where process *P1* sends a signal to *P3* between the two signals sent by *P2*. In this case the use of atomic sections clearly excludes some potentially crucial execution sequences of the system.



(a) Atomic sections used (b) All interleavings allowed

Figure 6.3: An example of interesting executions being left out by reducing concurrency

The problem can be solved by releasing the resource token temporarily while an OUTPUT statement is being executed. Basically, there are two possibilities: either release the resource token in the OUTPUT net transition and claim it back in the following net transition, or release the resorce token in the transition preceding the OUTPUT and reclaim it in the OUTPUT net transition. Both approaches allow changing the executing process when signals are sent, and thus do not exclude any orderings in which signals may be sent. Here, the second approach is selected because it allows for simpler translation.

SDL statements can be divided to two classes: *resource claiming* statements (**NEXTSTATE**, **OUTPUT**), and those that do not claim the resource token. The statements that do not claim the resource token can be furher divided to *resource releasing* statements (**STOP**, **RETURN**) and other statements. The statements, that are neither resource claiming nor releasing, release the resource token if and only if the following statement is a resource claiming one. The resource claining statements take the resource token if and only if the following statement is *not* another resource claiming statement.

The DECISION statement requires a bit more consideration, because the statement following it is selected dynamically. Here becomes apparent why we modeled the resource as a boolean value instead of just a token which either is in the *resource* place or not. The DECISION net transition takes the resource token out of the *resource* place and puts it back, but the value of the token is selected dynamically based on the selected execution branch. If the branch starts with a resource claiming statement, `true` is returned to the resource distributor place, otherwise `false`.

Also procedure calls may need releasing the resource token, if the procedure being called contains states. In this case, the statement preceding the call must release the resource token, and the following statement must reclaim it. In the case of a procedure which does not contain states, the whole procedure may be executed in the same atomic section.

Translation of the RETURN statement is made more complicated by the fact that the resource token is always released by a net transition preceding a resource consuming statement. A RETURN may, in fact, *be* the preceding statement when the execution of the program is concerned. Whether the following statement is resource consuming or not, depends on the CALL to which the RETURN is made. This information is, however, available on the translation time and because a separate RETURN net transition is generated for each CALL in any case, it is quite simple to release the resource token only in the correct net transitions.

The implementation of atomic sections described here is not as strict as it could be. Returning to the example in Figure 6.3, to achive the correct behaviour it is enough to release the resource token once, between the two OUTPUT statements of process *P2*. In this implementation the resource token would actually be released three times: before the first OUTPUT statement of *P2*, again before the second OUTPUT statement of the same process and also before the single OUTPUT statement of process *P1*. This leaves a number of unnecessary interleavings to the reachability graph. However, decreasing the number of times the resource token is released would make the translation considerably more complex.

6.2 INTERPRETING PId EXPRESSIONS ONLY WHEN USED

The **PId** of the sender is sent with each signal in a SDL system. This value is saved in the global *PId expression* place for each process separately. For each process, this place contains also process identifiers of the parent and offspring processes. It is possible that a process never uses these values, in which case they are unnecessarily stored. Additionally, their values affect the global state of the system, which in turn creates additional states to the reachability graph of the model. If a process does not contain special expressions SENDER, PARENT or OFFSPRING, all references to the *PId expression* place can be removed from all the net transitions of the process. The SELF expression may appear in the process, because it refers to the process identifier of the process itself which does not need to be fetched from the *PId expression* place.

7 IMPLEMENTATION TOOLS

The SDL parser and the initial implementation of the model generator had been implemented in the C++ language [29], so it was quite natural to continue using the same language. C++ is fairly suitable for the purpose due to its efficiency and portability [1].

On the other hand, C++ programs are particularly vulnerable to a problem called *memory leaks*. A program that dynamically allocates memory, and does not deallocate it when it is no longer used, is said to have a memory leak. A program with memory leaks will usually require more memory at run-time than an equivalent program with correct memory deallocation. *Garbage collection*, automatic deallocation of objects, is a powerful cure for the problem, but involves some run-time overhead. C++ does not have garbage collection, and because the implementation of the SDL front-end makes heavy use of dynamic allocation of objects and complex data structures, memory leaks were quite probable. There are not many freely available memory debugging tools. We used a tool called LeakTracer [1], which detects both memory leaks and situations where memory is tried to be deleted but which is not allocated. There were some problems using LeakTracer with the C++ Standard Template Library, but all in all the tool was very useful.

The SDL parser, which is part of the front-end, uses Flex [24] for generating the lexical analyzer, and Bison [7] for the generating the parser. As the model generator was implemented, the input language of the SDL parser needed to be slightly extended to introduce the special commands for controlling timer expiration.

The model generator is of not much use without the MARIA analyzer, because it only generates output in the MARIA input language. Additionally, the model generator depends on a MARIA static library, which must be present in a system when the model generator is compiled. For this reason, it is convenient to keep the tool set used in the implementation of the model generator close to that used in the implementation of the MARIA analyzer.

[1]A portable language means that a program developed for one system in such a language using a standard-conforming compiler can be compiler for (ported to) any system using some other compiler.

8 IMPLEMENTATION DETAILS

While an SDL specification is being parsed, a syntax tree is constructed. The syntax tree contains nodes of different types. The nodes can be divided to four groups: ENTITY, EXPRESSION, STATE and SORT. [18] presents a more detailed description about the syntax tree node classes.

After generating the syntax tree, references to other nodes in the tree are resolved and some semantic checks are done, for instance checking the number and types of parameters for procedure calls. Only after semantic checks are passed, the model generation phase starts.

In the model generation phase the syntax tree is traversed multiple times: at first the data types are generated, after that, places are added to the net and finally the net transitions are created.

Basically, the code concerning the net generation is contained in the syntax tree nodes. Each node contains a `generate` -function, which will generate the part of the net associated with the generating node. To be more specific, there are separate functions for each of the three phases, and additionally some functions having to do with the translation of expressions.

To reduce the amount of redundant code in the syntax tree nodes, some classes are added which encapsulate the most frequent net generation actions. Each group of places in the net is represented by a class. Some place types, like the `PID` pool place, appear just once in the generated net, while for some place types, for instance a queue place, an instance is created for each process in the system. The different place classes have operations allowing the creation of the corresponding net place, adding initial marking to it, and adding arcs between a transition and the place to the net.

The whole model generation is controlled by an instance of `Generator` class. It is a class which stores the shared resources used during the net generation phase, and contains operations for creating common types to the net. There is only one instance of the `Generator` class at any given time. The `Generator` object initiates the different phases of the model generation.

9 CONCLUSIONS

The SDL2PN front-end for the MARIA analyzer was created by adding a model generator part to the previously existing compiler front-end for the SDL language, described in [18].

The model generator supports most of the features of SDL-96, including *procedure calls* and *timers*. Most notable features of SDL-96 which are not yet supported by the model generator are *virtuality* constructs and *remote procedure calls*.

The model generator does the translation in phases: first the static structure of the SDL system is analyzed, then user-defined data types are translated and places and their initial markings are created, and finally net transitions are generated to model the statements in the SDL processes. Usually only one net transition is required to model an SDL statement, but some statements like an **OUTPUT** statement with a **VIA** constraint may require more than one net transition.

Most SDL language constructs were quite straightforward to translate to MARIA input language, largely due to the versatile type system of the MARIA language and the clean and efficient programming interface of the MARIA library. The translation of **VIA** and **VIA ALL** constraints of the SDL **OUTPUT** statement was the most difficult to implement. Another difficulty was dealing with erroneous SDL input.

Some features, such as the *resource place*, were added to the model generator to allow some model-level optimization to alleviate the state space explosion problem. Most of the work in reducing the complexity of the analysis is, however, left to the MARIA analyzer.

The SDL2PN front-end consists of a parser for SDL-96 and a model generator for MARIA input language. It reads an SDL system description in SDL/PR format and generates a text file containing the high-level Petri net model which can be read and analyzed by the MARIA tool. The combination of SDL2PN and MARIA can be used to analyze even quite large SDL systems without having to manually construct the model of the system.

BIBLIOGRAPHY

[1] Erwin Andreasen and Henner Zeller. LeakTracer README. http://www.andreasen.org/LeakTracer/README.html, August 2003.

[2] *SDL formal definition.* Recommendation Z.100. International Telecommunication Union, Geneva, Switzerland, March 1993.

[3] Dragan Bošnački, Dennis Dams, Leszek Holenderski, and Natalia Sidorova. Model checking SDL with Spin. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 – April 2, 2000, Proceedings*, volume 1785 of *Lecture Notes in Computer Science*, pages 363–377. Springer-Verlag, Berlin, Germany, 2000.

[4] Marius Bozga, Susanne Graf, and Laurent Mounier. Automated validation of distributed software using the IF environment. *Electronic Notes in Theoretical Computer Science*, 55(3), 2001.

[5] T. G. Churina. Coloured Petri net approach to modeling of SDL specifications with dynamic constructs. In *Joint Bulletin of the Novosibirsk Computing Center and A.P. Ershov Institute of Informatics Systems*, Computer Science, pages 18 – 39, Novosibirsk, 2000. NCC Publisher.

[6] T. G. Churina, M. U. Mashukov, and V. A. Nepomniaschy. Towards verification of SDL specified distributed systems: Coloured Petri nets approach. In *Proc CS&P'2001*, pages 37 – 48. University of Warsaw, Poland, 2001.

[7] Charles Donnelly and Richard Stallman. Bison - The YACC-compatible Parser Generator. http://www.gnu.org/manual/bison-1.25/html_chapter/bison_toc.html, November 1995.

[8] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL Formal Object-oriented Language for Communicating Systems.* Prentice Hall Europe, 1997.

[9] Hans Fleischhack and Bernd Grahlmann. A compositional Petri net semantics for SDL. In Jörg Desel and Manuel Silva, editors, *Application and Theory of Petri Nets 1998, 19th International Conference, ICATPN'98, Lisbon, Portugal, June 22–26, 1998, Proceedings*, volume 1420 of *Lecture Notes in Computer Science*, pages 144–164. Springer-Verlag, Berlin, Germany, 1998.

[10] Hartmann J. Genrich. Predicate/Transition Nets. In *Lecture Notes in Computer Science 254: Advances in Petri Nets 1986, Part I*, pages 208 – 247, 1986.

[11] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[12] Gerard J. Holzmann and Joanna Patti. Validating SDL specifications: an experiment. In Ed Brinksma, Giuseppe Scollo, and Chris A. Vissers, editors, *PSTV IX, Proceedings of the IFIP WG6.1 Ninth International Symposium on Protocol Specification, Testing and Verification, Enschede, The Netherlands, June 6–9, 1989*, pages 317–326. North-Holland, Amsterdam, The Netherlands, 1990.

[13] Nisse Husberg. SDL Modelling with High Level Petri Nets. In *Workshop on Concurrency, Specification and Programming, Berlin, September 1996*, 1996.

[14] Nisse Husberg, Markus Malmqvist, and Tero Jyrinki. Emma: A Tool For Analysis of SDL Programs. Technical report, Helsinki University of Technology, December 1996.

[15] Robert P. Kurshan, Vladimir Levin, Marius Minea, Doron A. Peled, and Hüsnü Yenigün. Verifying hardware in its software context. In *Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design, ICCAD'97, San Jose, CA, USA, November 9–13, 1997*, pages 742–749. IEEE Computer Society Press, Los Alamitos, CA, USA, 1997.

[16] Vladimir Levin and Hüsnü Yenigün. SDLcheck: A model checking tool. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18–22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 378–381. Springer-Verlag, Berlin, Germany, 2001.

[17] Markus Lindqvist, Erkki Ruohtula, Esa Kettunen, and Heikki Tuominen. *The TNSDL Book*. Nokia Telecommunications Oy, 1995.

[18] Marko Mäkelä. Implementing the Front-End of an SDL Compiler. Master's thesis, Helsinki University of Technology, November 1998.

[19] Marko Mäkelä. A Reachability Analyzer for Algebraic System Nets. Licenciate's thesis, March 2000.

[20] Marko Mäkelä. *Maria, Modular Reachability Analyzer for Algebraic System Nets*, September 2001.

[21] Markus Malmqvist. Methodology of Dynamical Analysis of SDL Programs Using Predicate/Transition Nets. Master's thesis, Helsinki University of Technology, April 1997.

[22] Andreas Mitschele-Thiel. *Systems Engineering with SDL*. John Wiley & Sons, Ltd, 2001.

[23] V.A. Nepomniaschy, V.S. Argirov, D.M. Beloglazov, A.V. Bystrov, T.G. Churina, M.Yu. Mashukov, and R.M. Novikov. Modeling and verification of SDL specified distributed systems using high-level Petri nets. In Gabriela Lindemann, Hans-Dieter Burkhard, Ludwik Czaja, Holger Schlingloff, Andrzej Skowron, and Zbigniew Suraj, editors, *Workshop:*

Concurrency, Specification and Programming, CS&P'2004, Caputh, Germany, Septemberber 24–26, 2004, Volume 1: Petri Nets and Automata, pages 100–111. Informatik-Bericht Nr. 170, Institut für Informatik, Humboldt-Universität zu Berlin, Germany, 2004.

[24] G. T. Nicol. *Flex: The Lexical Scanner Generator for Version 2.3.7.* Free Software Foundation, February 1993.

[25] A. Olsen and O. Færgemand. *Systems Engineering Using SDL-92.* Elsevier Science B.V., 1994.

[26] Wolfgang Reisig. Place/Transition Systems. In *Lecture Notes in Computer Science 254: Advances in Petri Nets 1986, Part I*, pages 117 – 141, 1986.

[27] Anders Rockström. *An Introduction to the CCITT SDL.* Televerket Stockholm, 1985.

[28] Mehmet Alper Şen. Verification of SDL systems with partial order methods. Master's thesis, Department of Electrical and Electronics Engineering, Middle East Technical University, Ankara, Turkey, May 1997.

[29] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1997.

[30] P.S. Thiagarajan. Elementary Net Systems. In *Lecture Notes in Computer Science 254: Advances in Petri Nets 1986, Part I*, pages 26 – 59, 1986.

[31] Daniel Toggweiler, Jens Grabowski, and Dieter Hogrefe. Partial order simulation of SDL specifications. In Rolv Bræk and Amardeo Sarma, editors, *SDL'95: With MSC in CASE, Proceedings of the 7th International SDL Forum, Oslo, Norway, September 26–29, 1995*, pages 293–306. Elsevier/North-Holland, Amsterdam, The Netherlands, 1995.

[32] Heikki Tuominen. Embedding a dialect of SDL in PROMELA. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999 & Toulouse, France, September 21 and 24, 1999, Proceedings*, volume 1680 of *Lecture Notes in Computer Science*, pages 245–260. Springer-Verlag, Berlin, Germany, 1999.

[33] Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen, and Tino Pyssysalo. PROD Reference Manual. Technical Report B 13, Helsinki University of Technology, August 1995.

[34] *CCITT Specification and Description Language (SDL).* Recommendation Z.100. International Telecommunication Union, Geneva, Switzerland, March 1993.

[35] *Corrections to Recommendation Z.100, CCITT Specification and Description Language (SDL).* Recommendation Z.100. International Telecommunication Union, Geneva, Switzerland, October 1996.

ISBN 951-22-7404-3
ISSN 0783-540X