Computer Support for SESIS: A User Manual

Mark Andrew
mark.andrew@de.bosch.com

September 8, 2006

Contents

List of Figures

1 Introduction

This document describes the various forms of computer support for SESIS (Systems Engineering for Software Intensive Systems) [1]

Support is offered in three main areas:

- language : the integration of the Real Time Modeling Extensions (RTMX) into a core language.
- graphics : support for graphically creating and editing models using these extensions.
- simulation : support for the execution, observation and testing of these models using a simulation framework embodying the concepts of Perfect Technology [3].

This development of this software is driven by the following qualities:

- Simplicity of Installation
- Portability
- Flexibility
- Robustness (against internal errors)
- Openness
- Problem and User Scalability (simple things are simple, hard things are possible)

2 A Gentle Introduction to the Main Features

The SESIS Modeling Language is a superset of Modula-3 [4] [5]. In the following sections we will look at some simple models to get a feel for the extent of the language. Note that the following examples are purely intended to give you a feel for the features of the language and do *not* correspond in any way to specific SESIS artefacts (domain models, domain architectures etc.) For further guidance here the reader is refered to [2].

2.1 Our First Capsule

Here is a simple example of a Capsule Interface, modeling a bank account:

```
CAPSULE INTERFACE Account ;
PORT movement : PROTOCOL
    INCOMING MESSAGE deposit(sum : INTEGER) ;
    INCOMING MESSAGE withdraw(sum : INTEGER) ;
END;
```

```
PORT information : PROTOCOL
    INCOMING MESSAGE requestBalance() ;
    OUTGOING MESSAGE balance(sum : INTEGER) ;
END;
END Account.
```

This text tells us that instances of this capsule understand two groups of messages, one group concerning `movement` and the other concerning account `information`.

The `movement` group specifies two messages, one each to withdraw and deposit sums of money. We see that the messages themselves can have associated parameters in fact, at first glance they look pretty much like procedure specifications. We will learn about the differences later.

The `information` group specifies one message - `requestBalance` - which it will expect to receive from the outside world and one message `balance` which it transmits to the outside world. Note that the order is not explicitly defined here.

2.1.1 So What Does It Do?

All we have done so far is to show the capsule from the outside. If we are to understand what it does then we have to look inside. Here is the body for that capsule.

```
CAPSULE Account ;
  VAR money : INTEGER := 0;
  ACTIVITY deposit (sum : INTEGER) =
    VAR newtotal : INTEGER;
  BEGIN
    newtotal := money + sum;
    money := newtotal;
  END deposit ;
  ACTIVITY withdraw (sum : INTEGER) =
    VAR newtotal : INTEGER;
  BEGIN
    newtotal := money - sum;
    money := newtotal;
  END withdraw ;
  ACTIVITY requestBalance () =
  BEGIN
     SEND balance(sum := money);
  END requestBalance ;

BEGIN

END Account.
```

The capsule body contains four entities, the variable `money` and three activities corresponding to the incoming messages listed in the interface. We should intuitively be able to read that the text is telling us how the incoming messages described in the interface are handled and how the outgoing message is generated. The simplest way to handle an incoming message is to define an `activity` with the same name as that message. Any parameters must match those in the interface.

The variable `money` is declared in the outermost scope of the capsule and is visible to all other elements. On the other hand the two variables named `newtotal` are local to the activities `withdraw` and `deposit` and are therefore only visible in those activities. Note also that, unlike `sum`, they do not survive from one invocation to the next.

Similarities to the procedures of conventional programming languages stop here however. The parameters of activities are always passed by value and activities cannot have a return value. Information is passed only via messages, so if we want to communicate our results in any way we have to send a message. This is done using the `SEND` construct, in which that keyword precedes the procedure-like invocation of a specified outgoing message, in this case `balance`

2.1.2 A Graphical Equivalent

In parallel with the textual form, we can also display our capsule graphically. If we load the text shown above into the capsule editor (see Section 4 below) we should see (possibly after moving and resizing some of the objects) something like Figure 1.

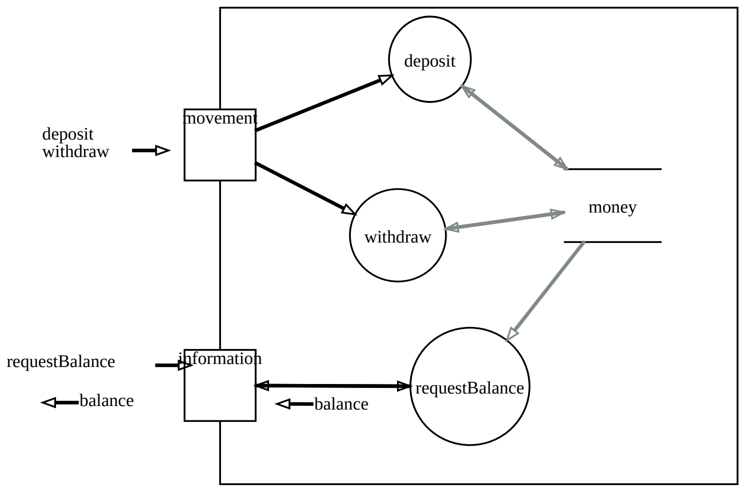


Figure 1: The Account Capsule

The Interface details are represented by the port and message components at the edge of the capsule on the left, and the activities have become circles. The capsule-level variable `sum` is represented by two parallel bars; any resemblance to nearly 20-year-old DeMarco SADT graphical notation is entirely intentional!! Although these variables require no additional syntax, we may refer to them as *datastores* rather than simply variables to strengthen this association. Note that, apart from the position and size information (which is stored in a separate file) all features of the diagram are derived directly from the text of the model. For example, the arrows connecting the activities with the datastores have been derived by doing a simple dataflow analysis across the capsule text. Note also the two different colors for the arrows, grey for data and black for messages.

2.1.3 Executing the Model

When we execute a model, we feed it with a set of test inputs and we can record or observe the outputs generated. First of all we need to compile the model text as follows:

```
m3 Account.i3
m3 Account.m3
```

Then we need some inputs to test the model. We will start with the following in the file `Account.inp`:

```
> deposit sum = 100
> withdraw sum = 10
> requestBalance
```

If this text is stored in the file `Account.inp` then the following invocation

```
runcap Account
```

will run the capsule, feeding it with the inputs and storing the results of the test in two files, a results file `Account.res` showing only the inputs and outputs, and a protocol file `Account.pro` which contains more detailed contextual information. These are stored by default in the `res` subdirectory. In the protocol file we might see something like the following:

The results file makes less interesting reading for us, but is more useful in comparing different models which should exhibit the same behaviour.

We can also echo the protocol straight to the console as part of the run by using the following command

```
runcap Account -cpro
```

2.2 Nesting and Instantiation of Capsules

Capsules are the building blocks of systems. This involves the definition of new Capsule *types*, composed, amongst other things, from *instances* of existing Capsule *types*.

2.2.1 A Second Component for the Bank

For our example we will use our Account capsule as one of the building blocks for a trivial virtual Bank. We first develop a second capsule `ATM` which models the behaviour of an Automatic Telling Machine. This has the following specification:

```
CAPSULE INTERFACE ATM ;
FROM BankTypes IMPORT Card;

PORT keyboard : PROTOCOL
    INCOMING MESSAGE inputPIN(pin : INTEGER) ;
    INCOMING MESSAGE inputSum(sum : INTEGER) ;
END;
PORT cardSlot : PROTOCOL
    INCOMING MESSAGE insertCard(card : Card) ;
    OUTGOING MESSAGE returnCard();
END;
PORT cashSlot : PROTOCOL
    OUTGOING MESSAGE emitCash();
END;
PORT accounting : PROTOCOL
    OUTGOING MESSAGE withdraw(sum : INTEGER);
END;
END ATM.
```

and the following body:

```
CAPSULE ATM ;
FROM BankTypes IMPORT Card;

  TYPE CardState = {Empty, Inserted, Authorized} ;

  VAR cardInfo : Card ;
  VAR cardState : CardState := CardState.Empty;

  ACTIVITY inputPIN (pin : INTEGER) =
  BEGIN
     IF cardState = CardState.Inserted  AND
        pin = cardInfo.encodedPIN
     THEN
        cardState := CardState.Authorized;
     END
  END inputPIN ;

  ACTIVITY insertCard (card : Card) =
  BEGIN
     cardState := CardState.Inserted;
```

```
      cardInfo := card;
  END insertCard ;

  ACTIVITY inputSum (sum : INTEGER) =
  BEGIN
    IF cardState = CardState.Authorized THEN
       SEND withdraw(sum);
       SEND emitCash();
       SEND returnCard();
       cardState := CardState.Empty;
    END;
  END inputSum ;
BEGIN

END ATM.
```

2.2.2 ATM at a glance

Whereas the graphical representation of the account object was not strictly necessary to get an at-a-glance comprehension of that capsule, we are now rapidly nearing the level of complexity where non-programmers will no longer be able to find their way through the text. That same text (with some positional adjustments) also reveals the more generally presentable diagram in Figure 2 when loaded into the capsule editor.

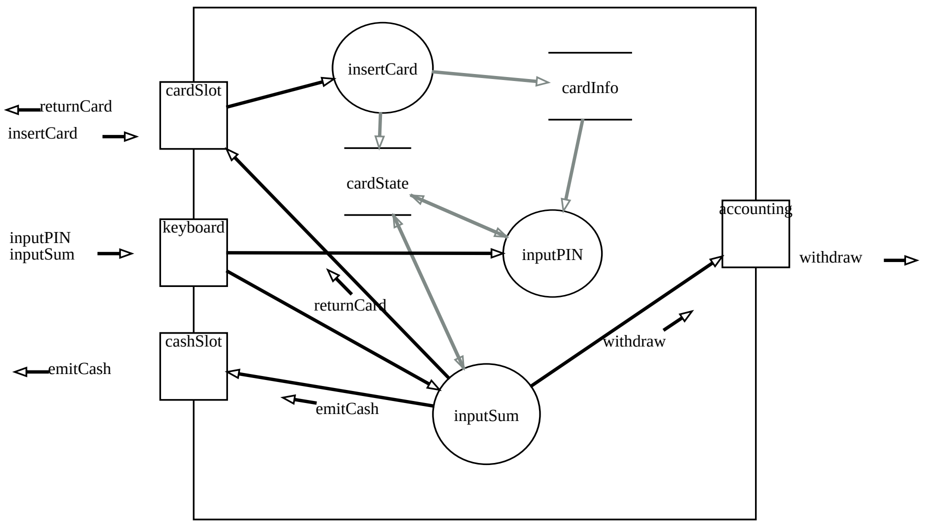


Figure 2: The ATM Capsule

2.2.3 ... and some more Testing

We can test the ATM Capsule by itself just as we tested the Account Capsule. In this case we could use an input file with the following contents:

```
> insertCard card = {encodedPIN: 1234}
> inputPIN pin = 1234
> inputSum sum = 1000
```

and using this command:
`runcap ATM`
we would be rewarded with the following results in `res/ATM.res`

2.2.4 Building the Bank

We now have two components which could be sensibly joined together to give us a new level of functionality. We create a parent Capsule `Bank` which contains one instance each of an `ATM` and an `Account` and connects them together so that withdrawals via the ATM influence the level of the Account. The structure is shown in Figure 3

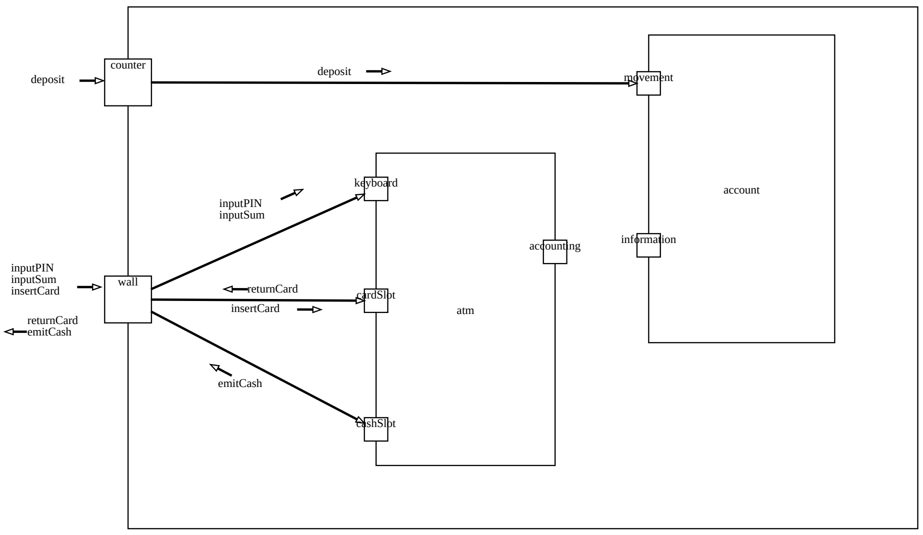


Figure 3: The Bank Capsule

The Bank itself has come to life as a kind of test harness for the components it contains. With respect to the ATM it is just passing through the calls from the hole in its own `wall`. The bank also has a `counter` where deposits can (and in real life must) be made.

The interface of the Bank capsule simply reflects this partition.

```
CAPSULE INTERFACE Bank ;
IMPORT BankTypes;
PORT wall : PROTOCOL
    INCOMING MESSAGE inputPIN(pin : INTEGER) ;
    INCOMING MESSAGE inputSum(sum : INTEGER) ;
    INCOMING MESSAGE insertCard(card : BankTypes.Card) ;
    OUTGOING MESSAGE returnCard();
    OUTGOING MESSAGE emitCash();
END;
PORT counter : PROTOCOL
    INCOMING MESSAGE deposit(sum : INTEGER) ;
END
END Bank.
```

and the capsule body shows the connections between the two child capsules and the parent in textual form.

```
CAPSULE Bank ;
IMPORT BankTypes;
  USECAPSULE ATM ;
  USECAPSULE Account ;
  VAR atm : ATM ;
  VAR account : Account ;
  CONNECT
    counter <=> account.movement;
    wall <=> atm.keyboard;
    wall <=> atm.cardSlot;
    wall <=> atm.cashSlot;
BEGIN

END Bank.
```

2.3 State Machines

In the implementation of the ATM above we structured our behaviour around three clearly identifiable states in which the card could be: either the card was not present at all, or it had just been inserted or it was authorized. Support for this kind of algorithmic structure is included both in the language and in the graphical representation. Here is the original behaviour implemented using this alternative notation:

```
CAPSULE ATMState ;
FROM BankTypes IMPORT Card;
  VAR cardInfo : Card ;
  START = BEGIN
    NEXT empty
  END;
  STATE empty
    ON insertCard(card : Card) =
      BEGIN
         cardInfo := card;
         NEXT inserted;
      END insertCard;
  STATE authorized
    ON inputSum(sum : INTEGER) =
      BEGIN
         SEND withdraw(sum);
         SEND emitCash();
         SEND returnCard();
         NEXT empty;
      END inputSum;
  STATE inserted
    ON inputPIN(pin : INTEGER) =
      BEGIN
        IF pin = cardInfo.encodedPIN THEN
           NEXT authorized;
        END
      END inputPIN;
BEGIN

END ATMState.
```

We can see here that code previously organized into activities is now divided up into states, and into the handlers those states provide for various *transitions*. Just as the activities did, the transitions correspond to messages previously described in the capsule spec. However in the case of transitions the same message may be handled in many states. The graphical equivalent can be seen in Figure 4

2.4 Timers

The run time system deals with time by simulating its passing. This means you can deal with arbitrarily long or short delays without worrying about whether your results are “realistic”. For this example we will consider the influence of time passing on a bank account. If we are lucky we will collect some interest from the bank. The following capsule uses a timer which elapses every year in order to add accrued interest to the account.

```
CAPSULE Interest ;
IMPORT Timer;
  VAR auditTimer : PERIODIC FIXED TIMER  DELAY 1 year ;
      interestRate : REAL := 10.0;
  ACTIVITY setInterestRate (rate : REAL) =
```

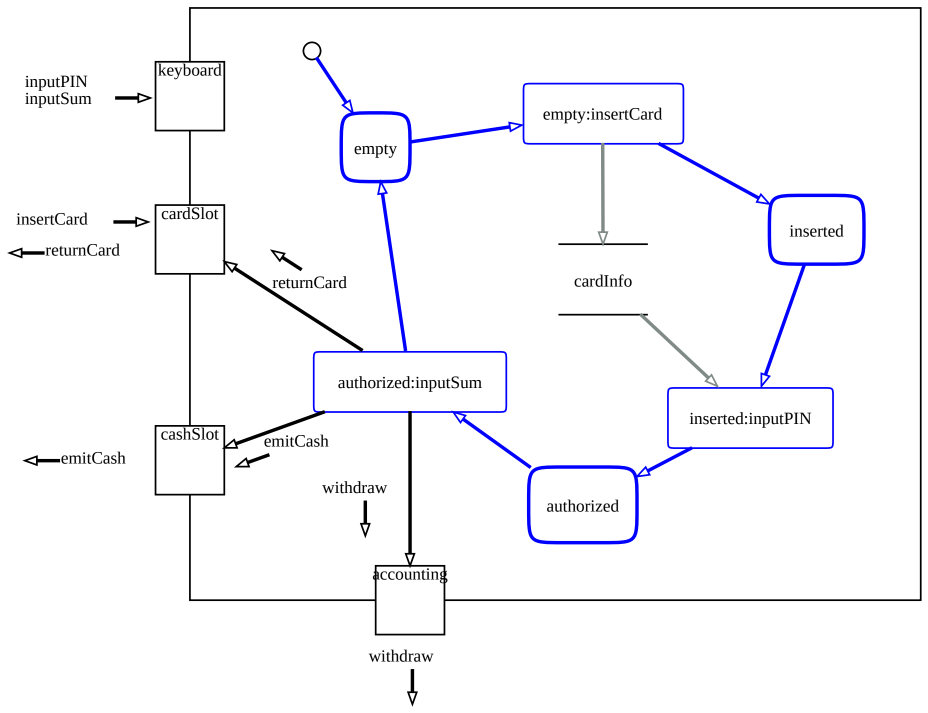


Figure 4: ATM using states

```
  BEGIN
     interestRate := rate;
  END setInterestRate ;
  ACTIVITY updateAccount (sum : INTEGER) =
     VAR accrued : REAL;
  BEGIN
     accrued := FLOAT(sum) * interestRate / 100.0;
     SEND deposit(TRUNC(accrued));
     SEND statement(sum + TRUNC(accrued));
  END updateAccount ;
  CONNECT
    auditTimer -> requestBalance;

BEGIN
   Timer.Start(auditTimer)

END Interest.
```

Graphically the timer can be seen in Figure 5 :

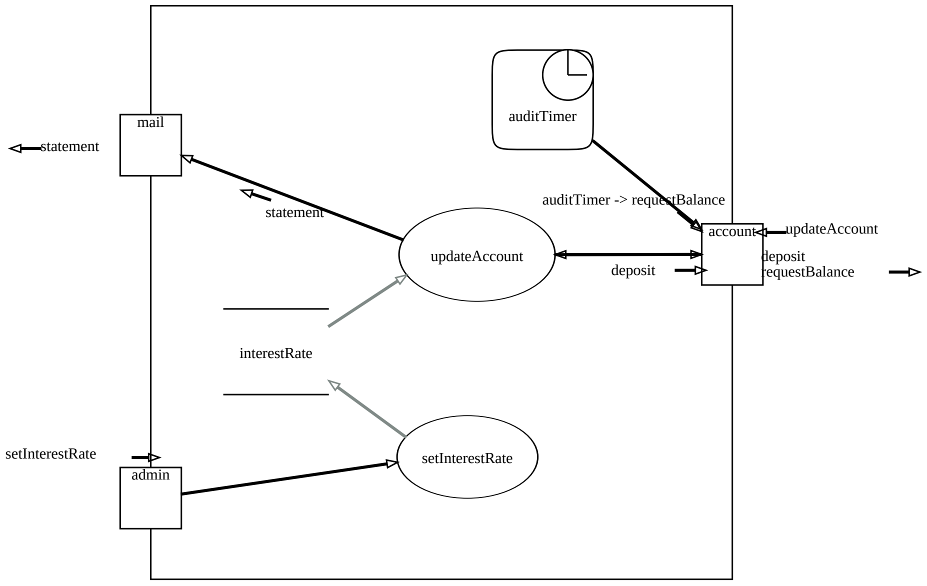


Figure 5: A Timer

The timer is started as soon as the capsule instance is created and, because it has been declared as `PERIODIC` times out once a year without any need for further intervention. The timer is connected directly to the outgoing message requestBalance and this capsule relies on the fact that it will configured by its parent in such a way that sending that message will result in the updateAccount message being subsequently sent back with the current amount of money in the account. For a better solution to this problem see the discussion of callbacks in section 3.7.1

If we feed this new system with the following test data

```
> deposit sum = 100
? 3 year
> insertCard card = {encodedPIN: 1234}
> inputPIN pin = 1234
> inputSum sum = 100
? 1 year
```

then we get the following output

2.5 Excursion: Time Granularity, a Smaller Interval and an Internal Message

We have just seen how to represent a wait of one year. The simulator supports time intervals ranging from 1 picosecond to 10E20 years, which should

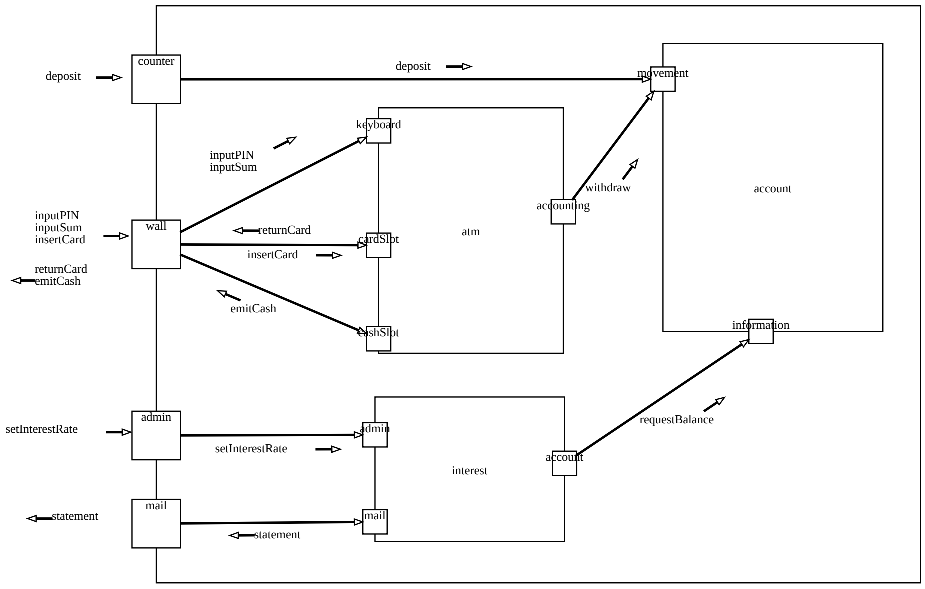


Figure 6: Banking with Interest

be adequate for most phenomena in the automotive domain. Any “waiting” is symbolic, there is no difference in overhead or simulator running time between 1 second and 1 year, the difference is simply that the first interval will elapse before the second if they have started at the same time.

On the other hand, if you declare a periodic timer with a value of 1 second and then submit an input script with a delay of 1 year then you should be prepared for a long wait while the simulator performs over 30 million timeouts, not to mention the message handling code which has been connected to the timer

In the following we refine the implementation of the ATM capsule so that if the customer gives no input within 15 seconds his card is given back to him. This involves changing our state-based ATM to look as follows:

```
CAPSULE ATMTimeout ;
FROM BankTypes IMPORT Card;
  IMPORT Timer;
  VAR cardInfo : Card ;
  VAR inputTimeout : ONESHOT FIXED TIMER  DELAY 15 s ;
  START = BEGIN
    NEXT empty
  END;
  STATE empty
    ON insertCard(card : Card) =
      BEGIN
         cardInfo := card;
         Timer.Start(inputTimeout);
         NEXT inserted;
      END insertCard;
  STATE authorized
    ON inputSum(sum : INTEGER) =
      BEGIN
         SEND withdraw(sum);
         SEND emitCash();
         SEND returnCard();
         NEXT empty;
      END inputSum;
    ON giveUp() =
      BEGIN
         SEND returnCard();
         NEXT empty;
      END giveUp;
  STATE inserted
    ON inputPIN(pin : INTEGER) =
      BEGIN
        Timer.Start(inputTimeout);
        IF pin = cardInfo.encodedPIN THEN
           NEXT authorized;
        END
      END inputPIN;
    ON giveUp() =
      BEGIN
         SEND returnCard();
         NEXT empty;
      END giveUp;

  CONNECT
    inputTimeout -> giveUp;
BEGIN

END ATMTimeout.
```

The following features have been added

- A `TIMER` named `inputTimeout` has been declared with the required 15 second delay. This time we have chosen for it to be `ONESHOT`.
- Two new transitions for the message `giveUp` have been declared, which eject the card and return us to the empty state.
- The timer has been connected to the new message.

Note that the specification, which we do not show, remains exactly as in the previous example. This means that the new transitions are not visible to the world outside this capsule. Further consequences of this separation between specification and body are described in Section 3.1.7 below.

We can test that this has the desired effect with the following input script

```
> insertCard card = {encodedPIN: 1234}   % Put card into machine ...
> inputPIN pin = 1234                    % ... and enter the right PIN but ...
? 16 s                                   % ... forget to ask for money
> inputSum sum = 1000                    % this will be ignored
```

and we are rewarded with the following result

the card has been returned after 15 seconds and the request for money has fallen on deaf ears.

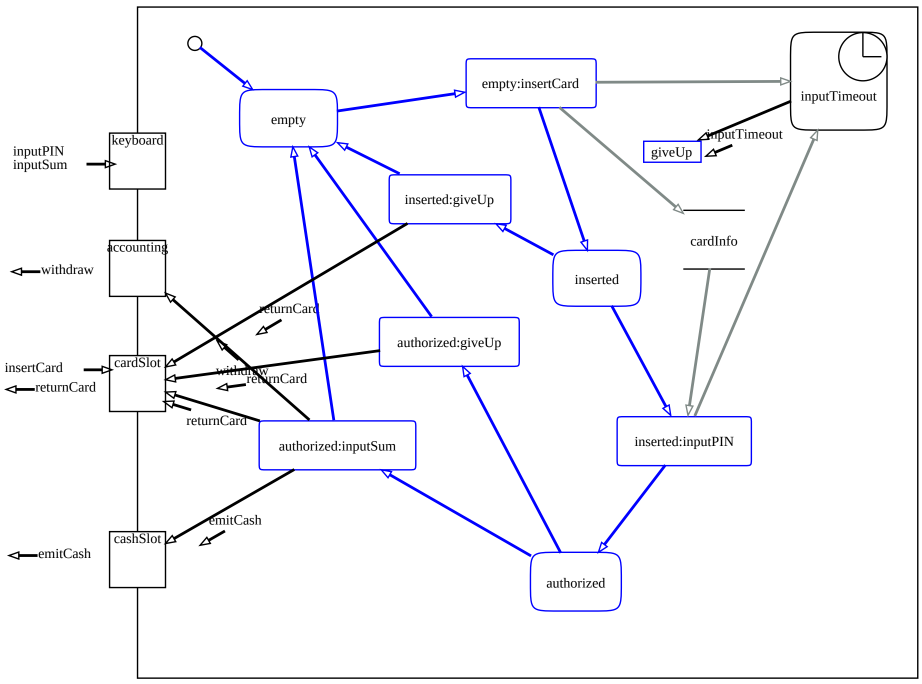


Figure 7: ATM with timeout

The diagram for this capsule introduces a new issue. If we look at Figure 7 we can see that the timer is connected to a free standing element labelled with the name of the transition we want to fire in the state machine. This element is a *transition proxy*, it stands for all transitions of the same name belonging to the states contained within this capsule and is used as a docking point for message connections to those transitions.

Externally visible transitions do not suffer from this problem because the message visible from the outside of the capsule is an unambiguous signifier for the possible multitude of transitions with that name within the capsule.

2.6 Triggers

Now we want to make our ATM somewhat safer. We will do this by telling the machine to retain the card and not to give out any money if the user gets his PIN wrong three times in a row. To do this we first record invalid PIN inputs in a new datastore. Then we define a trigger in terms of the value in that datastore and associate that trigger with a new transition which takes us straight back to the empty state, and resets the failure count on the way.

This results in the following code.

```
CAPSULE ATMSafe ;
FROM BankTypes IMPORT Card;
  VAR cardInfo : Card ;
  TRIGGER shadyCustomer ON failureCount = 3;
  VAR failureCount : INTEGER := 0;
  START = BEGIN
    NEXT empty
  END;
  STATE empty
    ON insertCard(card : Card) =
      BEGIN
         cardInfo := card;
         NEXT inserted;
      END insertCard;
  STATE authorized
    ON inputSum(sum : INTEGER) =
      BEGIN
         SEND withdraw(sum);
         SEND emitCash();
         SEND returnCard();
         NEXT empty;
      END inputSum;
  STATE inserted
    ON inputPIN(pin : INTEGER) =
      BEGIN
        IF pin = cardInfo.encodedPIN THEN
           NEXT authorized;
        ELSE
           failureCount := failureCount + 1;
        END
      END inputPIN;
    ON reject() =
      BEGIN
         failureCount := 0;
      NEXT empty;
      END reject;
  CONNECT
    shadyCustomer -> reject;

BEGIN

END ATMSafe.
```

The corresponding diagram is visible in Figure 8.

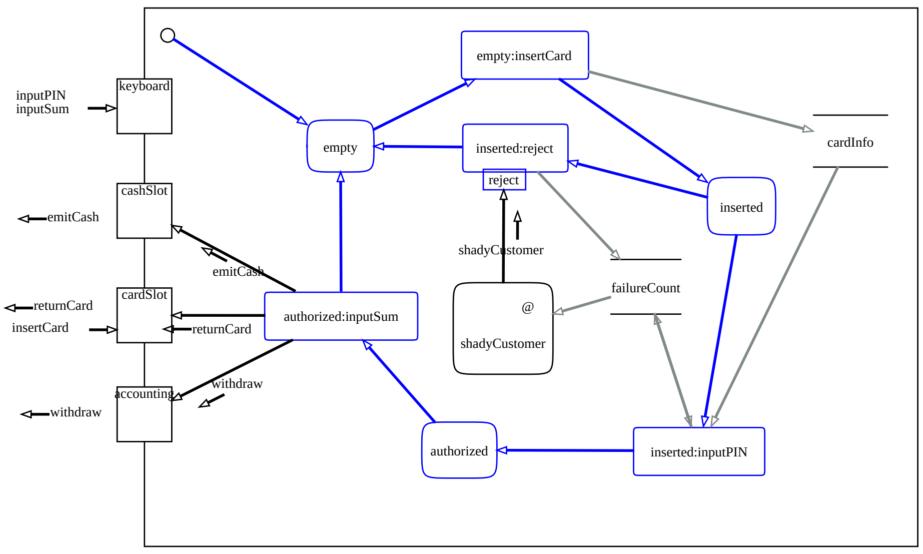


Figure 8: ATM with safety feature

2.7 How Triggers Work

Once defined, a trigger is entered into a global list of all triggers in the system. The test functions of all triggers in this list list are evaluated after every event occurrence. If the test function of any trigger evaluates to true (in the Python sense) for the first time, then a corresponding message is dispatched to the system. Any trigger which has already evaluated to true must thereafter evaluate at least once to false before a subsequent value of true results in a further message dispatch. See Section 3.8 below for details.

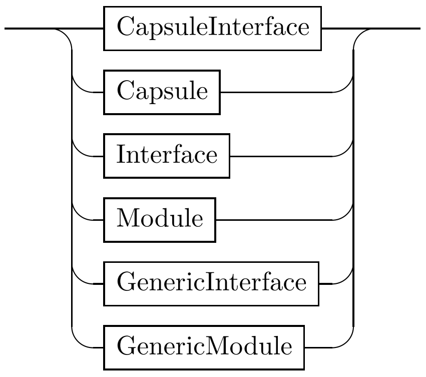
3 Language Elements

There now follow the gory details on all of the new language constructs. Also included are railroad-style syntax diagrams which, when taken together with the existing syntax for Modula-3, completely specify the Sesis Modeling Language.

3.1 New Compilation Entities

Modula-3 allows separate compilation and partitioning of programs according to modules alone. The *generic* feature is nothing more than a factory for creating new modules. As can be seen in the following syntax diagram we introduce two new compilation entities. (Note generic capsules would be a possibility for a future enhancement)

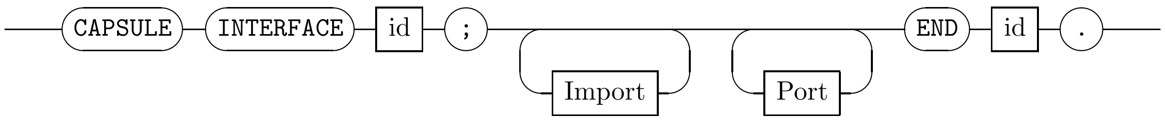
Compilation



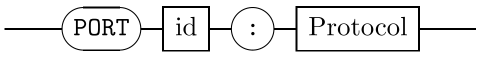
3.1.1 Capsule Interface

The syntax is as follows

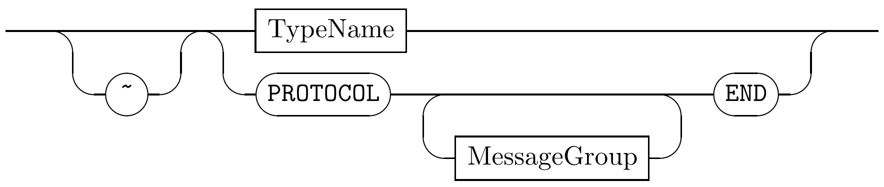
CapsuleInterface



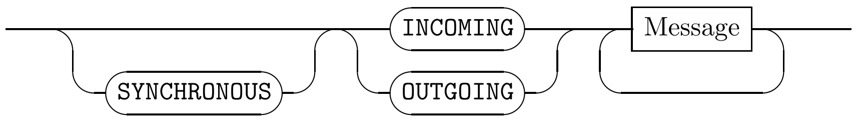
Port



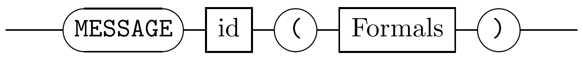
Protocol



MessageGroup



Message



3.1.2 Ports, Protocols and Messages

Ports are points of access to capsules. A `PORT` has an associated `PROTOCOL`, which is a special type, similar to a record but containing only messages as elements, describing the messages which can travel over that port.

Messages use the syntax of procedure signatures along with a keyword, `INCOMING` or `OUTGOING`, describing the direction in which the message is sent (from the point of view of the implementing capsule) and an optional keyword, `SYNCHRONOUS` which, if present, specifies that the transmission of the message is carried out immediately rather than queued as a request, the latter being the default case.

Synchronous messages correspond to procedures in the same way that asynchronous messages correspond to activities or transitions.

Synchronous message calls are distinguished by a leading `CALL` keyword as opposed to the `SEND` keyword used for asynchronous messages.

A protocol is a type which can be defined in one place - probably an INTERFACE - and used in several other places, just like any other type. It is also possible for a capsule to possess two ports of the same type.

3.1.3 Operations on Protocol Types : Conjugation and Aggregation

If two connected ports correspond to each other completely, then every `INCOMING` message of the one will correspond exactly to an `OUTGOING` message of the other, and vice versa. The protocol types of these two ports are then refered to as *conjugates.* Manual construction of such conjugated protocol types is repetetive and error prone, and violates the principle of defining everything once and only once. In such cases the conjugation operator `~` (tilde) should be used to produce a conjugate from the protocol type which it precedes.

As an example consider the following interface file which defines a protocol and its conjugate:

```
INTERFACE ConjTypes ;

TYPE aInbOut = PROTOCOL
  INCOMING MESSAGE a ();
  OUTGOING MESSAGE b ();
END;

TYPE aOutbIn = ~ aInbOut;

END ConjTypes.
```

This can now be imported and used in the definition of capsule specifications as follows:

```
CAPSULE INTERFACE ConjC1 ;
IMPORT ConjTypes AS CT;
PORT p1 : CT.aInbOut ;
END ConjC1.

CAPSULE INTERFACE ConjC2 ;
IMPORT ConjTypes AS CT;
PORT p1 : CT.aOutbIn ;
END ConjC2.
```

When instances of these two capsules are connected up, we know, by definition, that they must fit, giving us a picture as in Figure 9. It is now possible for us to add a new message to the protocol between these two capsules at the cost of a simple change to the interface ConjTypes. Of course this will not free us from the duty of implementing corresponding activities and send statements to make use of the protocol, but it does keep the description of the protocol itself in one place.

It may be the case that we want to reuse a given protocol type, but that we also need to introduce more messages. To achieve this we use the protocol *aggregation* operator `@`. This operator allows us to combine an arbitrary set of protocol types, anonymous or otherwise, to form a new protocol. The only limitation is that no message name may occur more than once.

Note that conjugation has a higher priority than aggregation. If we want to conjugate an aggregation with one operation we must use brackets.

If you make intensive use of these operators you must deal with the risk that the actual contents of a given protocol are not immediately visible to the reader of a model text at any given point. In practice, the capsule editor should be able to help you by telling you what is available at a given port, which connections use which messages and whether any messages may have been (intentionally or not) left unhandled or unused.

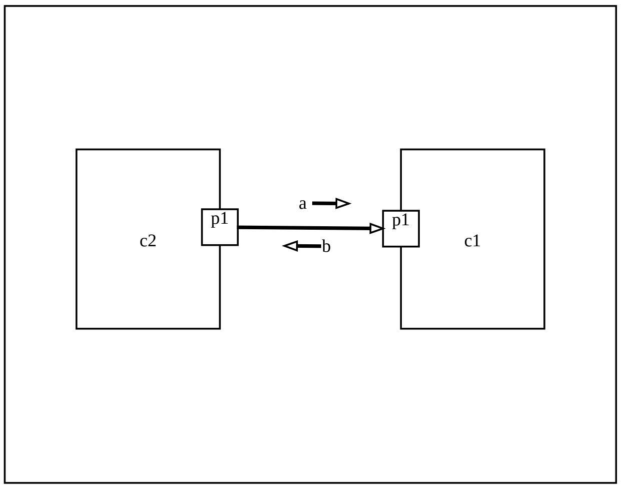
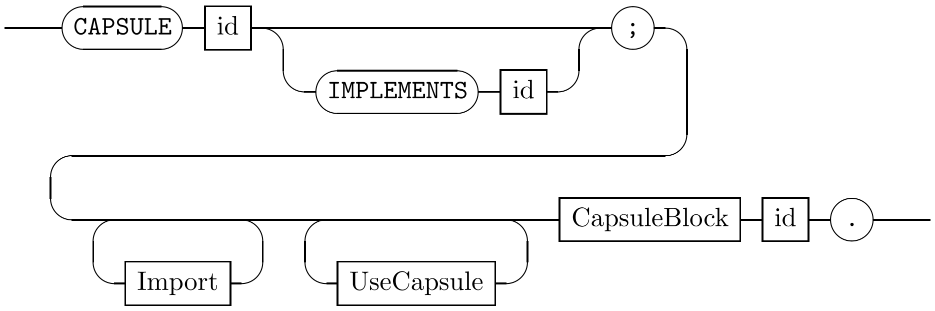


Figure 9: Capsules communicating over ports with conjugated protocol type

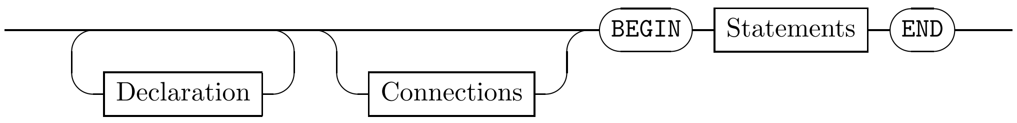
3.1.4 Capsules

The syntax is as follows:

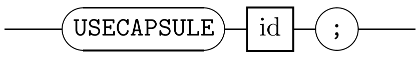
Capsule



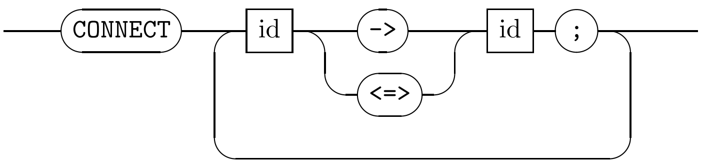
CapsuleBlock



UseCapsule



Connections



Anything that can be declared in a module can also be declared in a capsule. The difference is that the newly introduced types and declaration are *only* allowed in a capsule declaration.

The `USECAPSULE` statement introduces a second capsule into this capsule's namespace as a type. Instances of this type constitute child capsules of this capsule.

The `CONNECT` block uses the `->` notation to describe the *explicit* connections between

- Timers and their targets
- Triggers and their targets
- Parent and child messages
- Child messages of different children

The `<=>` notation is used to describe connections between ports as a whole. See Section 3.1.5 for details on connections in general and message routing in particular.

Statements placed in the BEGIN .. END section are executed when a capsule instance is created.

3.1.5 Message Routing

Routing in this context means: which activity (transitions are the same but we will just say activity from now on) is actually fired when a given `SEND` is executed. The simplest case of routing is both obvious and implicit, this is when an activity matches by name and signature to a message in the protocols of the specification. Here no explicit connection is needed. There are may however be cases where two messages of the same name are present in two different ports, where the message and the associated activity have different names, where a particular message in one port is connected to a particular message in another port, or where a child port is connected directly to a local activity. In these cases the connections must be specified explicitly. Finally, hopefully the standard case at least for capsules at the higher levels of a design, ports as a whole can be connected with each other. When such a connection exists, this means that all possible connections, based on identity of name and signature, are assumed. In combination with appropriate use of protocol type conjugation this can keep the textual descriptions for capsules brief.

It is obvious that an incoming message may be connected to more than one outgoing message, this corresponds to multiple use of a single resource. The opposite case is more controversial: can a single outgoing message be connected to more than one incoming message? The answer here is yes. The result is a kind of broadcast, with the originating SEND (which may be many capsules away and have no idea of the consequences of its actions) being replicated for each connected recipient.

3.1.6 Capsules, Types and Modules, similarities and differences

As a compilation entity a capsule has similarities to a Module. It is presented to the compiler in two files, an interface, describing the Ports and Messages available, and a body, containing the functionality of the capsule.

As a language entity a capsule has similarities to a Type in that many instantiations may be made from the same capsule. It differs from a type however - and could not be simply modeled as a Modula-3 object - because it can itself contain further type declarations.

3.1.7 Specifying and Using Alternative Capsule Implementations

So far we have seen capsules and their specifications always in pairs. On the other hand we have emphasised that, from the point of view of any containing capsule, only their specification is ever visible. We can exploit this consequence by defining capsules which share a single specification. By default, a given capsule matches a specification with the same name, however we can use the `IMPLEMENTS` feature we can specify some other specification, which may in turn be shared by several implementations.

This gives us a variation point. At design time, when we describe the contents of a given capsule in terms of child capsules and the connections between them, we only list the specifications of those child capsules. Only at run time do we need to commit ourselves to the actual bodies to be used.

The run time options to control implementation use are –gi (global implementation) and –ii (instance implementation). For global implementation we supply a comma separated list of capsule specification and capsule body name pairs.

```
runcap Cap1 --gi=Cap2=Cap3,Cap4=Cap5
```

The preconditions are that Cap2 and Cap4 are capsules which are used in the tree of capsules descending from Cap1, and that Cap3 and Cap5 have been declared to IMPLEMENT Cap3 and Cap4. For instance implementation, instead of the specification in general we supply individual instances using their hierarchical capsule names.

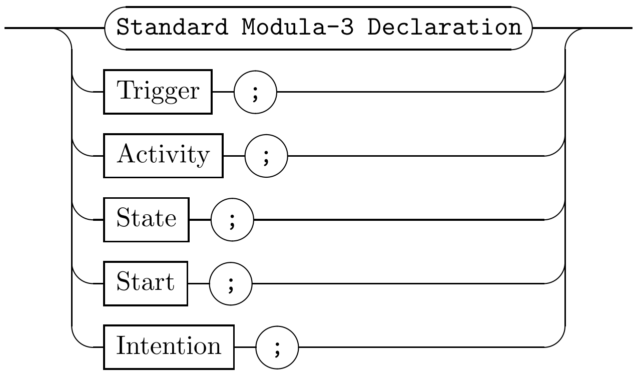
```
runcap Cap1 --gi=top.child1.grandchild=Cap3,top.child2=Cap5
```

Here the elements in the capsule names are the names of the capsules as defined within their respective parents. A conventional name of `top` is given to the top capsule.

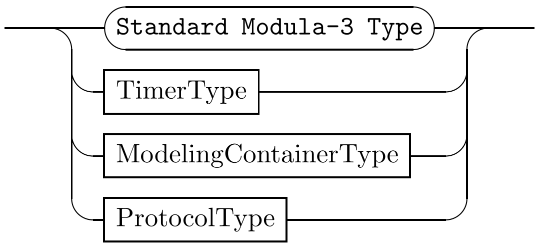
3.2 Types and Declarations

Several declarations, types and statements have been added to the language.

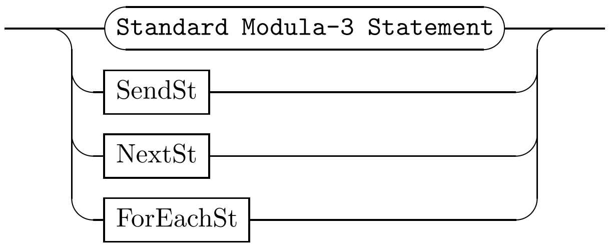
Declaration



Type

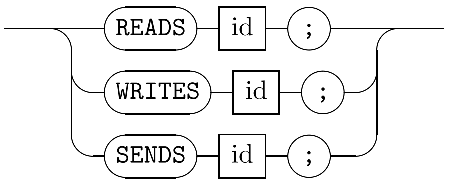


Statement



3.3 Intentions

Intention



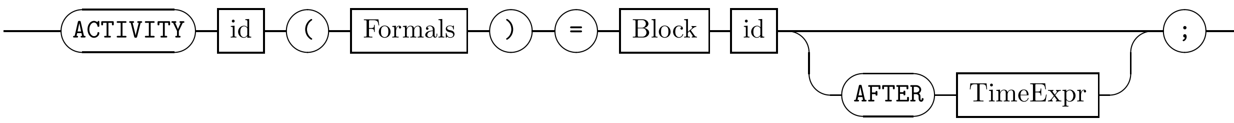
These declarations are added to activities and transitions by the capsule editor when the user creates a connection from one of those objects to some target datastore, message or port. They are regarded as general intentions which can only give a hint as to what the modeler wants to do. In this way they can be preserved in the model text but do not yet supply enough information for them to be executed.

We can also add such intentions textually ourselves. For example, if we know that two activities communicate over a given datastore x, then we can add `READS x` to the declarations of the one activity and `WRITES x` to the declarations of the other. The capsule editor renders this information with the appropriate arrows.

We can remove these intentions at the latest when we have written code which specifies this behaviour exactly in terms of assignments or specific `SEND` statements with actual parameters. This code is analysed by the model checker and the arrows are generated automatically.

3.4 Activities

Activity



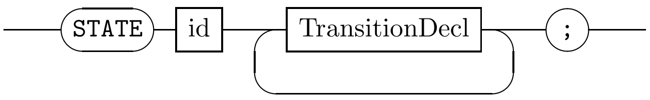
An activity is a collection of statements which are executed asynchronously as a result of a previously occurring `SEND` of a message connected, explicitly or otherwise, to that activity. The sender does *not* wait for the activity to complete. Use of the `AFTER` clause is described in Section 8

3.5 Datastores

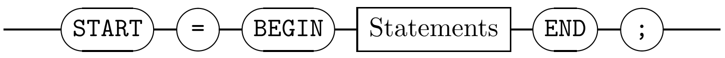
The datastore concept is not an extension to the Modula-3 language. All variables declared a the top level of a capsule (apart from timers and triggers) are visualised by the capsule editor as datastores.

3.6 States and Transitions

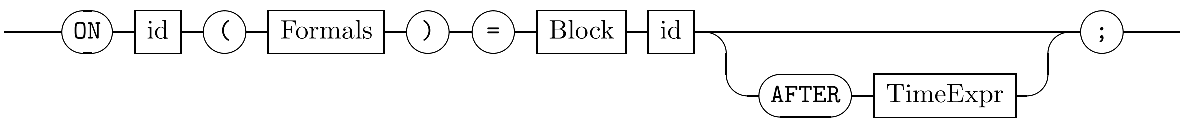
State



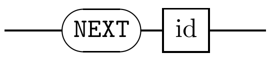
Start



TransitionDecl



NextSt



Capsules may contain state machines. State machines are organised primarily by state, and under each state are listed the transitions which are possible from that state. The state is the highest level syntactic unit, and it is theoretically possible to mix up states, activities and other declarations, but it is probably better if the states are grouped together.

Although transitions are defined at a lower syntactical level than activities, they are identical in every other respect except for the ability to select a new state using `NEXT`. They also share with activities the option for an `AFTER` clause, which is described in Section 8

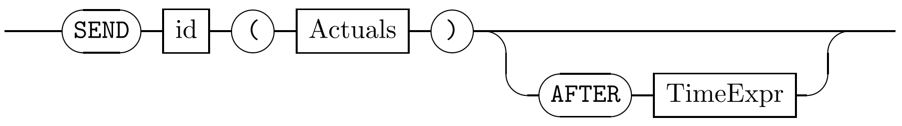
The state machine must have an initial state, this is defined by using the `START` pseudotransition and defining the initial state using `NEXT`.

If any transition has been defined to handle a given message, then that message is ignored if it arrives in any other state where no transition has been defined for it.

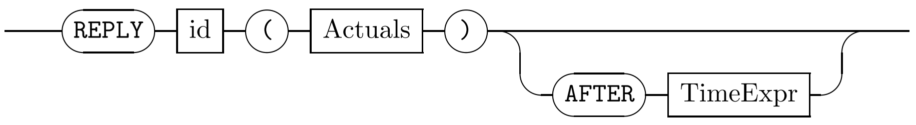
Graphically we have the problem that more than one transition handling the same message may be present in a diagram. If we want to show an internal connection (e.g. from a child capsule or a timer) using that message then we need a way to reference that group of transitions as a whole. This is done using a so-called transition proxy, a small graphical element which "catches" all references to the transition group. For an example of use see Figure 7.

3.7 The SEND, CALL and REPLY Statements

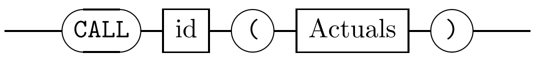
SendSt



SendSt



CallSt



The `SEND` and `CALL` keywords are used to indicate access to asynchronous and synchronous message interfaces, otherwise syntactically identical to procedure calls. For the use of `AFTER` see Section 8.

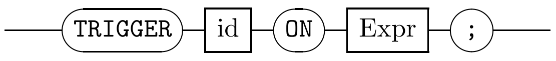
3.7.1 Callbacks using REPLY

In the normal case, the rules of message routing determine to whom we can send a message, or rather, who will receive a given message sent on a given port. There is however one particular case where this is not practicable. If a receiving activity wishes to reply to the sender of a given message, then it would normally do so using a `SEND` on a message which has been routed in the opposite direction. If, however the original message could have come from more than one sender, via multiple connections at some point on the inbound route, then the receiver will not know to which one it should reply. One alternative would be to broadcast a message in the reverse direction and let all potential senders work out which one was meant, but this is unlikely to be a general solution, and instead we can address this issue using callbacks.

This involves the original caller sending one of his own ports as a parameter in the first call. The receiver can then use a `REPLY` statement using any message available on this port to send a reply.

3.8 Triggers

Trigger

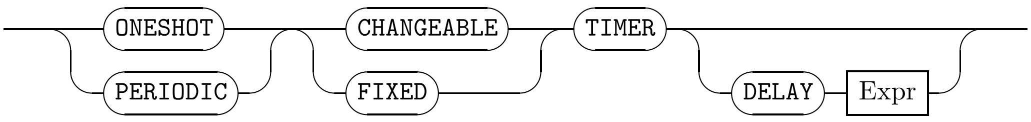


A trigger defines a signal *id* which is generated when the boolean expression *Expr* evaluates to TRUE for the first time or whenever it evaluates to TRUE having previously been false. The signal must be associated with an activity, transition or message using a CONNECT statement in order for it to have any effect.

Procedure calls returning values are also permitted in the *Expr* and these may make it easier to read and maintain, however you should avoid the use of side effects in these procedures. This is because the frequency with which the trigger is actually evaluated by the RTS (e.g. every simulator cycle, or only when the datastores mentioned in *Expr* are changed) is undefined.

3.9 Timers

TimerType



3.9.1 Timer Resolution

Timer resolution is measured in picoseconds, and values can be of arbitrary size. These values can be represented by literals of type `Timer.Time`, which uses the scaling features described in section 3.11.3 to define the type as follows:

```
TYPE Time = SCALED INTEGER {ps * 1000 = ns  * 1000 = us
                              * 1000 = ms  * 1000 = s
                              * 60   = min * 60   = hour
                              * 24   = day * 365  = year};
```

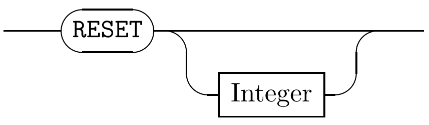
Note that in order for these literals to be used, the Timer interface must be imported, even if it is not being called explicitly to create objects of the `Timer.Time` type or to call procedures on timers offered by that interface. If you use the capsule editor, it will do this for you automatically.

3.9.2 Library Functions

- Start(T : TIMER) : starts the timer
- Stop(T : TIMER) : stops the timer
- Change(T : TIMER, newValue : INTEGER) : changes the timeout value of the timer. Has no effect on the current timeout if the timer is already running.
- GetElapsed() : INTEGER : returns the current elapsed time since the beginning of the simulation.

3.10 The RESET Statement

ResetStatement



Although our simulation is based on the concept of perfect technology, it is important that we can also model failures within the system so that we can investigate and predict their effects. This is supported by the `RESET` statement. This statements resets either the current capsule (the default case) or one of its ancestors, according to the optional supplied value (1 = parent, 2 = grandparent, etc.)

Resetting a capsule has the following effects

- delete all commands queued for that capsule its children
- delete all timers activated for that capsule and its children

3.11 Additions to the core language

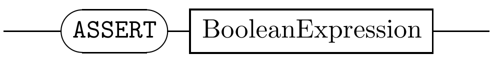
3.11.1 Predefined Exceptions

Existing implementations of Modula-3 did not support the handling of exceptions generated by the underlying runtime system. There was simply no way of referencing them in order to construct a handler. In this implementation the following exceptions are defined and can be referenced and handled by the user.

- ConstraintError : raised on attempt to violate array or subrange bounds.
- DivideByZeroError : raised on divide by zero.
- UninitialisedError : raised when an attempt is made to read a variable which has not been initialised.
- NullPointerError : raised on attempt to dereference a null pointer.
- AssertError : raised on failure of ASSERT (see below).

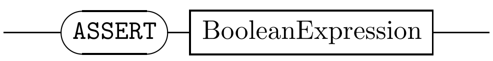
3.11.2 Assertions

AssertStatement



Using the `ASSERT` statement, expectations about the behaviour of the model can be included in the text without them being conflated with the algorithms of the model itself. The syntax is as follows:

AssertStatement



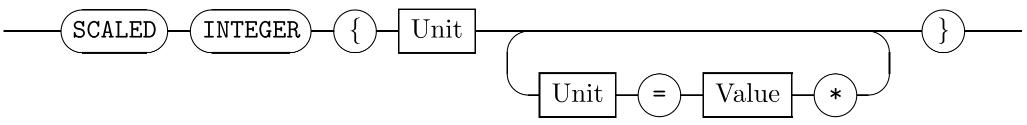
If the boolean expression evaluates to TRUE, then processing continues as normal. Otherwise the predefined exception `AssertError` is raised, unless the user invokes the model with the -a option, in which case an entry is made in the results and protocol files and processing continues.

3.11.3 Scaled types

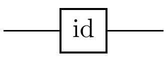
In engineering domains heavy use is made of integer types. Scaled types are provided to add additional meaning to such naked integer types without getting in the way. A Scaling is declared using the following syntax:

A scaling is declared as follows

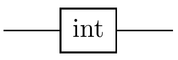
ScaledType



Unit



Value



For example:

```
INTERFACE ScaleTypes;

TYPE Volts = SCALED INTEGER {mV * 1000 = V * 1000 = kV * 1000 = MV};
TYPE Time = SCALED INTEGER {seconds * 60 = minutes *
  60 = hours * 24 = days * 365 = years};

END ScaleTypes.
```

Scaling declarations have the side effect of adding new user-defined capabilities to integer literal recognition. In the case of the types above this means we can now write `10 V`, a *scaled literal* which will produce the value 10000.

Assignment, addition and subtraction of values derived from differing scaled types will produce compiler warnings.

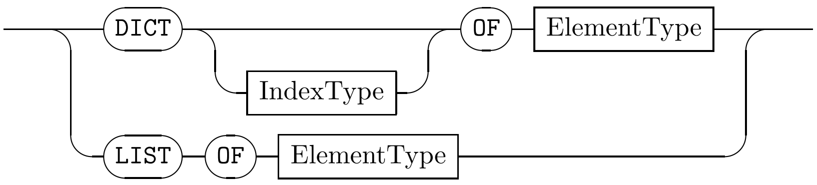
Scalings can also be used in input files (see Section 6.2 below. In this context they may only be used in combination with formal parameters for whose type the appropriate scaling has been declared.

Similary the predefined function IMAGE (Section 3.11.9 below) uses an available scaling to display an integer of a scaled type in a more readable manner. This is also used in the output of scaled values to results files.

3.11.4 Modeling Container Types and Iteration

Although Modula-3 provides a versatile and complete type system it leaves something to be desired in terms of the dynamic container types - lists and dictionaries - which we would expect to use when modeling a problem at a high level. Such data structures can be implemented in standard Modula-3 but their application in a type safe environment presupposes the construction of a `GENERIC` module for every new type. The two types offered here are an attempt to combine the expressiveness and convenience of python container types with the compile-time type safety typical of Modula-3.

ModelingContainerType



3.11.5 LIST

Lists are supported by the following built-in procedures:

- `APPEND(L,E)` : takes a list L and appends an element E to it
- `POP(L)` : removes and returns the last element of the list L
- `FIRST(L)` : returns the index of the first element (always 1)
- `LAST(L)` : returns the index of the last element (also corresponds to the length of the list)
- `DEL(L, index)` : deletes the item at the given index
- `INDEX(L, element)` : returns the position in the list of the the first matching element (raises ConstraintError if no element found)

Elements are accessed using the same notation as `ARRAY` objects. Note that the first element of the list always has an index of 1. Constructors for Lists are identical to those for Modula-3 ARRAYs, with the exception that the `..` feature is not supported (it makes no sense in the context of a list.) Membership of an object in a list can be tested using the `IN` operator in the same way as for `SET` types in standard Modula-3.

3.11.6 DICT

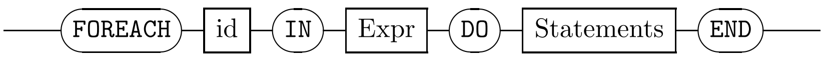
Dictionaries are accessed in a similar manner to lists and arrays but use an index of any type for which equality can reasonably be tested, including RECORDs, LISTs and DICTs. If none is specified then TEXT is assumed. Dicts are supported by the following built-in procedures:

- `KEYS(D)` : returns a List of `TEXT` values representing all the keys of the dictionary.
- `DEL(D,key)` Deletes the item with this key from the dictionary.

- `INDEX(L, element)` : returns the key in the list for an arbitrary matching element (raises ConstraintError if no element found)

3.11.7 Iteration using FOREACH

ForEachSt

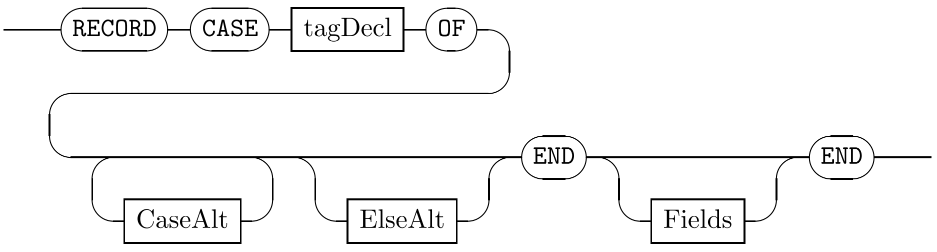


The `FOREACH` statement iterates over the items in a LIST or ARRAY. The loop variable is automatically created within the scope of the statement and successively has the value of each item. Using `KEYS` in the expression you can also iterate over the items in a dictionary.

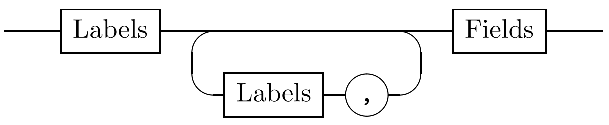
3.11.8 Variant Records

Standard Modula-3 does not provide variant records, on the basis that they "can be unsafe, and object types give you roughly the same benefits with absolute safety" [5, page 110]. However, we regard this construct as a valuable means of explicitly and declaratively modelling polymorphic data without having recourse to the indirections of objects and subclassing. The syntax is as follows (The `Labels` and `Fields` rules have been reused from the syntax for `CASE` and `RECORD` constructs respectively)

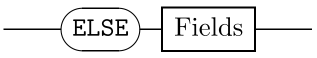
VariantRecordDecl



CaseAlt



ElseAlt



Constructors are the same as those for normal records, with the exception that the tag field is mandatory and that it must be statically defined (i.e. a literal or a constant.) In no sense do the alternatives overlay each other. The value of the tag is simply used to restrict access to the fields which are allowed for that value. Field names must be unique across the record as a whole. Images of variant records only show those fields which are valid for the current tag value. Here is a simple example:

```
INTERFACE Traffic;

TYPE
  vk = {car, truck, motorcycle};
  MetricWeight = SCALED INTEGER  {g * 1000 = kg * 1000 = tonne};
  vehicle = RECORD
    CASE kind : vk OF
    |  vk.truck =>
         axels : [2..10] := 2 ;
         weight : MetricWeight ;
    |  vk.car =>
         doors : [2..5] ;
    ELSE (* mandatory, shows we have not forgotten *)
    END;
    topSpeed : [30 .. 300];
  END;

VAR
  myCar   := vehicle{vk.car, doors := 3, topSpeed := 210};
  myTruck := vehicle{vk.truck, weight := 5 tonne};
  myBike  := vehicle{vk.motorcycle, topSpeed := 200};
END Traffic.
```

3.11.9 IMAGE

This additional predefined function converts arbitrary values into `TEXT` objects. Conversion rules are as follows:

- TEXT, CHAR, Integers, Floats : as expected
- Enums : the text of the enum value
- RECORD, DICT : "{" (key_id ":" IMAGE(value) ",")* "}"
- ARRAY, LIST : "[" (IMAGE(elt_value) ",")* "]"

3.11.10 PYTHON

The `PYTHON` statement allows the user to insert an arbitrary piece of python into the generated code. This is primarily of use in workarounds and tracking down of compiler bugs.

3.12 Un(der)implemented Features of Modula-3

It was a declared goal to implement Modula-3 in full, such that users can refer to any textbook on the language for guidance on how to code algorithmic parts of their models. This has been by and large achieved. There are however still some restrictions:

- Threads : All issues of concurrency are covered by the concepts of capsules, asynchronous message calls and run to completion. Threads are not and will never be supported.
- Objects : The current implementation of Objects is primitive, see tests for examples.
- Partial Revelation : some support, no cross-module type-checking.
- Persistence and pickling : No support.
- Type Equivalence and branding : No support
- Constants are not readonly and only simple numeric expressions are supported for constant values.

4 The Capsule Editor

4.1 Windows used by the capsule editor

If you just start the capsule editor using the `ce` without any parameters you will see two windows. One is the message window, in which all messages are displayed. The other is the central window, which contains a few buttons for some basic commands independent of any capsule. To do anything useful you need to open a capsule or two, which you can do either by using the "OPEN" button of the central window or by supplying the capsule name as a command line parameter when you invoke the tool. For each opened capsule, a source window and a graphical window is presented.

4.2 Graphical Editing

The most useful commands are hidden away in a menu which is popped up only when you press the right mouse button over any given graphical window. The commands are as follows.

The first group covers mode changes (see more on modes below)

- move : Go into move mode
- select : Go into select mode
- new object : Go into new object mode
- connect : Go into connect mode

- align : Go into align mode
- locate : Go into locate mode
- explain : Go into explain mode

The second group covers printing

- print : Generate scaled postscript and png of the complete design canvas (works on capsules of arbitrary size and complexity)
- gen scene : generate a povray scene file, only interestingif you have Povray installed on your computer and you want to impress people with three dimensional models

Finally we have some other functions

- open : Open a new or existing capsule
- save : save and generate code for current capsule
- undo : undo the last change to the model text (graphics-only changes not included)
- redo : redo what was undone
- redraw : reparse and redraw the current capsule
- reload : reload the source files of the current capsule (e.g. if they have been edited by another tool)
- test : dump a list of avatars to console (for internal test purposes)
- run : run the system with the default input file
- debug : start a debug run (see section on debugging below)
- options : change various options at run time (instead of using command line)
- input : edit and manage the input file used for a test run

4.3 Editing Modes

Amongst other things, the right mouse menu is used to change the current editing mode. The mode determines what happens when you press, drag or release the left mouse button.

The modes and their effects are as follows

- Move : The default mode, you can move objects by dragging them around the window or you can resize them if you click over a corner of an object while it is red

- Select : In this mode you can drag to create a box around items which you can then either move as a group in move mode or manipulate in other useful ways together in align mode
- Align : Acts on the currently selected set of objects and allows you to line them up, evenly distribute them or make them all the same size
- New Object : Create new objects, for each left click a menu of possible objects is shown, once a selection has been made a dialog is presented asking for the necessary details
- Locate : clicking on any graphical item will highlight its source in the text window
- Explain : selectively show and hide connections from all or selected avatars
- New Connection : Create new connections: simply drag from the source object to the target object, the tool will guess what you want to do

4.4 Working with source

The source window of the capsule editor provides a simple means of editing models at the source level. Once you have changed the code in one of these windows you will notice an asterisk appearing in the title field of the window to show that something needs to be done. A right button click in this editor brings up a menu allowing you to either `Accept`: commit your changes and recompile the capsule, or `Abort`: throw away your changes and revert to state before you edited the text.

4.5 Emacs integration

Although it is theoretically possible to do everything with the source editor, it is more likely that you already have a text editor with which you are more productive or can perform more complex operations. If you use emacs then you can put the following in your .emacs file.

```
(push (concat (getenv "M3_HOME") "/bin") load-path)
(require 'modula3)
(require 'reload)
(setq auto-mode-alist
   (cons '("\\.m3$" . modula-3-mode) auto-mode-alist))
(setq auto-mode-alist
   (cons '("\\.i3$" . modula-3-mode) auto-mode-alist))
(setq m3::abbrev-enabled t)
```

This does two things. First it loads an augmented modula-3 mode so that you will get syntax highlighting of your model text. Secondly it makes a simple communication mechanism available by which emacs can talk to the capsule editor and tell it to reload text which you have edited "behind its back" This is bound to the Alt-R key by default. Another useful habit to get into is to switch on `auto-revert-mode` for those files which you have opened in emacs and which you are nonetheless editing with the capsule editor. Note that you can get very confused and will probably lose work if you chop and change between two editors without being careful to save your changes in between.

4.6 Compilation

Every time you add an object graphically or accept changes in the text editor, the capsule is recompiled. Any errors are shown in the message window in red - so you are unlikely to miss them. A click on the error text will cause the offending line of source code to be highlighted in its text editor.

4.7 Graphical Debugging

Providing that your capsule has compiled without errors and you have prepared an `.inp` file of the same name as your capsule, you can start a simulation run by selecting the `debug` right-button entry and step through it using the `next` button of the central window. At each step the currently running activity is shown highlighted in red, and the contents of datastores are also shown. Previously visited activites are highlighted in green. Any child capsules called during the simulation run are opened by the editor if necessary.
[1]

5 Tool Invocation

The toolset consists of a number of scripts which are invoked from the command line. Options of interest to the user are described in the following sections. For a more complete list (including obscure developer options) invoke any of the tools with the `-h` option.

5.1 m3 - the Compiler

```
  -h, --help            show this help message and exit
  -c, --track-constant-expressions
                        track constant expression calculation
  -e, --stop-on-error   stop on error (rather than just complain and go on)
  -lLIBRARY, --library=LIBRARY
                        library to store object files and generated code
  -m, --no-summary-messages
                        suppress summary messages (restricts output on
                        regression tests)
```

[1]This currently has a problem debugging several instances of the same capsule in a running program.

```
  -p, --no-python       do not generate python code
  -s, --syntax-only     quit before naming pass
  -SSCRIPT_OUTPUT, --script-output=SCRIPT_OUTPUT
                        where to put script output (default SessionScript.py

  -vVERBOSITY, --verbose=VERBOSITY
                        Track syntax checking over rules 0=none;1=matched
                        symbols;2=all rules
  -w, --no-warnings     suppress warnings
  -X, --raise-unhandled-exceptions
                        raise unhandled exceptions rather than transforming them
                        to M3 world
```

5.2 ce - the Capsule Editor

```
  -g, --generate-scene  generate POVRAY scene and quit
  -G, --ignore-geometry
                        ignore Window geometry resources
  -H, --hide-ports      Do not show messages in ports (useful for domain models
                        containing activities only)
  -I, --no-implicit-connections
                        do not render implicit connections
  -R, --ignore-resources
                        ignore all resources (use random positions)
  --screenshot          generate a screenshot of the capsule(s) and quit
```

5.3 runcap - the Capsule Launcher

```
usage:
usage: runcap.py [options] model [testscenario [outputfilebasename]]



options:
  -h, --help            show this help message and exit
  -a, --ignore-assertfailures
                        Do not raise exception when assert fails
  -cTXTSRC, --console-output=TXTSRC
                        output protocol (pro) or results (res) to console
  -d, --debug           enter debugger
  -D, --dump-connectors
                        dump connector tables
  --debug-port=DEBUGPORT
                        port for graphical debug messages to server
  --gi=GLOBALIMPLEMENT  set global implementations for capsule interfaces
  --ii=INSTANCEIMPLEMENT
                        set instance implementations for capsule interfaces
```

```
-i, --trace-invocations
                      trace all activity and transition invocations
-p, --preprocess-only
                      stop after preprocessing
-P, --profile         generate profile information
-sSTATISTICS, --statistics=STATISTICS
                      dump statistics in protocol
                      0=none;1=basic;2=detailed;3=gory
-rRESULTS, --results=RESULTS
                      path for results subdirectory (defaults to res)
--raw-python          use raw python types
-t, --trace           dump trace buffer
```

6 RTS and Testing Issues

6.1 Toolset Dataflow Overview

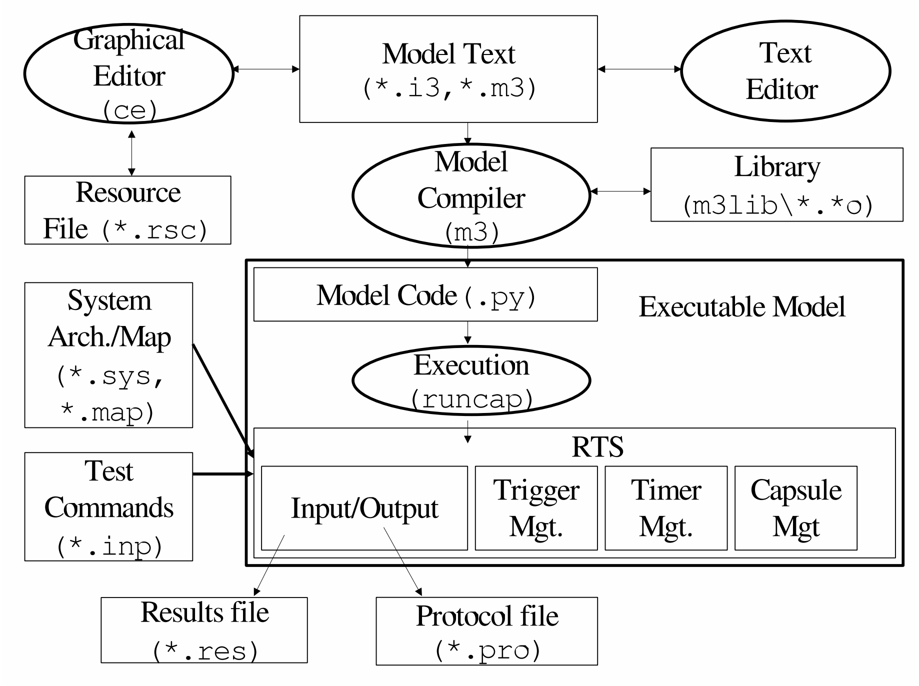


Figure 10: Files used in the course of modeling

The following file types are used during the modeling process:

- `*.rsc` : Resource files describing the position of modeling objects within a diagram. If no such file can be found and a model is loaded into the capsule editor it will make no attempt to make things "pretty", but places the objects randomly and relies on the user to position them in a readable manner

- `*.i3,*.m3` : Model source files.
- `m3lib/*.*o` : Internal, intermediate compiler files in xml format, these are all stashed away in the library where they won't disturb you
- `m3lib/*.py` : Executable results of compilation. Not for direct use. These should be accessed via the `runcap` command only.
- `*.inp` : Test input files - described in Section 6.2.
- `*.sys` : System Architecture Description files - described in Section 8.4.
- `*.map` : Processor Capsule Mapping files - described in Section 8.5.
- `*.res,*.pro` : Test result files - described in Section 6.3.

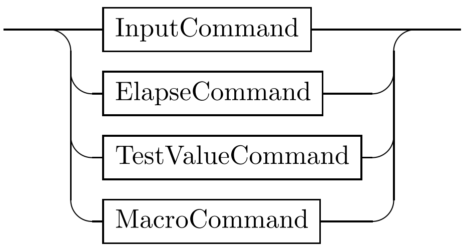
6.2 The Input File Sublanguage

6.2.1 Lexical issues

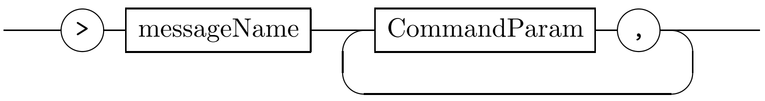
An input file is a series of commands to the simulator. Each command is on a new line. Commands can be spread across several lines using a trailing backslash. Comments are preceded with a `%` sign and are effective to the end of the line.

6.2.2 Syntax

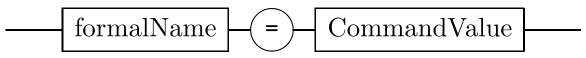
Command



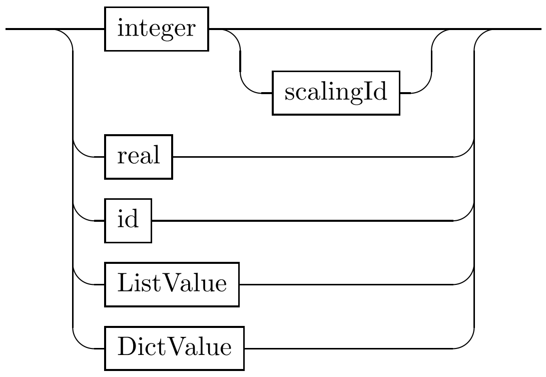
InputCommand



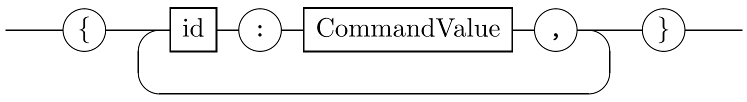
CommandParam



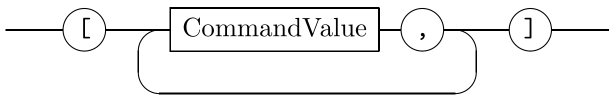
CommandValue



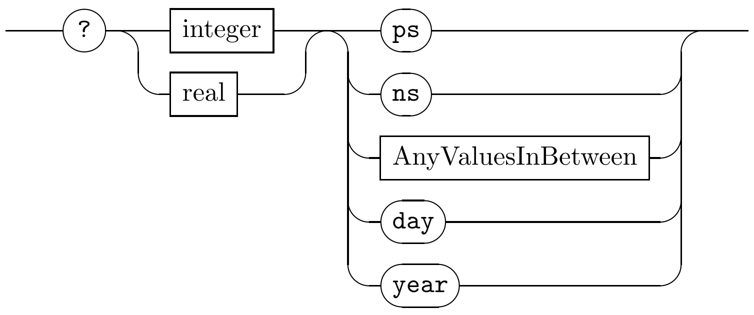
DictValue



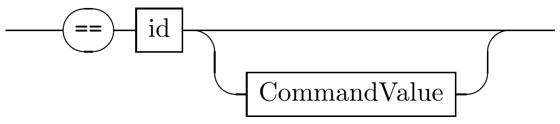
ListValue



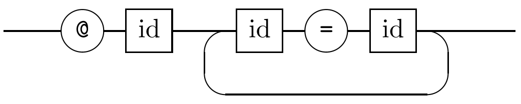
ElapseCommand



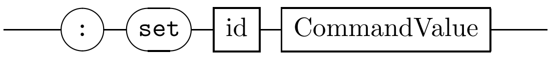
TestValueCommand



MacroCommand



DataPortCommand



6.2.3 Message Input

As shown in the syntax above, an input command consists of a `>` sign followed by a message name and and any number of parameters each of which is a pair consisting of the formal parameter name and the actual parameter value.

Parameter values can be written for any internal Modula-3 types other than references. Integers and reals map as expected. Id values map to objects of type `TEXT` and to enumerated types. ListValues map to ARRAYs and LISTs and DictValues map to RECORDs and DICTs. The mapping is the exact inverse of that used for message output, which is identical to the predefined IMAGE function (see section 3.11.9 for details). See section 6.4 below however for a powerful extension to support patterns in expected values.

If a value can not be mapped to the type of the corresponding model parameter then an exception is raised and the model is not executed.

For example suppose we have a model whose top level capsule is as follows ...

```
CAPSULE INTERFACE TestInputTypes ;
IMPORT InputTypes;
PORT p1 : PROTOCOL
    INCOMING MESSAGE testInput(myRec : InputTypes.MyRecType) ;
END;
END TestInputTypes.
```

... and type definitions used are as follows ...

```
INTERFACE InputTypes ;
     TYPE Color = {red, blue, green} ;
     TYPE Vector = ARRAY OF INTEGER ;
     TYPE MyRecType = RECORD
        color : Color ;
        vector : Vector ;
     END ;
END InputTypes.
```

... then the following input command will feed valid data to the model

```
> testInput myRec = {color: red, vector: [6,3,1]}
```

6.2.4 Simulating Time Elapse

Time is caused to elapse using a command consisting of a `?` sign followed by a value and a factor. The valid factor identifiers are the same as those of the constants in the `Timer` library interface. See Section 3.9.1 above for details. To cause an elapse of 10 millisecond, write

```
? 10 ms
```

To cause an elapse of two and a half hours, write

```
? 2.5 hour
```

6.2.5 Dataports and TestValue Commands

Data items within the model can be written directly using a command from the input file. Value mappings are as described above under Message Input. Data items at an arbitrary depth in the runtime capsule tree can be adressed using dotted notation. To set an INTEGER `x` in your top level capsule to the value 99, write

```
: set x 99
```

To set an object `rec` of type `RECORD b : BOOLEAN ; i : INTEGER END` in the child capsule b of the child capsule a of the top capsule, write

```
: set a.b.rec {b: TRUE, i: 99}
```

Similarly the values of data items can be read directly from the model using the test value command (`==`). Execution of this command creates an entry in the result and protocol files which includes an image of the current value of the data item supplied as the first parameter. The second parameter, if present, is tested for equality against that value. See section 6.4 below for more on this technique.

6.2.6 Using Macros

Some sequences may be repeated several times with small changes in one test sequence, or (for example in a setup sequence) reused over many separate test sequences. For this purpose a simple macro facility is provided.

When a macro is encountered using the `@` sign, the file of that name is read, parameter substitution is performed and the resulting text is inserted into the the command parser. Parameter substitution is done using any key value pairs which supplied with the macro invocation. For each pair with the form `formal=actual`, instances of the pattern `$(formal)` are replaced with `actual`. [2]

[2] The only values currently supported are integers and strings. Contact the author if you have further requirements

Currently the macros files are expected in the same directory as the file using them.

For example we can package the whole business of inserting our card, inserting our pin and requesting money into one macro, which we store in the file `withdraw.mac` as follows:

```
> insertCard card = {encodedPIN: 1234}
> inputPIN pin = 1234
> inputSum sum = $(sum)
```

and then we can write a test in which we make a deposit, two withdrawals and then wait a year.

```
> deposit sum=100
@withdraw sum=4
@withdraw sum=6
? 1 year
```

Remember that this bank has generous interest rates

6.3 Input, Result and Protocol Files

A test run produces at least two files as output, one with the extension `.res` and one `.pro`. See section 6.4 below for a further file used in case of failures. The basename of the output files is by default the basename of the `.inp` file, which by default is the basename of the model itself. Thus the invocation:

```
runcap Account
```

will expect that we have at some time compiled a Capsule named Account and that we have prepared a file of test inputs called `Account.inp`. It will produce test outputs in `Account.res` and `Account.pro`.

The following command

```
runcap Account WithdrawTest
```

... will expect the test input in called `WithdrawTest.inp` and will produce test outputs in `WithdrawTest.res` and `WithdrawTest.pro`, and finally this command

```
runcap Account WithdrawTest SpecialResults
```

will leave test outputs in `SpecialResults.res` and `SpecialResults.pro`.

So what is the difference between a result and a protocol file?

A Result file describes only the interactions of the model with its environment. It shows nothing more than the inputs and outputs and the simulation time at which they occurred. As such it is suited for regression testing of single models or for comparing the results of different models to see if their behaviour is identical. See the section on Feedback Testing below for the details. Each line in a result file takes one of the following forms

`>` **input-message-name message-parameters** a test input message

`<` **output-message-name message-parameters** a model output to the environment

`?` **time-value** an elapse request from input file

`!` **time-value** notification of elapsed time in the simulation

A Protocol file is a more detailed human-readable account of the test run. It contains adds the following information

- A header describing the circumstances when the test was run, by whom and which files were used
- Commands are listed in the form they were presented in the test input file, including any comments
- Trigger events, denoted with a `*`
- Output from the application layer of the model, accessible to the user via the Results interface (see Section 9.1)
- A summary of the test run giving a configurable depth of detail (use the -s option)

6.4 Feedback Testing : Advanced Input Files

As described above, input files can be used simply to specify inputs for the system and the elapse of time between those inputs. It is however also possible to specify expected behaviour in an input file. Both the expected time and expected values for individual outputs can be specified. If such expectations are stated but not met then the test run is regarded as having failed, corresponding comments are written into the results and protocol files, and a new file is created with the ending `.fail`

This feature reuses the syntax of the results file in such a way that, for a given model, the results file can be fed back in to that model as an input file. In the context of an input file, the output (`<`) and elapsed (`!`) tokens take on new meanings.

`!` Set the base time for all subsequent expectations to the given absolute time value

`<` Expect the following message with the following parameters at the currently set base time, or within some window surrounding it

`==` Check that the following datastore reference has the given value

The idea is that the lazy modeller, once he has specified the inputs and their timing, can let the model write his test file for him. If he is very lazy he will just copy the res file over the original inp file and carry on. [3] As he works with the model and refines it, it is more likely that he will want to loosen the absolute constraints of the original res file in the hope of creating a test suite which is relevant for models at various stages of detail and completeness, and this is what we look at next.

6.4.1 Expected Values

There are cases where we may not want to specify the expected value at all, just that we do expect one, this can be done by using an asterisk instead of a value with the given parameter. Note that it is not considered a failure if extra, unspecified parameters are sent by the model. The intention is that simple early tests should still be able to work on mature complicated models. The real power of the asterisk lies in the ability to embed it within values. For example if we originally receive the following message from the model

```
< fire x = [ 3, 1, 2, 0 ]
```

but we know that the second element in the list or array which is sent may change over time, then we can use the asterisk to loosen only this constraint, and place the following in our input file

```
< fire x = [ 3, *, 2, 0 ]
```

There are other cases where we may not be sure of the exact value to expect, but know that it will either lie within a given range (if it is a scalar type) or that it will assume the value of one of a set of alternatives. For these cases we need some extra syntax. If we know that the third value in the list will lie between 2 and 5 we specify the range of expected values as follows:

```
< fire x = [ 3, *, [[2 .. 5]], 0 ]
```

This obviously only makes sense for scalars. Finally there is the possibility to specify a set of disjoint values, one of which we expect. For example if we know that the 4th element is always either 0, 2 or 4, we write the following

```
< fire x = [ 3, *, [[2 .. 5]], [[0; 2; 4]] ]
```

The notation can be used at any level, using and within values of arbitrary complexity.

[3]There is a limitation with respect to macros, because the preprocessor deals with them first and because they are essentially invisible to the test input processing mechanism. Macros need to be dealt with by hand. This means, if you have used macros in your input file you really need to split your result file back up into the corresponding pieces so that you retain the reuse benefit of your macros.

6.4.2 Expected Times

By default, an expected output must occur at precisely the currently set base time. If this is too strict, an expected output can be given a time window in which it may occur relative to the currently set base time.

For example if we want to specify that the signal `fire` with a parameter x of unknown value is expected at any point within the 10 ms previous to the current base time we write the following:

```
< fire [ 10 ms, 0 ms ] x = *
```

If we just want to say that it can occur at any point after the current base time, but now know that the value of x should be exactly 99, we write the following:

```
< fire [ 0 ms, * ] x = 99
```

6.5 Debugging

Invoking the system with the `-d` option will open a simple debugger with which you can step from one event to the next and look at the time and command queues. Graphical debugging (see section 4.7)is available from the capsule editor.

7 Simulation Concepts

7.1 Time

Following the tenets of perfect technology as discussed in [3] and [2], initial SESIS models run in a VPS or Virtual Perfect System. This system has unlimited memory and performs all operations in zero time. Using a contemporary desktop computer rather than our final embedded target frees us from any practical concerns about memory, but what about time? Here it is not adequate to simply say that our simulation machine is faster than our target machine. Although internal operations are performed in zero time, time itself still exists in the outside world and the representation of these external time intervals is vital to the construction of any meaningful model for the realtime, embedded domain.

For a description of timers and how to declare them, consult section 3.9. For information on how to control the elapse of time using test input files consult section 6.2.4. For information on how to test the time behaviour of your model by stating your expectations in the test input file consult section 6.4.

7.2 Run To Completion

This concept can be understood on two levels; the first, simplest and most common level is that activities or transitions are never interrupted or interleaved but happen one after another. Concretely this comes into effect when the execution of a `SEND` statement leads to the corresponding message being placed on a queue for later processing. The activity performing the `SEND` must finish first. If we combine this with the absence of any mechanisms for implicit parallelism in the simulation environment then this kind of run to completion is practically unavoidable.

There is however another level. Here we regard "completion" as the completion not only of the activity itself, but also the completion of the local consequences of its `SEND` statements. This becomes relevant in models with several layers of capsules. In such cases we want to know that the dispatch of a message to a given capsule at a given layer will not result in any messages for that capsule being "left in the queue" afterwards. This means we can change the internal organization of a capsule at any level confident in the knowledge that, if its direct behaviour towards its environment in terms of messages and timing is unchanged, our system is guaranteed to perform as previously.

An example for this behaviour is available in the `test` subdirectory in the CapRTC* family of files. In Figures 11 to 13 we depict a nested set of capsules which form the following tree.

```
CapRTC1
    CapRTC21
    CapRTC22
        CapRTC221
        CapRTC222
```

The main activity in CapRTC1 sends three messages, s1 to CapRTC221, s2 to CapRTC222 and finally s3 to its own capsule. On receipt of s1, the activity in CapRTC221 sends s4 to CapRTC222.

The question is: in what order should the dispatched messages result in an activity being called in their respective destinations?

The answer is as follows:

1. s3, although the last to be sent by the main activity, is the most local and therefore has the highest priority, so it is scheduled first.
2. s1 is next: the targets of s1 and s2 are both at the same level, so by default the first to be dispatched is the first to be scheduled.
3. while the recipient of s1 is running, it dispatches s4 to its sibling capsule. This is the most local outstanding request so s4 is scheduled next.
4. s2 comes last.

8 Parallelism, System Architecture and Deployment

Parallelism and perfect technology are rather uneasy partners. If our technology is truly perfect we should have no need for parallelism at the computational

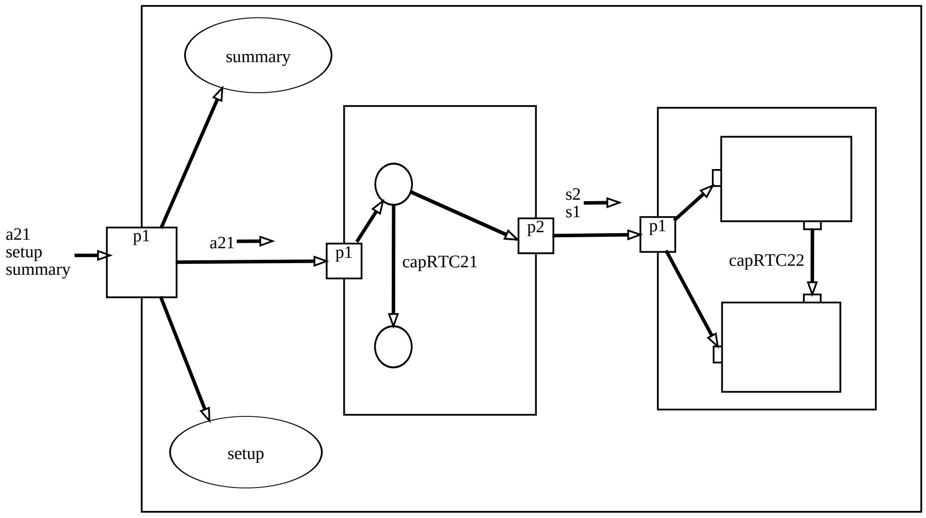


Figure 11: Run To Completion Scenario: Top Level

level. However, if parallelism is a potential aspect of our implementation, at some stage we will want to model this as well.

Up till now we have seen time as being actively influenced by the outside world only; practically this involves using a delay statement in a test script. The approach we add now is based on the notion of time as a resource which can be consumed internally. Note that we are only concerned with true physical parallelism here, rather than thread or process based pseudo-parallelism.

8.1 Time weighted execution

Our term for a unit of parallelism is a *processor*. Within a model where parallelism is being considered, any given capsule will be allocated to a certain processor. Unless otherwise defined this means that all the children of that capsule will also be allocated to the processor. While time is being consumed internally by a given processor, that processor is blocked and none of the allocated capsules can perform any operations. By default we simply have one processor to which all capsules are allocated. It remains for us to specify the consumption of time. This is done using two new language features:

- Delayed SEND : we append an `AFTER` *time* annotation to a `SEND` statement
- Delayed completion : we append an `AFTER` *time* annotation to the `END` of an `ACTIVITY` or `TRANSITION`

A delayed `SEND` has two effects. Firstly the message is only sent when the specified amount of time has elapsed. Secondly the owning processor of the

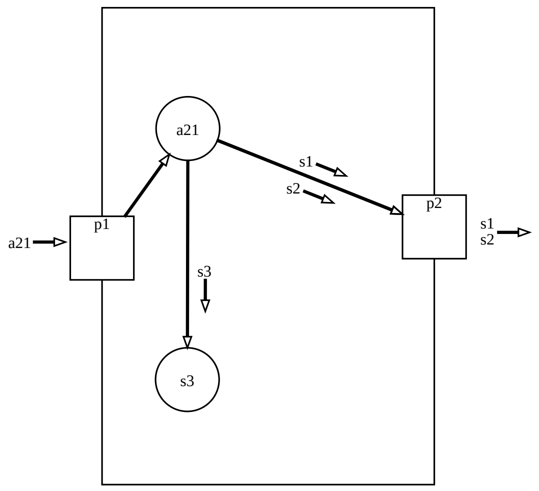


Figure 12: Run To Completion Scenario: Left Hand Child

sending capsule is blocked for at least the time. Delayed completion simply guarantees that the sending capsule is blocked for at least the time specified. These two annotations can be combined, in which case the actual blocking time for the capsule is the maximum of all delayed `SEND`s and the delayed completion.

Should any of the `SEND`s consume less time than this calculated maximum then they will be dispatched earlier. Of course this will only be of relevance if their targets are allocated to another processor.

8.2 A Single Processor Example

Consider the following code:

```
CAPSULE INTERFACE SimplePar ;
PORT p1 : PROTOCOL
 INCOMING MESSAGE getBusy() ;
 OUTGOING MESSAGE delegate() ;
END;
END SimplePar.

CAPSULE SimplePar ;
  ACTIVITY getBusy () =
  BEGIN
     complex1();
     SEND delegate() AFTER 5 s;
     complex2()
  END getBusy AFTER 10 s;
  PROCEDURE complex1 () =
```

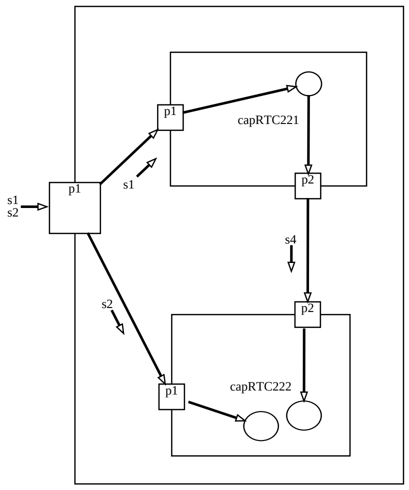


Figure 13: Run To Completion Scenario: Right Hand Child

```
  BEGIN
  END complex1 ;
  PROCEDURE complex2 () =
  BEGIN
  END complex2 ;
BEGIN
END SimplePar.
```

This capsule, on receiving the message `getBusy`, must first do some work itself, before it can send the `delegate` message, which we here describe as being sent 5 seconds after the activity has started. After sending the signal, it must do more 5 more seconds of work and so as a whole it blocks the processor (we only have one of them so far) for a total of 10 seconds.

If we test this capsule with the following script:

```
> getBusy
?  6 s
> getBusy
? 100 s
```

we are rewarded with the following results:

Here we can see that the first call to `delegate` is sent out after 5 seconds, while the activity, and the capsule as a whole, is blocked for 10 seconds, such that the subsequent `getBusy` call from the test file is delayed for that time, and the subsequent `delegate` is sent at 15 seconds.

8.3 Multiple Processors

As described above, by default all capsules belong to one default processor. If we wish to partition our system across processors, the simplest way to do this is via the command line with the -p argument. To demonstrate this we need to add at least another capsule. In this case we match the capsule defined previously with a receiver for the delegate message and instantiate them both within a top level, depicted in Figure 14:

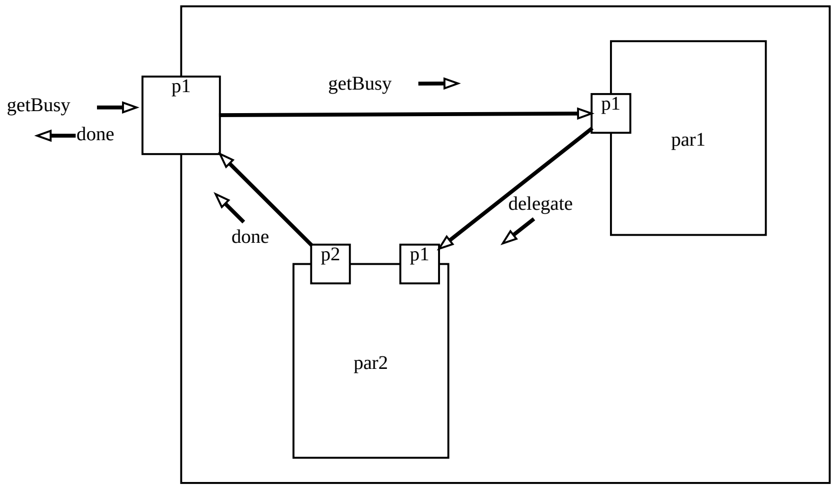


Figure 14: Top Level View of simple parallelism example

And here is the partner capsule

```
CAPSULE INTERFACE SimplePar2 ;
PORT p1 : PROTOCOL
 INCOMING MESSAGE delegate() ;
END;
PORT p2 : PROTOCOL
 OUTGOING MESSAGE done() ;
END
END SimplePar2.

CAPSULE SimplePar2 ;
IMPORT Timer;
  ACTIVITY delegate () =
  BEGIN
     SEND done() AFTER 3 s;
  END delegate AFTER 6 s;
BEGIN
```

```
END SimplePar2.
```

We can now run this new top capsule with the same input as before and with only one processor to get the following results:

However when we divide the two capsules over two processors, `proca` and `procb` (using the -r parameter so that our results end up in a new subdirectory and do not overwrite those from the previous run)

```
runcap ParTop --processors=par1=proca,par2=procb --results=par
```

... then we see the following results:

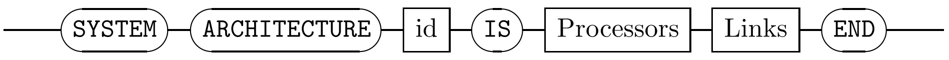
```
> getBusy
?  6 s
!  6 s
> getBusy
? 100 s
!  8 s
< p1.done
!  18 s
< p1.done
```

It is left as an exercise for the reader to convince himself that the timing figures are as should be expected.

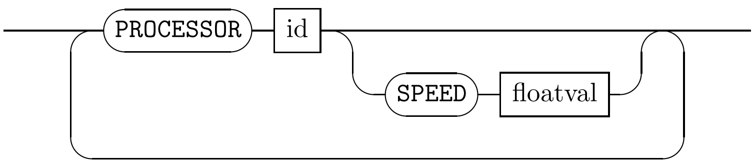
8.4 Defining System Architecture

While the previous technique might be useful for experimentation, another approach is available for permanent and detailed documentation of *system architecture* in terms of processors, their relative performance and the speed of the links between them. These features are described in extra descriptor files using a special purpose language with the following syntax:

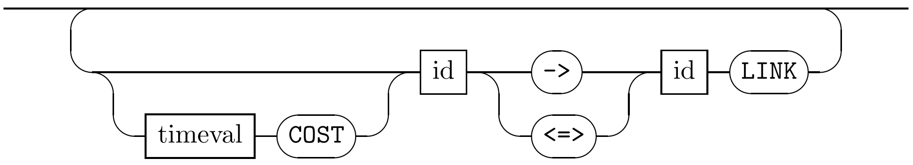
SystemArchitecture



Processors



Links



In this description we record not only the existence of processors but also:

- Processor speed : this is a floating point factor by which any time values used in `AFTER` clauses by activities of capsules allocated to that processor is divided.
- Link costs between processors. This cost, given as a scaled time expression is added to the delay time of any `SEND` between the two given processors. If the link is symmetrical in terms of cost we can use the `<=>` notation to cover both directions. Attempts to send between processors for which no link has been defined will cause a run time error.

Here is an example which we will apply to our previous example:

```
SYSTEM ARCHITECTURE Arch IS
  PROCESSOR proca SPEED 1.0;
  PROCESSOR procb SPEED 2.0;
  LINK proca -> procb COST  2 s;
END
```

8.5 System Mapping

One way to use this definition, stored in this case in the file `Arch.sys` is to reference it at run time using the `--system-architecture` option, and then use the defined processor identifiers with the `--processors` option as follows

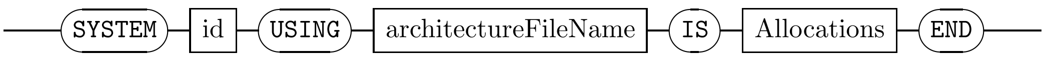
```
   runcap ParTop --processors=par1=proca,par2=procb --system-architecture=Arch
--results=parsys

> getBusy
?  6 s
!  6 s
> getBusy
? 100 s
!  8 s + 500 ms
< p1.done
!  18 s + 500 ms
< p1.done
```

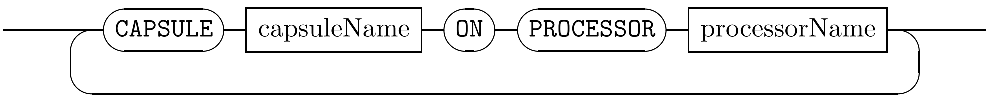
Here we notice that the first `done` message now comes back in 8.5 seconds. This breaks down into 5 seconds for the work done by par1, 2 seconds for the link cost and 1.5 seconds for the original 3 seconds reduced by the 2.0 speed factor.

Finally, if we want to permanently document both our System Architecture and the mapping of the capsules in our design to that architecture, we can do so in a separate mapping file. This uses the following syntax:

Mapping



Allocations



If the following is in the file `Sys1.map`

```
SYSTEM Sys1 USING Arch IS
   CAPSULE par1 ON PROCESSOR proca;
   CAPSULE par2 ON PROCESSOR procb;
END
```

then the following, supplying the mapping file alone, will have the same effect as the previous invocation

```
   runcap ParTop --system-mapping=Sys1
```

8.6 Restrictions

Synchronous calls across processor boundaries are not detected and may therefore produce misleading results.

9 The Library

Modula-3 has a substantial library of interfaces. Most of these however are either of no relevance to the modelling/embedded domain or are superceded by the more powerful modeling types included in the language extensions documented here. The following are presented either to give a starting point for users coming from Modula-3 (Text, IO and Fmt) or to provide access to SESIS run-time resources (Results and Timer).

9.1 Results

This offers an interface from the application layer of the model to the results and protocol files

```
INTERFACE Results;

PROCEDURE Write(msg : TEXT);
(* Write msg to results file and protocol file *)

PROCEDURE WriteProtocol(msg : TEXT);
(* Write msg to protocol file only *)

END Results.
```

9.2 Timer

```
INTERFACE Timer;


TYPE Time = SCALED INTEGER {ps * 1000 = ns  * 1000 = us
                              * 1000 = ms  * 1000 = s
                              * 60   = min * 60   = hour
                              * 24   = day * 365  = year};

PROCEDURE Start(VAR T : TIMER);
PROCEDURE Stop(VAR T : TIMER);
PROCEDURE Change(VAR T : TIMER; newValue : Time);
PROCEDURE GetElapsed() : Time;
END Timer.
```

9.3 IO

```
INTERFACE IO;

(* The following perform simple output to the terminal, for testing
   purposes it is recommended that you use the features provided by
   the Results Interface *)

PROCEDURE Put(T:TEXT);

PROCEDURE PutInt(I:INTEGER);

PROCEDURE PutReal(f:REAL);

PROCEDURE PutChar(C : CHAR);

END IO.
```

9.4 Fmt

```
INTERFACE Fmt;

(* Converts basic data types into text, for use with the IO package
   Present to provide rudimentary compatibility with standard Modula-3
   It is recommended that you use the more powerful new built-in IMAGE
   procedure instead *)

PROCEDURE Bool (b: BOOLEAN) : TEXT;

PROCEDURE Char (c : CHAR) : TEXT;

(*TYPE Base = [2..16];*)

PROCEDURE Int (n : INTEGER) : TEXT ;

END Fmt.
```

9.5 Text

```
INTERFACE Text;

(* Basic operations on TEXT type *)

TYPE T = TEXT;

PROCEDURE Equal(t,u : T) : BOOLEAN;

PROCEDURE FromChars (READONLY a : ARRAY OF CHAR) : T ;

END Text.
```

10 C Code Generation (highly experimental)

During the normal course of modelling, the model is compiled into python code and tested using the test framework. An alternative path, useful at a later stage of development, translates models written in a subset of SML into C code which can be compiled for and run on the target hardware.

10.1 Invocation

Use the "-c" option of the m3 compiler to create C code. You will be notified if you have used any restricted features. Additionally the "–main" option should also be used when compiling the module whose "BEGIN" .. "END" block you wish to run as main.

This will leave you with a collection of ".h" and ".c" files. Consult the manual for your cross compiler to find out how to compile and link these. For windows users who have installed Microsoft Visual C such that it is callable from the command line a simple batch file `ccomp.bat` has been provided in the `bin` directory. Consult the `test` directory for examples (hint: the test driver `ccode.bat` is a good place to start.)

If you use library functions, for which C code has been provided, then do not forget to compile and link these modules as well.

10.2 Mappings

10.2.1 Modules

Modules and Module Interfaces are transformed into ".c" and ".h" files. Interfaces which declare only types and have no corresponding implementation will result in the generation of a header file only. For Interfaces which contain variables an empty Module file must be provided which will map to a ".c" file containing the variable declarations.

The flat C namespace is split up by naming all entities (variables and procedures) in the system according to the module in which they are declared. Thus the integer `i` declared in the module `mod` is known in C as `mod__i` both locally in the module where it is declared and also whereever else it referenced in the program.

When several modules are compiled and linked together, one of them must have been selected as the main module, using the "–main" option described above. The execution of the program as such consists of the execution of the "BEGIN" .. "END" block of this module, *preceded* by the execution of such blocks for any other modules used. [4]

[4]Currently no guarantee is given as to the order in which the blocks are called, other than that the main block is called last. If you have complex interdependencies in your startup sequence you are safer if you make these explicit using normal procedure calls from the main module.

10.2.2 Types

- The C type `int` is used for all enumerated types, BOOLEAN, INTEGER and all their subtypes. The C type `float` is used for REAL.
- All REF types are mapped to C pointers, with the NEW function calling `malloc` with the size of the referent.
- RECORDs are mapped to `struct`
- ARRAYs to C arrays, with the offsets of any subscripts adjusted accordingly wherever they are used.

10.2.3 Parameters

VAR and VALUE parameter semantics is preserved for all types other than arrays. This means that the error prone (when done by hand) C mechanism of explicitly passing adresses or values depending on parameter mode is handled automatically.

Array parameters are always passed by reference. The use of the VAR qualifier is erroneous for array parameters.

10.3 Restrictions (permanent)

- `OBJECT`, `DICT` and `LIST` types are not supported.
- Nested procedures are not supported.

10.4 Restrictions (hopefully temporary)

- Anonymous types may sometimes produce uncompilable code. Solve this by using named types.
- Support for predefined procedures is (still) patchy at best.
- Initialisation of static objects with dynamic complex values (e.g. REFs with NEW, arrays with non-constant aggregates). The workaround here is to initialise them later with an assignment statement.
- Algol scope is not supported, the original declaration order is retained instead.
- Mutually recursive types cannot be defined. If you need these on the target your code is probably too fat anyway.
- Default field values in nested record declarations are ignored by constructors
- WITH only creates aliases for l-values (e.g. WITH a = 10 DO .. will not work)

10.5 The C Run Time System

To Be Completed : currently under implementation.

A Installation

The toolset should run without change on any computer with a current implementation of Python. Practically this has been tested for Win32, Intel-Linux and Mac OS X.

A.1 Prerequisites

- Install the latest python interpreter (2.3.3) available from www.python.org
- Check out the standard SESIS toolset from the subversion repository (or get hold of the zip file somehow)

You may also optionally wish to install the following

- Ghostscript - if present on your path, the commands `ps2pdf` and `gswin32c` are called to convert raw postscript output from the tool into `pdf` and `png` formats respectively.
- POVray - a ray tracing program. The capsule editor can create scenery files which can be rendered into high-quality 3-dimensional images by this program.

A.2 Environment variables

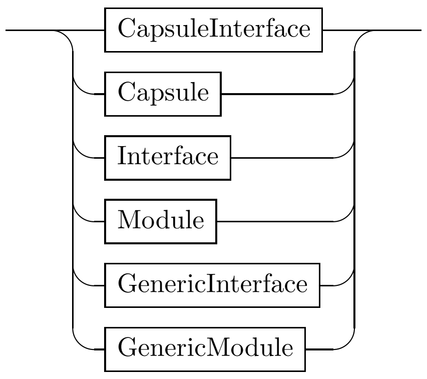
- Set the environment variable `M3_HOME` to the top level directory of the toolset – i.e. the one containing `bin`, `src`, `lib` and friends.
- Set your path to include the `bin` directory below `M3_HOME` and also the directory where you have installed python.
- Set your path to include the `bin` and `lib` directories of Ghostscript (if you have installed it)

A.3 Check that everything is working

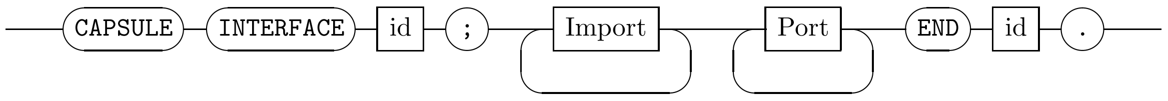
In the `test` subdirectory you will find a number of test programs for all parts of the language. Change into that directory and call the `testall` script. This may take a few minutes. If this runs without errors then your installation is OK, otherwise contact the author of this document immediately.

B Complete Syntax Summary

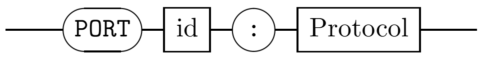
Compilation



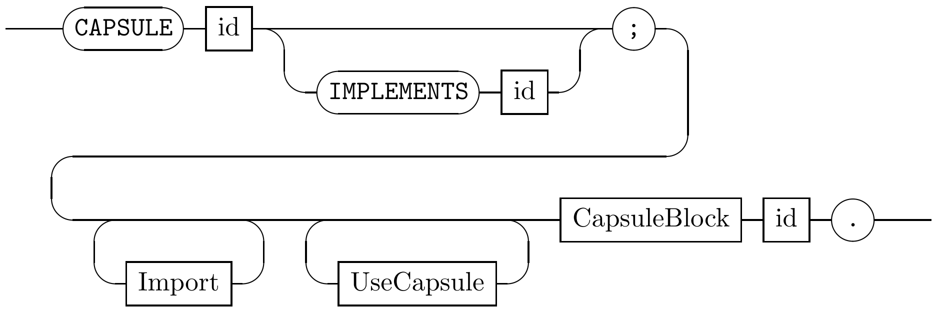
CapsuleInterface



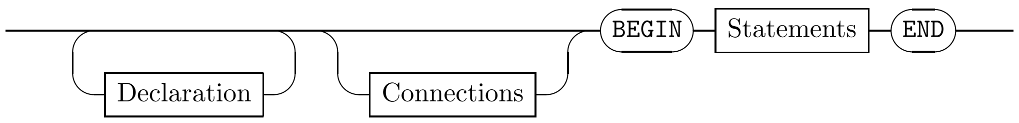
Port



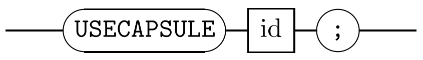
Capsule



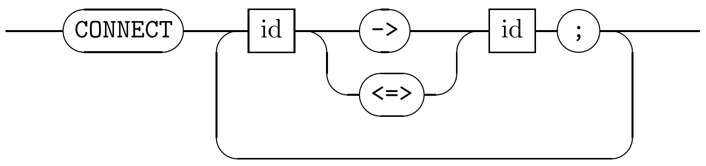
CapsuleBlock



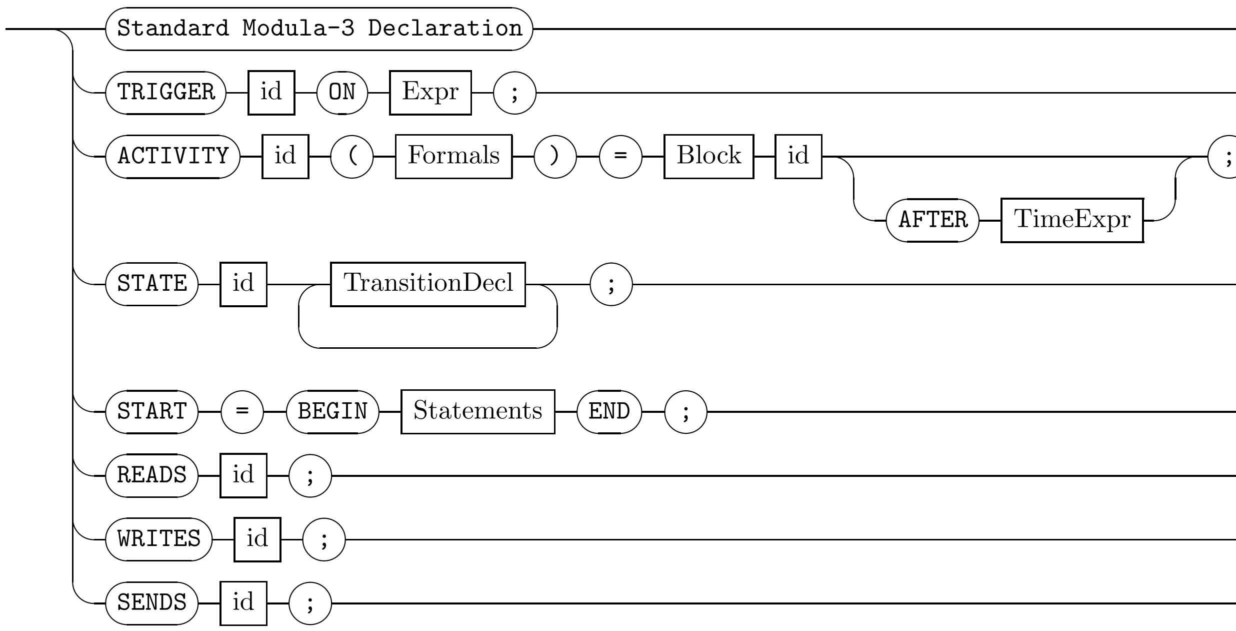
UseCapsule



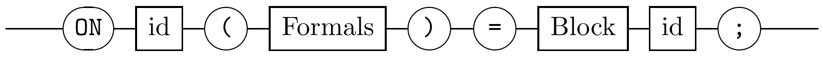
Connections



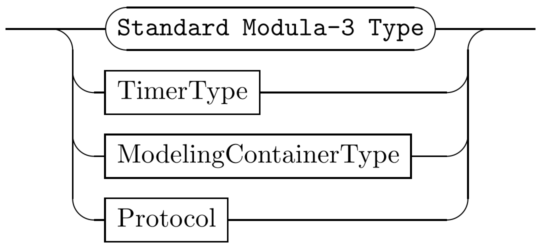
Declaration



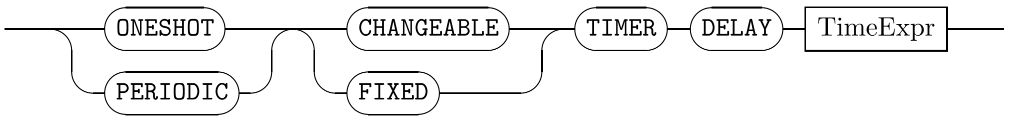
TransitionDecl



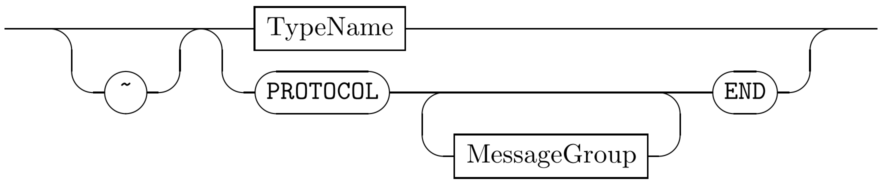
Type



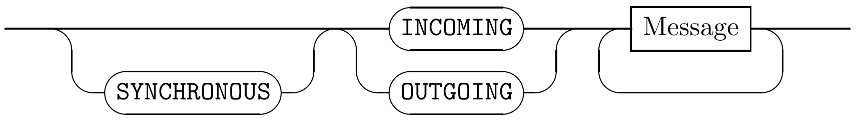
TimerType



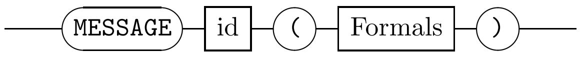
Protocol



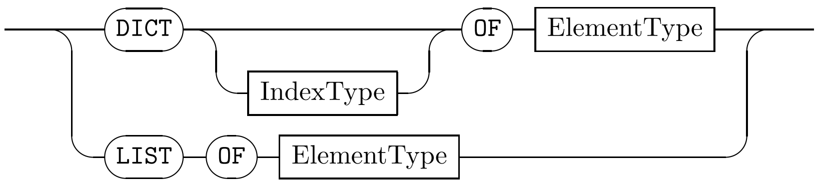
MessageGroup



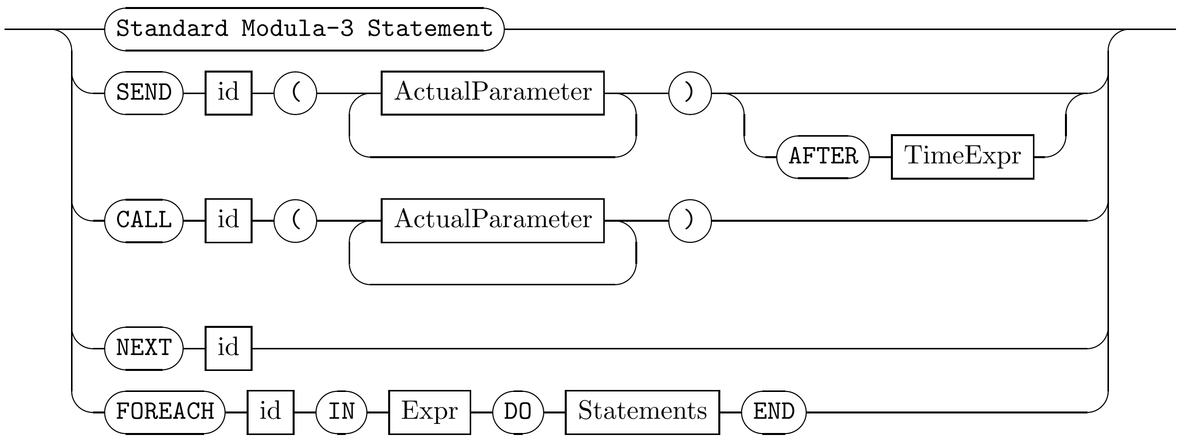
Message



ModelingContainerType



Statement



C Interfacing to the Generated Code

C.1 Implementing Modules in Python

If you need to access to some resource which is not provided by the current libraries, it is straightforward to slot in python code under a Modula-3 interface.

- For a given module X, write the interface `X.i3` as usual in Modula-3 and compile it.
- Create a python module named `XMod.py` making sure that it is on your path and that the functions declared in the python module correspond exactly in name and number of arguments with the interface.
- Dereference parameters as necessary to get at their raw python values (see below)
- Make sure that your `PYTHONPATH` includes the directory where you have stored XMod.py

C.2 Example

Here is how the IO library is currently implemented. In IO.i3 we have the following:

```
INTERFACE IO;

(* The following perform simple output to the terminal, for testing
   purposes it is recommended that you use the features provided by
   the Results Interface *)

PROCEDURE Put(T:TEXT);

PROCEDURE PutInt(I:INTEGER);

PROCEDURE PutReal(f:REAL);

PROCEDURE PutChar(C : CHAR);

END IO.
```

and in IOMod.py we implement the functionality as follows:

```
import sys
def Put(t):
    sys.stdout.write(t.val)
def PutInt(i):
    sys.stdout.write("%d" % i.val)
```

```
def PutReal(f):
    sys.stdout.write("%f" % f.val)
def PutChar(c):
    sys.stdout.write(c.val)
```

C.3 Modula-3 Type and Object Representation

Every Modula-3 object which you will receive as a parameter is an instance of one of the classes in the python module `M3Objects.py`. Each of these objects has a reference to its type which is defined in the python module `M3Types.py`. These two modules can be found in the `src/rts` subdirectory of the installation. Operations such as assignment, dereferencing and numerics are handled by the object itself, whereas type checking is handled by the type of the object. For further information consult the source code of those two modules and just try experimenting!

D Accessing the RTS from Python

This appendix describes how to use the python API to the run time system. Using the classes and methods described below, you can create models in "pure" python. This has the advantage that you have all the features of the python language immediately at your disposal. It also should be mentioned here that it has the *disadvantage* that your model is not statically checked for completeness by a compiler.

D.1 Installation

Proceed as described in A above, ignoring the optional components.

Set the environment variable `RAW_PYTHON` to a non-null string value.
[5]

No tools are considered here, only code which is made available to the python interpreter via `PYTHONPATH`

D.2 Capsules

Capsules are declared as classes deriving from the base class `M3CapsuleRuntimeType` in the `RTSTypes` module. The activities of that capsule are declared as methods of the class. Any other contents of the capsule, such as timers, triggers, connections or application data are declared in the constructor (`__init__`) of the capsule.

D.2.1 Example

Here is a simple capsule, in the file `pyact.py` corresponding to the diagram in Figure 15.

```
import Simulator
import RTSTypes
class act1(RTSTypes.M3CapsuleRuntimeType):
    def a1(self, foo, bar):
        print "activity a1 called with foo %s and bar %s" % (foo, bar)
    def __init__(self,level):
        RTSTypes.M3CapsuleRuntimeType.__init__(self,level)

Simulator.run(act1(1))
```

D.2.2 Running the Capsule

Before this capsule can run, some input must be prepared. In this case we have a activity named `a1` which expects two parameters `foo` and `bar`

[5]This tells the RTS not to expect Modula-3 types synthesized by the compiler, the –raw-python option can also be used as an additional command line argument to the same effect

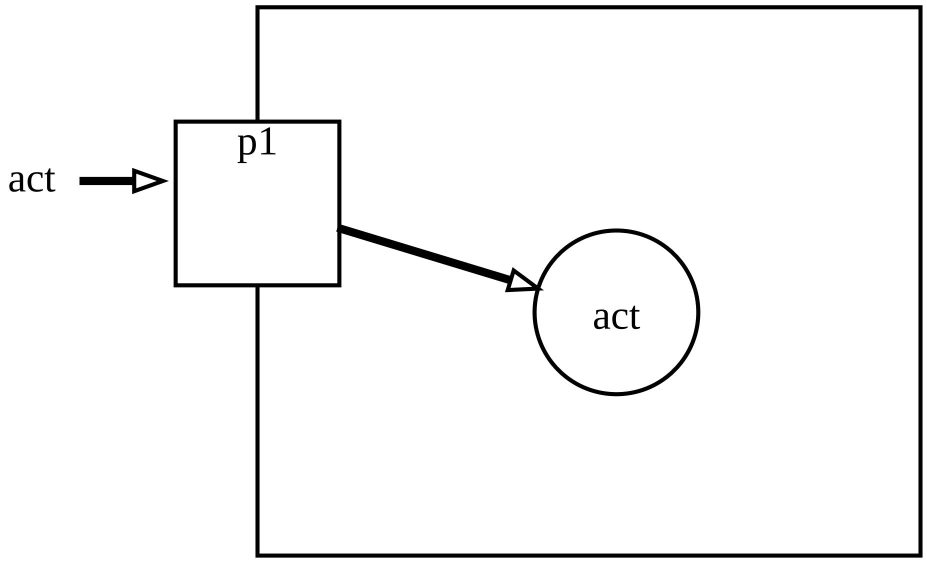


Figure 15: The Simple Capsule

In order to send a message to the system with those two parameters we prepare an input file `pyact.inp` with the following input.

```
> a1 foo=1, bar=2
```

and run the capsule using the python interpreter with the following command:

```
python pyact.py pyact
```

and this generates the following output, which is simply the result of the print statement in the activity:

```
activity a1 called with foo 1 and bar 2
```

Apart from the definition of the capsule class `pyact`, the example contains only the call to the `run` function of the `Simulator` module which manages the feeding of the input into the capsule.

D.3 Activities

Activities are declared as methods of capsule classes. The parameters can be of any python type. Optional parameters are also supported.

See Section D.10 below for further details on conversion of input to python parameters.

D.4 Timers

D.4.1 Example

Here is a capsule containing a timer, connected to an activity which itself uses the local data item `i`. It corresponds to the diagram in Figure 16.

```
import Time
import RTSTypes
# Defining a capsule
class pytimer(RTSTypes.M3CapsuleRuntimeType):

    # Defining an activity (which is simply a method of the capsule)
    def timeact(self):
        # Application code
        self.i += 1
        print "Timer Demo %s" % self.i
        # Sending a message
        self.send(originator=self,
                  level=self.level,
                  msgName='foo',
                  myParam=self.i)

    # Defining the capsule contents and structure in the constructor
    def __init__(self,level):
        # Call parent constructor
        RTSTypes.M3CapsuleRuntimeType.__init__(self,level)
        # Defining a timer
        self.myTimer=RTSTypes.M3TimerRuntimeType(delay=10*Time.F_psec,
                                                  periodic=True,
                                                  changeable=False).createObject()
        # Connecting the timer to an activity/message
        self.myTimer.connect(context=self,
                             srcMsg ='',  # not needed by timer
                             destMsg = 'timeact',
                             destObj = self,
                             level = level)
        # Start the timer
        self.myTimer.start()
        # Store level for use by "send" later
        self.level = level
        # Application specific data
        self.i = 0
if __name__ == '__main__':
    import Simulator
    Simulator.run(pytimer(1))
```

D.4.2 Timer Definition

Timer definition occurs in two phases. First the timer type is defined by instantiating an object of the class `M3TimerRuntimeType`. The constructor expects the following parameters

- `delay` : the time delay in picoseconds (see below).
- `periodic` : if this is set then the timer will restart automatically on elapsing.
- `changeable` : defines whether the timer can be changed.

Once the type has been created, then a timer object is created from that type using the `createObject` method. In the example above, type and object creation are run together in one line. It is however possible to create several separate timer objects from one type.

D.4.3 Timer Functions

Once created, a timer offers the following methods to the user:

- `stop` : stops the timer
- `start` : starts the timer
- `change <integer-value>` : changes the value of the timer - with no influence on the current delay if there is one.
- `connect` : see section D.5 below.

D.5 Making Connections with connect

In the previous example we saw our first connection. Connections are used to attach timers activities which will be called when they time out. A connection is created by calling the connect method on the timer with the following parameters:

- `context` : the capsule in which the connection is created (normally just use `self`)
- `srcMsg` : not used by the timer, can be given a dummy value.
- `destMsg` : a string, which is the name either of the activity (which is the case here) or of a message.
- `destObj` : the destination object, in the case of a local connection this is `self` again. More on this below.
- `level` : The nesting level at which the connection is created. Use the `level` value which has been given to you as a constructor parameter.

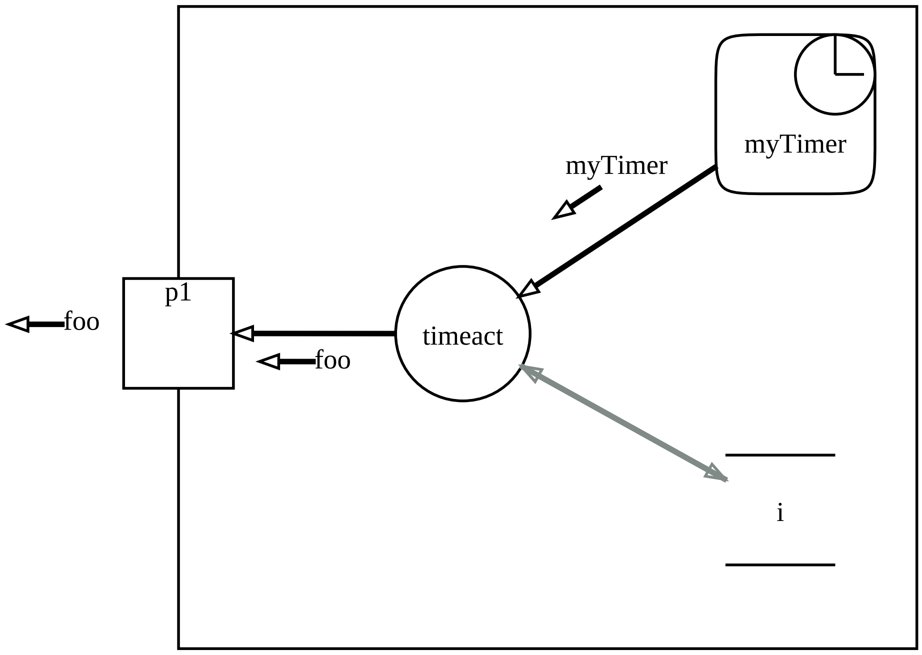


Figure 16: The Timer Capsule

D.6 Message Dispatch with send

Messages are sent using the `send` method of the owning capsule. This method has the following parameters:

originator The capsule in which connection is created (normally just use `self`).

level The nesting level at which the connection is created. Use the `level` value which has been given to you as a constructor parameter.

msgName The message you want to send.

... application specific parameters ... These must fit those of the final destination using the normal python parameter handling features (i.e. keywords and default parameters).

D.7 Nested Capsules

Nestability is an essential feature of capsules, and it is only when we start to nest capsules that features such as message routing and run to completion (see section 7.2 above) become relevant. This nesting is naturally easier to visualise graphically, and the diagram for the following example is in Figure 17. In this example we have a top-level capsule C1, with two child capsules of type C2 and

C3 respectively. The test message `input` is connected to the input message of the child instance `c2` which in turn sends a message to `c3` which finally sends the message `response` which is connected back to the outputs at the top level.

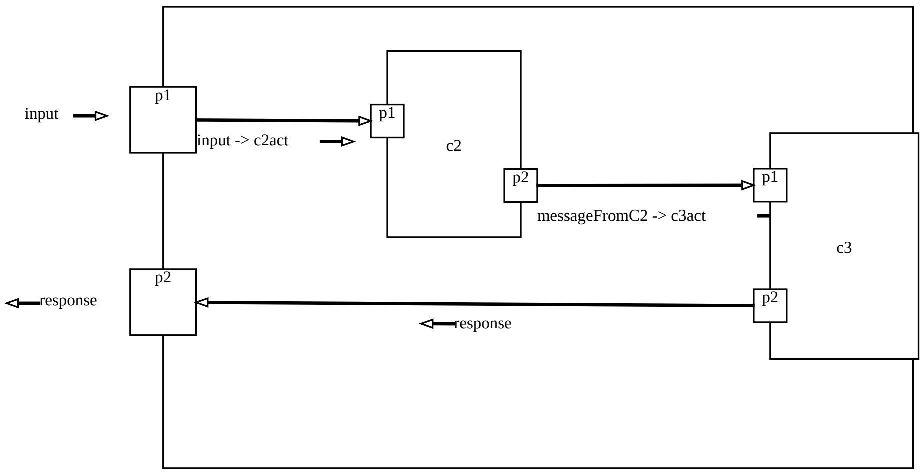


Figure 17: Nested Capsules

The code for the top level is as follows:

```
import RTSTypes
import Simulator

import C2
import C3
class C1(RTSTypes.M3CapsuleRuntimeType):
    def __init__(self,level):
        self.level = level
        RTSTypes.M3CapsuleRuntimeType.__init__(self,level)
        self.c2=C2.C2(level+1)
        self.c3=C3.C3(level+1)
        self.connect(context=self,
                     srcMsg='input',
                     destMsg='c2act',
                     destObj=self.c2,
                     level=level)
        self.c2.connect(self,'messageFromC2','c3act',self.c3,level)
        self.c3.connect(self,'response','response',self,level)
Simulator.run(C1(1))
```

From looking at the code we can deduce nothing about the structure of the two subcapsules C2 and C3. We have presupposed, in the connection statements, that they will be able to answer and send the appropriate messages.

Here is the code for the child capsules:

```
import RTSTypes
class C2(RTSTypes.M3CapsuleRuntimeType):
    def c2act(self):
        self.send(self, self.level, 'messageFromC2', )
    def __init__(self,level):
        self.level = level
        RTSTypes.M3CapsuleRuntimeType.__init__(self,level)

import RTSTypes
class C3(RTSTypes.M3CapsuleRuntimeType):
    def c3act(self):
        self.send(self, self.level, 'response', )
    def __init__(self,level):
        self.level = level
        RTSTypes.M3CapsuleRuntimeType.__init__(self,level)
```

D.8 More about Connections

The capsules in the previous example were held together by three connection statements placed in the constructor of the root capsule.

```
        self.connect(context=self,
                     srcMsg='input',
                     destMsg='c2act',
                     destObj=self.c2,
                     level=level)
        self.c2.connect(self,'messageFromC2','c3act',self.c3,level)
        self.c3.connect(self,'response','response',self,level)
```

Note that all `connect` method calls are made with the `context` of the current capsule (`self`) but that the object on which the method calls is made is different. The object on which the method is called is always the source of the message which is being connected.

D.9 Triggers

D.9.1 Example

A trigger is a component which waits for a condition to occur in the system and then generates a message. In our example we have an input message `dec` causing a local variable to be decremented. The variable has been initialised on capsule construction with a value of 2. The trigger tests whether the variable has the value 0, and then, because it is so connected, causes the `alarm` message to be sent out.

Here is the code corresponding to the diagram in Figure 18

```
import Simulator
import RTSTypes
class pytrigger(RTSTypes.M3CapsuleRuntimeType):
    def triggerChecker(self):
        return self.counter == 0
    def dec(self):
        self.counter -= 1
    def handler(self):
        self.send(self, self.level, 'alarm', )
    def __init__(self,level):
        self.level = level
        RTSTypes.M3CapsuleRuntimeType.__init__(self,level)
        self.isZero = RTSTypes.M3TriggerType(self.triggerChecker)
        self.counter = 2
        self.isZero.connect(self,'','handler',self,self.level)

Simulator.run(pytrigger(1))
```

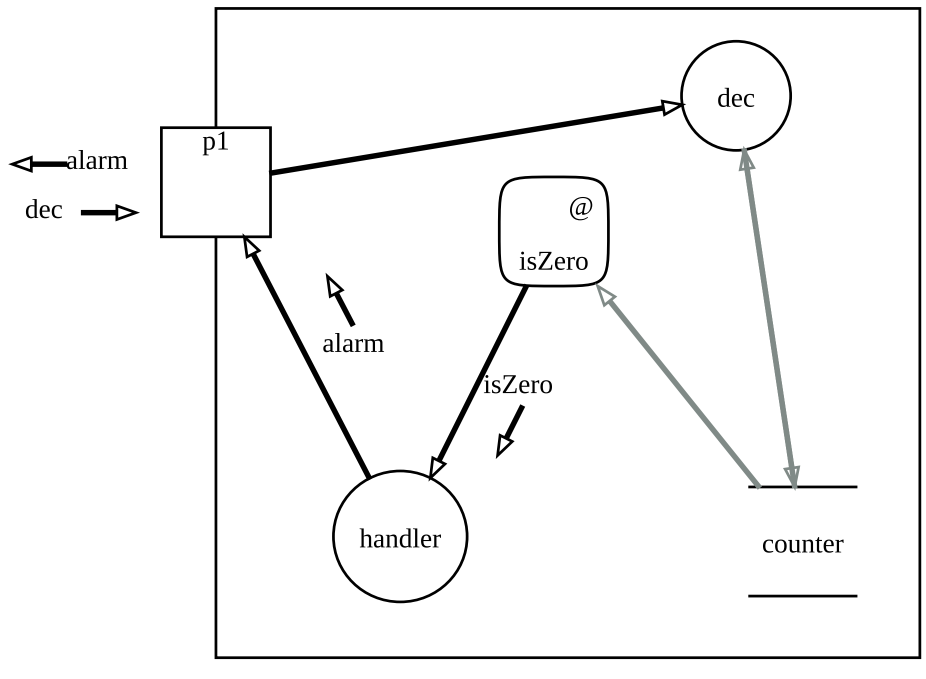


Figure 18: Diagram for Trigger Example

D.9.2 Defining a Trigger

The definition of a trigger comes in two parts. First we define the test function, this can be any valid executable python object. The standard python definition of truth/falsehood is used. In the example we have used a method as the test function, which has the advantage that we can access the environment of the capsule, which in this case includes the counter we are testing.

The second part is the definition of the trigger itself which is an instance of the `M3TriggerType` class. We supply the test function as a parameter to the constructor.

Finally, we connect the trigger, in this case to a local activity, just as we connected the Timer in section D.5

D.10 Testing and type conversion

For the syntax of input files see Section 6.2 above. When using python, (i.e. with the RAW_PYTHON environment variable set) compound input values are mapped to the `Dict` and `List` types. Simple input values are mapped to `Float`, `Int` or `String` by testing whether they can evaluate to such a value.

For example the following input file:

```
> a1 foo={a: 1, b: 3.415, c: blather} bar = [1,2,3,4]
```

can be fed to the following capsule:

```
import Simulator
import RTSTypes
class act1(RTSTypes.M3CapsuleRuntimeType):
    def a1(self, m, foo, bar):
        print "activity a1 called with foo %s and bar %s" % (foo, bar)
        print "keys of foo are: ", foo.keys()
        print "length of bar is:", len(bar)
    def __init__(self,level):
        RTSTypes.M3CapsuleRuntimeType.__init__(self,level)

Simulator.run(act1(1))
```

and will produce the following output:

```
activity a1 called with foo {'a': 1, 'c': 'blather', 'b': 3.415} and bar [1, 2, 3, 4]
keys of foo are:  ['a', 'c', 'b']
length of bar is: 4
```

E Comparison with ASCET

Axis	ASCET	SESIS Tool
business model	commercial	in house
status	productive	experimental
asset storage	database	language
market	vertical	horizontal
OS	ERCOSEK	VPS
code	closed	open
timing	time triggered	event based
size	big	small
cardinality	single model	multi model
execution	compiled	interpreted or compiled

F Quo Vadis?

The question arises whether the current approach to SESIS tooling is scaleable. If we decide that this is NOT the case then we need to think about possible future developments or even *exit strategies* if and when the ideas behind SESIS achieve a critical degree of acceptance. In brainstorming mode (i.e. no idea is too crazy) future directions for the tool/language could be

1. Eclipse Plugin
2. XMI Output
3. C Output
4. Front End for gcc
5. UML Metamodel
6. Integrate with leading case tool
7. Customize case tool (GME ROSE TAU)
8. No Exit : spread the word on cognitive automation and fork the tool for each new project.

References

[1] M. Rittel, Ein Systems-Engineering-Ansatz für software-intensive Produktlinien, Robert Bosch GmbH, Frankfurt 2002

[2] M. Rittel, Formale Domänen-Modelle, Robert Bosch GmbH, Frankfurt 2003

[3] S.M. McMenamin, J.F. Palmer Essential Systems Analysis, New York, 1984

[4] Greg Nelson, Ed., Systems Programming with Modula-3, New Jersey, 1991

[5] Samuel P. Arbison, Modula-3, New Jersey, 1992

[6] http://www.python.org

Index