SDL '87: State of the Art and Future Trends
R. Saracco and P.A.J. Tilanus (eds.)
Elsevier Science Publishers B.V. (North-Holland)

AUTOMATIC PROGRAM GENERATION OF SDL-SPECIFICATIONS: PRINCIPLES AND SOLUTIONS

ULRIK JOHANSEN, ELAB AND EIRIK VEFSNMO, A.S COMPUTAS

ELAB
O.S. Bragstads plass 6
N-7034 Trondheim-NTH
NORWAY
+47 7 592672

A.S. Computas
P.O. Box 3765, Granaaslia
N-7001 Trondheim
NORWAY
+47 7 917500

ABSTRACT

The paper describes results from a research and development project carried out in a cooperation between A.S Computas and ELAB. The main objectives for this project was to study principles for automatic program generation of SDL-specifications, and to use these principles in developing a program generator for the SDL-tool DASOM. (DASOM is described in a separate paper)

As an introduction, the paper discuss main principles for implementing descriptions based on finite state machines.

1. INTRODUCTION

The main objective with this paper is to describe the specification and implementation of an automatic program generator. The program generation is based on SDL-specifications containing functional descriptions which again is based on the concept of finite state machines (FSMs).

In addition to the description of the program generator itself, different implementation strategies of SDL-specifications are discussed in order to indicate problems in optimizing implementations of systems described in a functional manner (and by the aid of finite state technics).

2. PRINCIPLES FOR IMPLEMENTING FSMS

At least two main principles for implementing SDL-specifications (finite state machines) exists. The first solution is called "mechanism adoption" where a mechanism which may be programmed directly with data definitions and state transition diagrams are used (We also use the expression "interpreting solution" for this). The second solution is called "language adoption" where the functional processes (described by state transition diagrams) are translated into a programming language directly (often with the use of SDL/PR as an intermediate step).

These two principles are discussed in more details in the next two sections.

2.1 MECHANISM ADOPTION

The mechanism adoption may be implemented both in hardware and software. Here we consentrate on software implementations.

The software implementation of the mechanism adoption includes implementation of a general program (FSM support) which performs tasks common to all finite state machines (FSMs). The structure in such an implementation is shown in Figure 1; and it consists of the following elements:

- FSM support system
- State transition diagrams
- Action descriptions
- Process individ data
- Input signal

The FSM support system's task is to perform:

- DO FOREVER
  - Get next input signal
  - Decode receiver address information in the input signal and find the process individ data
  - Use the current state value to find the state transition description
  - Scan through the state transition diagram for current state value (for the process) in order to find a match between the input signal and specified signals valid for current state.
  - Execute specified action
  - Set specified next state as current state for the process
- END

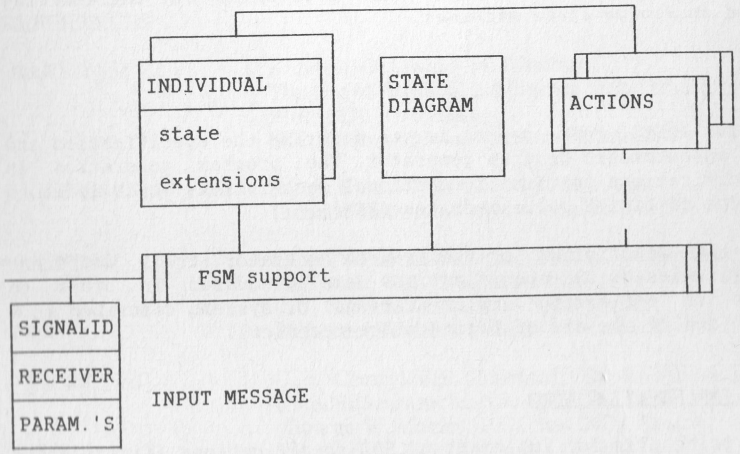

Figure 1: FSM support system with one process type

Figure 1 indicates that the FSM support system is able to handle one state transition diagram, and several process instances of the same type. The FSM support mechanism may, however, be extended to handle several process types, each type with their own state transition diagrams; and there may also be several process instances for each process type. The requirements for such a

structure are, however, that both process type and -individ identification (receiver identification) is contained in the input signal.

The state transition diagrams are normally implemented as data. The data may be structured in different ways. The criteria for the structuring of the data is most often concerned with the optimization of memory space, or the reduction of searching times. Usually the implementation of a state transition diagram is organized as a description of a set of states. For each state a set of transition definitions exists. For each transition definition the following elements are specified:

- Signal identifier (Used to match the input signal)
- Next state value (Next state value for this process individ)
- Action description (Reference to actions to be performed)

The implementation structure for state transition diagrams depends on both the application requirements and the implementation programming language. I.e. if all signal types are defined in all states in a state transition diagram, the state transition diagram may be implemented as a two dimentional array. The first index may be the state value (represented as an integer), and the second index may be the signal identifier (also represented as an integer). For each element in the array the next state value and the action description must be given. If the state transition diagram uses many different signal types, but only a few transitions are defined for each state, the previous solution for the implementation will be very memory consuming. In such cases a solution using indexed array to point at states and a linked list describing transitions for each state, will be preferable.

As described above the implementation according to mechanism adoption may result in many different solutions. Each solution has it's advantages and disadvantages. The solution should be optimized with respect to the application described, and will be limited by the programming language used.

2.2 LANGUAGE ADOPTION

By using language adoption in the implementation of state transition machines, the solution will be much more closely related to the implementation programming language used. The most used programming language mechanisms are the IF and CASE statements. Using a pseudo language as the expression form, the most common implementation form is:

```
CASE state of
 S1: CASE insignal of
     SIGA: { actionsA, state = S2 },
     SIGB: { actionsB, state = S1 }
     END
 S2: CASE insignal of
     SIGC: { actionsC, state = S1 }
     END
END
```

where "state" is the current state value of the process instance and "insignal" holds the value of current input signal.

The state transition diagram in the example contains two states, S1 with defined input signals SIGA and SIGB, and S2 with input signal SIGC.

A general comment to the language adoption strategy is that it often gives efficient solutions with respect to execution time. That is because of the direct use of the programming language's own mechanisms. Another advantage

with this strategy is that it is simple to use since there is no need for a FSM support system. The main disadvantage with language adoption is the close relation to the implementation programming language. The portability of the program code is reduced. In addition, this implementation will be relative expensive with respect to use of memory.

2.3 CONCLUSIONS

In general, both implementation strategies mentioned in this section have their advantages and disadvantages. What are advantages and what are disadvantages depend on the application itself, and on the implementation programming language which is used for the implementation.

The purpose with discussing these items here are to show that making an optimized software implementation of a system designed by using SDL-specifications (finite state machines) are not easy. This optimization will depend on many factors as mentioned, for instance the type of application and the hardware used.

This section also shows that a functional description is not enough input to an automatic program generator for generating an optimized program code. The software designer must be able to control the program generator in order to get an optimized application system solution. This requirement has been the most important one in the specification and implementation of the program generator described in the next section.

3. AUTOMATIC PROGRAM GENERATION

3.1 REQUIREMENTS TO THE PROGRAM GENERATOR

The most important requirements to the program generator are:

a) Target programming language independency

The program generator itself, and the philosophy for how it works, shall as far as possible be independent of the programming language used.

b) Program generation of system components

It shall be possible to select parts of a total system, and to generate program code only for the selected parts. This possibility is important especially during the development of a system, while program generation for the total system is most interesting in production, where the configuration of a system is essential.

c) Flexibility

A very important requirement is that the program generator shall be flexible. It implies the possibility to control the program generator in such a way that special requirements to the implementation code can be fulfilled. Examples of such requirements may be the optimization with respect to execution time or reduction of memory space usage. It may also be requirements related to equipment hardware adaption.

d) Integration

The program generator shall be integrated as a part of DASOM, and therefore it

shall be possible to control and operate it through the normal DASOM user dialogue. The primary information to be used by the program generator shall be the SDL-specifications defined in the application part of DASOM's relational data base.

3.2 SATISFYING THE REQUIREMENTS TO THE PROGRAM GENERATOR

The implementation of the program generator has taken care of the specified requirements as far as possible. Below we discuss how each of the main requirements are fulfilled.

a) Target programming language independency

As far as possible, the program generator shall know nothing about the target programming language desired. Principally, this means that the implementation of the program generator for a new target programming language shall imply no changes in neither the program generator, nor in the functional model describing the target system.

This is achived by the use of "skeleton files". These files contain a skeleton of the program code to be generated. During the generation of the skeleton files an implementation strategy must be selected. This has been possible because of the relative simple mapping between the functional model based on SDL-specifications (finite state machines) and the implementation model.

To have any program code generated at all, it is necessary for the program generator to interpret the skeleton files. Therefore, there is defined a set of commands which the program generator will recognize when it interprets the skeleton files. However, it is important to notice that these commands are not related to the target programming language which the skeleton files are based on.

b) Program generation of system components

The program generator is an integrated part of DASOM, and because there will be information in DASOM describing the functional and implementation model of the application, the possibility to control the program generator in such a way that the program generation may comprise either a part of the system, or the whole system is available. Experiences have resulted in a standarized way of implementing software systems, and from these experiences the following levels for program generation of system components have been selected:

- Program generation for the whole system
- Program generation for a software process
  (A software process may for instance be a CHILL-process, and it may contain one or more functional processes)
- Program generation for a functional process (or SDL-block)

c) Flexibility

The main factor allowing flexibility in the program generator, is the use of skeleton files.

By using a set of rules each user may generate their own set of skeleton files. The rules require knowledge about:

- The hierarchical structure of the logical model description.
  The logical model is a mapping of the DASOM model information into an internal representation used by the program generator.

- Symbolic names for identifiers used to identify values defined in the logical model.

- Knowledge about defined commands that the program generator is able to recognize and interpret.

It is also important for the program generator's flexibility that the results from the program generator is symbolic code, which is possible to edit and make changes on, even after the program generation.

Compilers for different computers, but for the same programming language, mainly have syntactical differences in the programming language. This means that relative small chance will be necessary for the adoption of the program generator to another computer. The modifications necessary will be possible to perform only by changing the skeleton files.

It is possible to use different program skeletons for the generation of program code in the same application. The selection of skeleton type will depend on the intention with the program generation. During the development phase, it is important to generate program code for testing of the system where modules of the total system may be tested separately. This allows what is commonly called "rapid prototyping".

d) Integration

The program generator is integrated as a separate part of DASOM. The program generator is defined in DASOM as an own object which is activated and controlled through the ordinary user dialogue of DASOM.

3.3 PRINCIPAL SOLUTION

Figure 2 shows the implementation structure of the program generator. How it works is briefly described below.

a) Activation

The program generator is activated through DASOM by using the user dialogue. The activation is done when a user wants to generate programs for an application system or subsystem. When activated, a reference is given to the program generator telling what functional model in DASOM that is selected.

b) Logical model

When the program generator is activated it starts building a logical model by using the information retrieved from the functional model in DASOM. The main difference between the information in DASOM and the information in the logical model is that information not necessary for the program generation is removed (I.e. graphical layout information). Besides, the structuring of the information is adopted to fulfill the program generator's requirements.

c) Program generation

The program generation is performed when the program generator reads and interprets the contents of the specified skeleton files, and combines this information with information retrieved from the logical model. The result is the "result files". These files contain symbolic compilable program code.

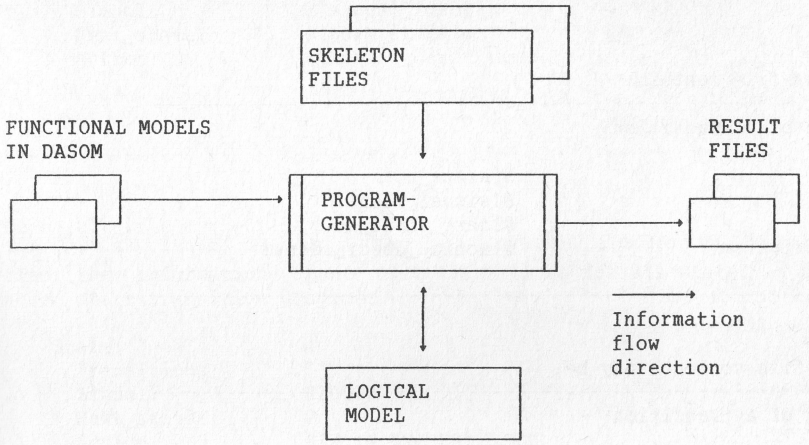

Figure 2: Principal solution for the program generator

3.4 COMMANDS TO THE PROGRAM GENERATOR

Making the program generator able to generate more program code than what already exists in the skeleton files, it is necessary to give the program generator instructions about what to do in order to pick up information from the logical model describing the application, and to substitute this information with that stored in the skeleton model.

Studies have shown that only a few different command types are necessary to make the program generator work. So far, only three different types have been specified. However, until now we have considered relatively limited program generations. We have also selected the programming language which we believe will give the simplest mapping between our functional model, and the chosen implementation model. The programming language "C" has been used. Development of more complex program skeletons, or use of other programming languages will probably make it necessary to extend the repertoire of available commands for the program generator. The three different command types defined are:

1. Substitution of a logical variable

DASOM makes use of defined notions to identify elements in the description of the application (I.e. state transition diagram, state, transition, functional process). Such elements are called objects in DASOM. Each object consists of an identifier and possibly a set of attributes (I.e. for the object transition such attributes may be next state, input signal and action description).

For the program generator the notions for object types and their attribute values are known. The mapping between symbolic names used in the skeleton files and the object type definitions in DASOM are defined in a table read by the program generator during activation. It is therefore possible to ask the program generator to substitute actual values for the formal values specified by symbolic names in the skeleton files. The syntax used for the substitution command in a skeleton file is:

.. text .. @!substitution_identifier! .. text ..

Example:

The skeleton file contain:

```
Description of a transition:

State:                          @!state_no!
Signal:                         @!signal_ident!
Next state:                     @!next_state_no!
Action description:             @!action_descr_ident!
```

The result file contents may be:

```
Description of a transition:

State:                          2
Signal:                         REQUEST
Next state:                     5
Action description:             SEND_SIG_GRANTED
```

The content of the result file tells that the chosen transition description has a start state 2, the next state is 5, and the associated action reference is SEND_SIG_GRANTED.

2. Repetition of an object set

In the description of the application system, the attribute values for an object often represents a set of new objects. The number of elements in these sets may vary within large limits. It is therefore necessary to have the possbility to inform the program generator that treatment of a whole set of objects is necessary. (I.e. give an description of all transitions with a specified start state). The syntax for the "repetition set" command is:

```
@*   .. text ..   @!substitution!   .. text ..   $
```

A repetition sequence starts with @* and terminates with $. Everything between @* and $ will be repeated as many times as there are elements in the set defined by @!substitution!. How the repetition set command works is best illustrated by an example:

The skeleton file contain:

```
Write the information about all transitions for all states in the
chosen state transition diagram:

@*   State:              @!state_no!
       Transitions:
@*     Signal:           @!signal_ident!
       Next_state:       @!next_state_no!
       Action:           @!action_descr_ident!
$

$
```


The result file content may be:

```
Write the information about all transitions for all states in the
chosen state transition diagram:

     State:              1
       Transitions:
       Signal:           REQUEST
       Next_state:       2
       Action:           SET_IN_REQUEST_QUEUE

       Signal:           FREE
       Next_state:       3
       Action:           SET_IN_RESOURCE

     State:              2
       Transitions:
       Signal:           FREE
       Next_state:       2
       Action:           GIVE_RESOURCE_TO_REQUEST
```

The example shows that the state transition diagram have the states 1 and 2. State 1 has 2 transition definitions with the receiption of signals REQUEST and FREE, while state 2 has one transition with the receiption of signal FREE. More than one level of repetition of a set may be active at the same time, and it is possible with more than one substitution within the same repetition of set loop.

3. Inclusion of new skeleton files

During a program generation it will be desirable to use the same skeleton file, or set of skeleton files, more than once within the same program generation (I.e. program generation for more than one process type). The repetitive use of skeleton files are made possible through the "include skeleton file" command to the program generator. The syntax for the command is:

```
@<skeleton_file_name>          <result_file_name>
```

How this command works is demonstrated by an example:

The skeleton file contain:

```
Generation of program code for all process type definitions:

@*   @<pr_typ_skeleton>  <prtyp_@!pr_typ_ident!>
$
```

The result file contents may be:

```
-----------------------------------------------------------------
Generation of program code for all process type definitions:

      pr_typ_skeleton      prtyp_ALLOCATOR
      pr_typ_skeleton      prtyp_PROCESS


-----------------------------------------------------------------
```

The example above shows that the skeleton file called "pr_typ_skeleton" is used twice. Two result files with the names "prtyp_ALLOCATOR" and "prtyp_PROCESS" are generated. The content of the result files will be the program code defining the process types with identifiers ALLOCATOR and PROCESS. The example also shows that the include-skeleton-file-command is legal within a repetition-set-command, and that substitution is allowed within the include-skeleton-file-command (I.e. the functional process identifiers (ALLOCATOR, PROCESS) are used to decide the physical program code file names). It is also allowed with several nesting levels of include-skeleton-file-commands (Not shown in the example).

4. SUMMARY AND CONCLUSIONS

The development of the program generator has been a cooperation between A.S Computas and ELAB, and started in January 1986. After preliminary studies and selection of solution principles the first phase in realization was performed. This phase contained the realization of the program generator kernel with the interpretation of skeleton files and the connection to the logical model. At first, the logical model was retrieved from a text file instead of from DASOM.

Phase 2 of the development was the integration into DASOM, where retrievement of information from the data base of DASOM and the generation of the logical model were the main tasks.

The program generator was integrated into DASOM late 1986 and is today a part of DASOM, and is able to work on the SDL-specifications in DASOM's data base.

This paper states that a quite simple principal solution is used. However, the effective use of the program generator requires a standarized way to implement a system (only a limited set of program skeletons are desirable). All examples where the program generator have been used are base on the use of finite state machines, but within this area the use has been successful.

REFERENCES

[1] U. Johansen: "RUNSYS - An object oriented task scheduling system"
ELAB-report STF44F87006, January 1987

[2] R. Bræk: "Veileder i SOM (SDL-Orientert Metodikk)
ELAB-report STF44A84187, October 1984
(In norwegian)