Automatic code generation from SDL to a declarative programming language

Magnus W Fröberg[a] [b]

[a]Computer Science Laboratory,
Ellemtel Utvecklings AB, Box 1505, S - 125 25 Älvsjö, Sweden

[b]Published in SDL '93, Proceedings of the 6th SDL-Forum : Edited by A. Sarma and O. Faergemand

This paper describes a prototype code generator from SDL/PR to Erlang, a declarative language which supports concurrency, distribution, real-time, error handling and code updating in running systems.

The code generation is automatic and complete. The results indicate that the number of generated Erlang lines of code are almost the same as the originating number of SDL/PR lines. The resulting Erlang code can be run immediately.

The close mapping between SDL and Erlang is discussed. A brief introduction to Erlang will be given. The model of the SDL system used will be described and a simple example will be given.

1. INTRODUCTION

For many years, SDL [1] has been used to specify real-time applications. SDL has mostly been used in the area of telephony.

Today there exist powerful tools that can simulate SDL flows and generate program code from SDL. A drawback is that the generated code is often C, or similar languages, which are poorly suited for telephony. However, there is often a large gap between the specification and the implementation.

In this paper we describe a method of transforming SDL specifications to the declarative programming language Erlang. Erlang is a declarative, functional real-time language which has many similarities with SDL and is extremely well suited for telephony [2, 3]. The transformation is complete and automatic.

The results indicate that the number of generated Erlang lines of code are almost the same as the number of SDL/PR (the textual phrase representation of SDL) lines.

The combination of a well known and established specification technique, SDL, with modern declarative programming languages proves to be a powerful design environment for telecom software development.

The SDL descriptions used in this project were written using SDT (SDL Design Tool). SDT is a toolset from TeleLOGIC Malmö AB. It consists of a number of tools, e.g., create and edit SDL/GR (the graphical representation), transform SDL/GR to SDL/PR, check of syntax and semantics, code generation and a simulator.

2. ERLANG

The Erlang language grew as a result of a series of experiments, aimed at creating a language suitable for programming large scale telephony applications [4, 5]. It is a concurrent declarative language with real-time support. Some of the main characteristics of Erlang are as follows:

- Functional notation.
- Explicit concurrency.
- Asynchronous message passing between processes.
- Pattern matching.
- Distribution.
- Sophisticated Error handling.
- Modular.

Erlang provides the following data structures:

- **atoms** - Constants and fixed atomic data.
- **numbers** - Integers and floating point numbers.
- **tuples** - A fixed number of items grouped together.
- **lists** - A variable number of items grouped together.

Variables in Erlang are single assignment and are used like in mathematics or in other functional languages.

The Erlang pattern matching mechanism makes it possible to write programs in a declarative style, e.g., the factorial function is written:

```
factorial(0) ->
  1;
factorial(N) when N > 0 ->
  N * factorial(N - 1).
```

the second clause of the function definition matches for all `N` greater than `0`. The first clause matches (the base case) when `N` is equal to `0`.

Erlang handles concurrency explicitly through light-weight processes. Within any individual process the behaviour is sequential but any number of such sequential processes are allowed to execute concurrently. The size of a process increases and decreases dynamically. New process instances can be started using the `spawn/3`[1] Built In Function (BIF).

Processes communicate by sending and receiving messages. Messages are sent to another process with the binary infix operator "!".

[1]The notation Fn/N denotes a function named Fn of arity N.

```
ProcessId ! {hello, world}
```

`ProcessId` is a variable representing the process identity of the receiving process. The expression on the right hand side of the "!" operator is the actual data that will be sent to the other process. The receiving process could be suspended in a `receive` statement where active message reception is used. The `receive` statement could look like:

```
receive
  {hello, moon} ->
    do_moon();
  {hello, What} ->
    do_hello(What)
end
```

In the receive statement, messages will be selectively received, i.e., the `{hello, world}` message will be received in the second branch, binding `What` to `world`. Messages will be queued if no `receive` branch matches and the process will remain suspended.

The concept of time is handled by the use of timeouts in `receive` statements.

```
receive
  Whatever ->
    do(Whatever)
after 1000 ->
    do_timeout()
end
```

The timeout time is specified in milliseconds. In the example above, if no message has been received within one second a timeout will be generated, and the function `do_timeout()` will be called.

A process can monitor the behaviour of another process using the error handling mechanism. A process can set up a *link* to another process during execution. When a process terminates, an exit signal is sent to all processes in its link set. A process that receives an exit signal will by default terminate. A process can trap exit signals in order to act upon the signal, e.g., restart the terminated process.

The Erlang module system allows the user to divide a large program into a set of modules.

3. DESIGN OF THE CODE GENERATOR

3.1. General

A SDL/PR parser was generated with **yecc**, a LALR-(1) parser generator (similar to yacc) written in Erlang. Yecc produces Erlang code for the parser. The subset of SDL/PR grammar used was described in 250 lines of yecc. This subset is sufficient to process the examples given in this paper.

The code generator takes the parsed SDL/PR as input and generates the Erlang code in one pass. No optimization of the generated code has been done. The code has been generated in order to be readable. The code generator is also written in Erlang and consists of 1043 lines of Erlang code.

3.2. Mapping to SDL

There are a lot of similarities between SDL and Erlang. The concurrency of Erlang supports the concept of SDL concurrency. Processes in Erlang behave independently and asynchronously. The dynamic change of process sizes in Erlang makes it possible to use many processes and describe the application with no concern for implementation considerations. Erlang supports time and the asynchronous communication between Erlang processes is similar to the SDL communication mechanism. Erlang is also well suited for implementing extended finite state machines (EFSM).

In the following subsections the important design decisions will be described. The described primitives will be accessed through the **sdl** module.

3.2.1. The system

A SDL system is composed of blocks, channels and processes.

Channels

A channel between two blocks is implemented as an Erlang process called the *channel handler* for that particular channel. The receiving block initiates the channel handling process for a channel. Processes register themselves as being interested in signals. The channel handler directs each signal to the interested process. Only signals sent without receiving address are handled by a channel handler. All other signals are sent directly to the receiver from the sender.

All channels to/from the surrounding environment are implemented as one Erlang process. The environment channel handler is started during system startup. As above, all interested processes register themselves as possible receivers of signals.

Blocks

Each block in a system is represented by an Erlang module. Each module consists of the block initialization routines and the generated code for each specified process in the block diagram.

Signal routes, not connected to any channel, between processes in a block are handled as channels above. Signal routes connected to channels are handled by the corresponding channel handler.

Processes

In order to monitor the number of process instances in the system, there is a *process handler* process. In SDL one can specify the maximum number of simultaneous instances of a process. The `spawn_proc` function spawns a new process if the process handler allows it. When a process terminates it will be removed from the instance list of the process handler.

In SDL, every process has four implicitly predefined variables which are represented in Erlang as follows:

- **SELF** - The identification of the process itself is done by the Erlang BIF `self()` which returns the process identifier of the calling process.

- **PARENT** - The process identifier of the process which performed the create request will be stored in the variable `Parent`. If a process has been created by the system the `Parent` variable will have the value `novalue`.
- **OFFSPRING** - The process identifier of the last created process will be stored in the variable `Offspring`.
- **SENDER** - This variable will be bound whenever a signal has been recieved. It will be stored in the Erlang variable `Sender`.

System initialization

At system initialization the process handler and the environment channel handler are started. All blocks in the system are initiated, that is channel handlers for all channels directed towards a block are started. All processes specified to be present at system initialization are created.

Process initialization

At process initialization, signals received on channels or signal routes are registered, in the corresponding channel handler, for the process.

Default values are set to the process variables before the process starts the execution.

3.2.2. Signals

If the SDL output symbol does not specify the receiver of the signal, it will be sent to the corresponding channel handler. If the receiver is specified, the signal will be sent directly to the receiver. One primitive is provided to send signals:

- `send(To, Signal, Data)`

If the signal is to be sent to the channel handler, the `To` argument is the name of the channel handler. The channel handler name will be converted to a process identifier by the use of the Erlang BIF `register/2`.[2] If the destination process is known the `To` argument is the process identifier of the receiver.

SDL signals are represented as tuples, which are sent as ordinary Erlang messages:

```
{Sender, Signal, Pars}
```

where

- `Sender` - The process identity of the sending process.
- `Signal` - The name of the signal.
- `Pars` - A list of signal data.

[2]register(Name, Pid), registers a name, `Name`, for the process with `Pid` as process identifier.

3.2.3. Timers

Each declared timer in the system is implemented as an Erlang process. The timer processes are started when the corresponding (owning) process is initialized.

The timer process knows which name of the timeout signal to send and the identity of the receiving process.

The SDL operation NOW is interpreted as zero, i.e., the SDL expression SET(NOW + 3, `Time2Go`) results in the timeout signal `Time2Go` in 3000 ms. Two primitives are provided to manipulate timers:

- `setTimer(Timer, Time)` - Send a **set** message to the `Timer` process, ordering the timer to generate a timeout in `Time` ms.
- `resetTimer(Timer)` - Send a **reset** message to the `Timer` process and remove all corresponding timeout signals from the process message queue.

The code for the timer process is as follows:

```
timer(Name, Owner) ->                  % Init routine
  timer_loop(Name, Owner, infinity).

timer_loop(Name, Owner, Time) ->
  receive
    {set, NewTime} ->
      timer_loop(Name, Owner, NewTime);
    {reset} ->
      timer_loop(Name, Owner, infinity)
  after Time ->
    Owner ! {self(), Name, []},        % Send the timeout signal
    timer_loop(Name, Owner, infinity)
end.
```

When a timeout occurs a timeout signal, named `Name`, is sent to the `Owner` process. The atom `infinity`, in the example above, specifies an infinite delay time, i.e., no timeout will be generated.

When a process terminates all timers owned by that process also terminate.

3.2.4. Variables

The way SDL handles variables conflicts with the non-destructive assign semantics of Erlang variables. Several different interpretations can be considered.

In this implementation a variable is represented by a tuple, {`Variable, Value`}. A list containing the variable tuples of a process is maintained during the process state transitions.

Two primitives, `set` and `get`, are provided for variable manipulations:

- `set(Variable, Value, ListOfVars)` - Set the `Value` in the `Variable` tuple and return a new list of variables.
- `get(Variable, ListOfVars)` - Returns the `Variable` value.

3.2.5. States

Each state described in the systems process diagrams is represented by an Erlang function. State transitions are interpreted as function calls.

The functions consist of a `receive` statement with associated code for each input to the state. The `receive` statement could look like:

```
phone_Idle(VarList) ->
  receive
    {Sender, sig1, Pars} ->
      ...
      phone_Off_Hook(VarList); % State transition
    {Sender, sig2, Pars} ->
      ...
      phone_Ringing(VarList);  % State transition
    Other ->
      phone_Idle(VarList)      % Remain in this state
end.
```

The `Other` branch in the `receive` statement consumes all signal inputs not handled in the state. If all other signals were supposed to be saved, the `Other` branch would have been omitted. In that way all other signals remain in the process message buffer.

If, for example, only the signals `sigsave1` and `sigsave2` should be saved, the `Other` branch would look like:

```
phone_Idle(VarList) ->
  receive
    ...
    {Sender, Signal, Pars} when Signal =/= sigsave1,
                                Signal =/= sigsave2 ->
      phone_Idle(VarList)
end.
```

This branch will only match if `Signal` isn't `sigsave1` or `sigsave2`, i.e., `sigsave1` and `2` will remain in the message buffer while all other signals will be consumed.

4. THE DAEMON GAME EXAMPLE

The Daemon Game is a simple game suited to illustrate SDL specifications. It will also be an illustrative example of the Erlang generated code. The generated code can be immediately compiled and executed.

4.1. Purpose

The idea of the game is that somewhere in the system there is a daemon, producing outputs at random times. A player, located outside the system, tries to guess whether an even or an odd number of daemon outputs has occured. The player can probe when he believes that an odd number has occured. The player scores when an odd number has been hit.

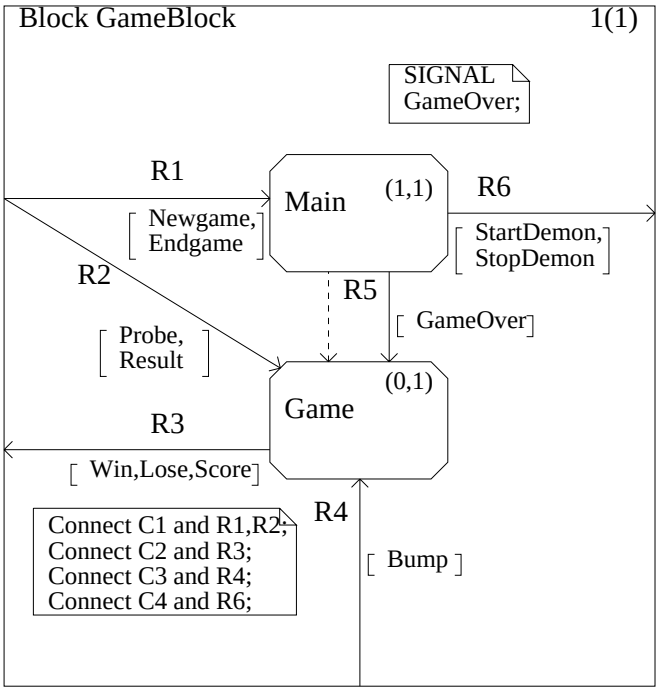

Figure 1. The GameBlock block SDL specification.

4.2. System level

The daemon game system consists of two blocks, GameBlock and DemonBlock. Two channels, *C1* and *C2*, connect the Gameblock with the surrounding environment, *C1* is an incoming channel and *C2* an outgoing one. The signals *Newgame*, *Probe*, *Result* and *Endgame* can be received on the *C1* channel. The system sends *Win*, *Lose* and *Score* on the *C2* channel. Two channels, *C3* and *C4*, connect the two blocks with each other. The *C3* channel is directed towards the GameBlock (with the *Bump* signal) while *C4* is directed towards the DemonBlock (with the *StartDemon* and *StopDemon* signals).

Three Erlang modules are generated for the system. One for each of the blocks and one for the system itself. The modules are called GameBlock, DemonBlock and DemonGame respectively. The DemonGame module performs the system initialization. Three channel handlers are generated, one for the channels connected to the environment and one for each of the *C3* and *C4* channels. The channel handlers will be registered with the global names `DemonGame.env`, `GameBlock.C3` and `DemonBlock.C4`.

4.3. The GameBlock

In this section the code generated for the GameBlock module is shown. The close mapping between the SDL descriptions and the generated code can be seen.

In Figure 1 the block diagram for the GameBlock is shown. One signal route, *R5*, is not connected to any channel. A channel handler, `Game.R5`, will be registered for this signal route.

At this level one can see that the Main process can receive the *Newgame* and *Endgame* signals. During the initialization of the Main process theese signals will be registered

for this process in the `DemonGame.env` channel handler. A Main process instance will be created during system startup and this will be the only instance of that process. The dynamic creation/deletion of the Game process will be monitored by the system process handler.

The SDL process diagram for the Main process is shown in Figure 2. The variable list of the process contains three variables, `GameP`, `Parent` and `Offspring`.

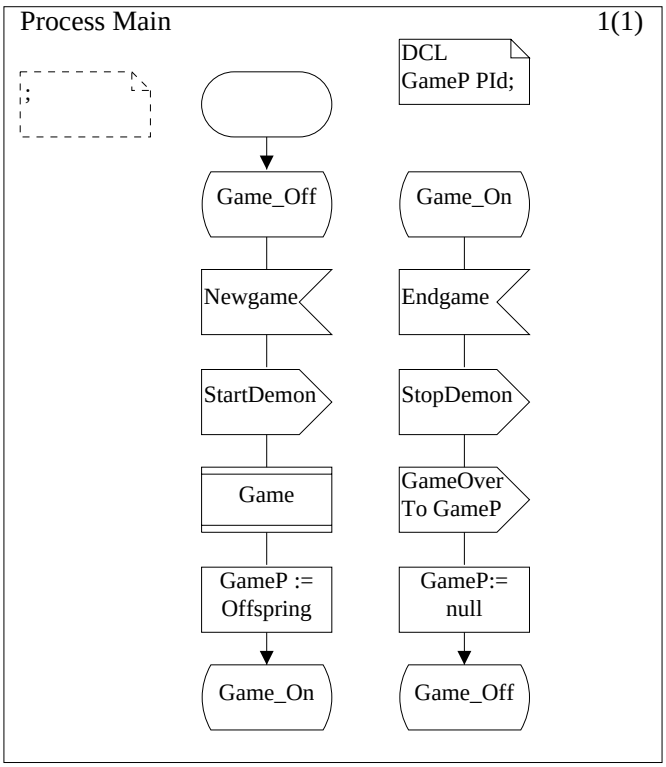

Figure 2. The Main process SDL specification.

In the `Game_On` state the difference between signal sending on a channel and direct signal sending can be seen. In the direct case the `GameP` variable will be used to address the receiver while the *StopDemon* signal will be sent to the process which has registered an interest in that signal to the corresponding channel handler.

```
'Main'(Parent) ->
  sdl:register_sig('DemonGame.env',['Newgame','Endgame']),
  'Main_init'([{'GameP',novalue},{'Parent',Parent},
               {'Offspring',novalue}]).

'Main_init'(Vars_0) ->
  'Main_Game_Off'(Vars_0).
```

```
'Main_Game_Off'(Vars_0) ->
  receive
    {Sender,'Newgame',[]} ->
      sdl:send('DemonBlock.C4','StartDemon',[]),
      Vars_1 = sdl:set('Offspring',
                       sdl:spawn_proc('DemonGame',
                                      'GameBlock',
                                      'Game',
                                      []),
                       Vars_0),
      Temp = sdl:get('Offspring',Vars_1,Local_1),
      Vars_2 = sdl:set('GameP',Temp,Vars_1),
      'Main_Game_On'(Vars_2);
    Other ->
      'Main_Game_Off'(Vars_0)
  end.

'Main_Game_On'(Vars_0) ->
  receive
    {Sender,'Endgame',[]} ->
      sdl:send('DemonBlock.C4','StopDemon',[]),      % Via channel
      sdl:send(sdl:get('GameP',Vars_0),'GameOver',[]), % Direct
      Vars_1 = sdl:set('GameP',null,Vars_0),
      'Main_Game_Off'(Vars_1);
    Other ->
      'Main_Game_On'(Vars_0)
  end.
```

Figure 3 shows the process diagram for the Game process. The asterisk (*) notation used means that in every state the Game process may receive the *Result* and *GameOver* signals. The *Result* and *GameOver* signals have been included for each state in the generated code. The code for the `Winning` and `Losing` states are similar, only the `Losing` state is shown below.

```
'Game'(Parent) ->
  sdl:register_sig('GameBlock.C3',['Bump']),
  sdl:register_sig('DemonGame.env',['Probe','Result']),
  sdl:register_sig('Game.R5',['GameOver']),
  'Game_init'([{'Count',novalue},{'Parent',Parent},
               {'Offspring',novalue}]).

'Game_init'(Vars_0) ->
  Vars_1 = sdl:set('Count',0,Vars_0),
  'Game_Losing'(Vars_1).

'Game_Losing'(Vars_0) ->
```

```
receive
  {Sender,'Probe',[]} ->
    sdl:send('DemonGame.env','Lose',[]),
    Temp = sdl:get('Count',Vars_0) - 1,
    Vars_1 = sdl:set('Count',Temp,Vars_0),
    'Game_Losing'(Vars_1);
  {Sender,'Bump',[]} ->
    'Game_Winning'(Vars_0);
  {Sender,'Result',[]} ->
    sdl:send('DemonGame.env','Score',[sdl:get('Count',Vars_0)]),
    'Game_Losing'(Vars_0);
  {Sender,'GameOver',[]} ->
    stop;
  Other ->
    'Game_Losing'(Vars_0)
end.
```

When the Game process receives the *GameOver* signal, the process terminates. All data related to the process instance are discarded by the Erlang system. The *process handler* is noticed through the Erlang error handling mechanism and deletes the process instance from the instance list.

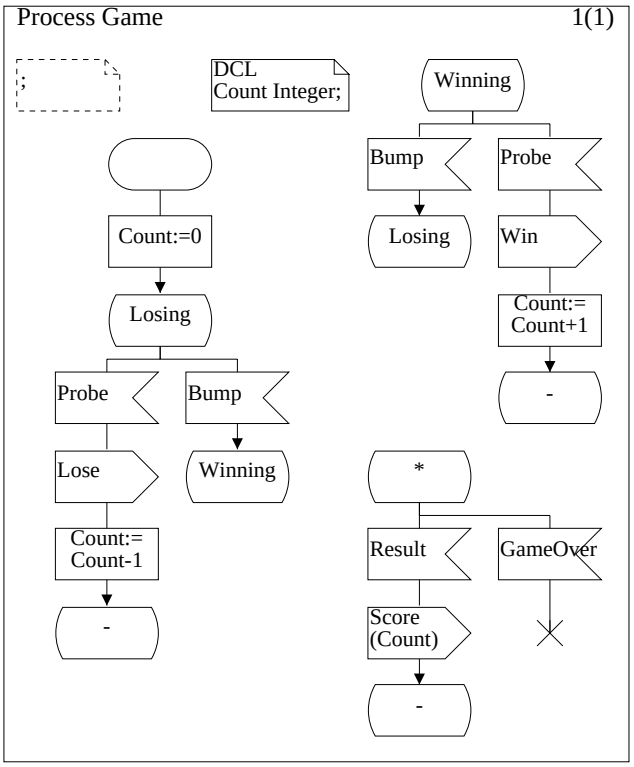

Figure 3. The Game process SDL specification.

5. DISCUSSION

In further experiments[3], we have investigated the transformation of POTS (Plain Ordinary Telephone Service) to Erlang. In the POTS example some more SDL constructs have been used, for example decisions, Syntypes, Newtypes, procedure calls and save. The code generated has been used to control real telephony hardware.

The results indicate that the number of generated Erlang lines of code are almost the same as the lines of SDL/PR generated by SDT. In the Daemon game example 190 lines of Erlang code were generated from 147 SDL/PR lines and in the POTS example it was 301 Erlang lines from 308 lines of SDL/PR.

The SDL approach with user-defined abstract data types can be tricky to generate automatically into Erlang. In the prototype only simple data types have been used.

The prototype shows that it is possible and easy to automatically generate Erlang code from SDL. The generated code can be run immediately. The subset of SDL handled by the code generator is growing.

Interestingly Erlang was used not only as the target language for the SDL transformation, but also as the language for the cross compiler itself and for all software tools associated with the process.

Finally, since we have not mentioned it elsewhere, it should be noted that recent implementations of Erlang approach the efficiency of (unoptimized) C [6].

REFERENCES

1. Functional Specification and Description Language (SDL), Recommendation Z.100 with annexes, Nov 1988.
2. Persson M., Ödling K. and Eriksson D., A Switching Software Architecture Prototype Using Real Time Declarative Language, XIV International Switching Symposium, Oct 26-30, 1992, Yokohama.
3. Danne A., Bauner J.-O. and Ahlberg I., Prototyping Cordless Using Declarative Programming, XIV International Switching Symposium, Oct 26-30, 1992, Yokohama.
4. Armstrong J. L. and Virding S. R., Erlang - An Experimental Telephony Programming Language, International Switching Symposium, May 27-June 1, 1990, Stockholm.
5. Armstrong J. L., Virding S. R. and Williams M. C., Concurrent Programming in Erlang, Prentice Hall, 1993.
6. Hausman B., Turbo Erlang: An Efficient Implementation of A Concurrent Programming Language, submitted to ILPS'93, International Logic Programming Symposium, Vancouver, Canada, 1993.
7. Belina F., Hogrefe D. and Sarma A., SDL with applications from protocol specification, Prentice Hall, 1991.

[3]Not described here for the reason of space - details are available from the author.