SDL versus C Equivalence Checking

Malek Haroud[1] and Armin Biere[2]

[1] STMicroelectronics NV
Advanced System Technology Group
Champ-des-filles 39, 1228 Geneva, Switzerland
[2] Johannes Kepler University
Institute for Formal Models and Verification
Altenbergerstr. 69, 4040 Linz, Austria

Abstract. We present a tool that automatically checks the existence of a bisimulation relation between an SDL specification and the corresponding auto-generated C code. The tool has been used to verify part of the C implementation of a WiFi Medium Access Controller (i.e.; IEEE 802.11) that has been derived from its original SDL specification using the Telelogic CAdvanced Code Generator.

1 Introduction

In embedded SW design, especially in the telecommunication field, the developer usually starts with a functional model written in SDL[19, 11, 15] or in any other similar high-level executable language. This model is extensively simulated, revised and sometimes model checked[18] until it becomes the golden reference model. In a second phase, this model is translated into an optimized implementation model, usually written in C [11, 9]. The translation is usually automatic using for instance compilers from SDL to C. Many companies still rely on manual translation for efficiency reasons with respect to speed, power or other technical issues.

Currently, the implementation model is simulated again and compared to the reference model to look for discrepancies. Unfortunately, simulation requires a great deal of time to set-up test benches. Additionally, simulation inputs are necessarily redundant at times and incomplete at others especially when concurrency features of the system are at stake.

The inefficiencies of simulation can mean dramatically higher costs, longer run times and persistent doubts [20]. As a direct consequence, verification becomes a very expensive process and is today swallowing up almost all resources and manpower. Our goal is to have a more efficient validation procedure than testing to assert the correctness of the implementation refinement. Moreover, verifying a code generator formally is very expensive since all the proofs have to be conducted all over again when the code generator changes whereas the validation we propose here occurs at each run of the compiler with a specific SDL program at hand. We present a tool called SCEC (i.e.; SDL C Equivalence Checker) that provides a fast and accurate validation of the C implementation derived from

SDL models. Currently our tool handles SDL'96 language constructs with few exceptions and targets mainly the Telelogic CAdvanced Tau 3.5 code generator. We argue that our approach is not limited to this version of SDL and to this particular tool, but can be applied to a broad class of asynchronous languages and compilers targeting imperative languages.

2 Related work

Originally applied to synchronous languages, the concept of *translation validation* was introduced by Pnueli, Siegel, and Shtrichman [16, 17] . Necula [14] generalizes the work by applying it to the verification of optimizing compilers. In the field of behavioral circuit description [4], C is verified against Verilog and uses Bounded Model Checking to verify the consistency [2, 3] between the two descriptions.

In the SDL context, [8] proposes a method to check refinements between SDL models by translating them into a process algebra formalism called CCS[12]. The main problem is that all the data part is abstracted away and the translation leads to overly simplistic CCS models. Our approach is more general in the sense that it addresses the implementation language and handles both the control and data flows[10, 13]. Moreover, we propose a practical equivalence model for asynchronous languages in general.

3 SCEC tool

SCEC is a tool that has been developed in ANSI C (18'000 lines) together with Flex and Bison generators to produce the scanners and the parsers for C and SDL. A WxWidgets based graphical user interface has been developed in order to browse the intermediate representation of the programs. Starting from the syntax tree, SCEC can record all the transformations that are applied on the trees up to the final normal form. This feature was valuable to debug the tool itself.

4 Flow

SCEC (cf. figure 1) generates the Abstract Syntax Tree for both SDL and C programs and performs standard semantic analysis. For the C part, AST comprises data type definitions, global data declarations, and complete function bodies,whereas for the SDL part, SCEC stores type definitions, signals and process bodies. The ASTs are gradually transformed using rewrite rules. Some of them are generic while others are specific to the CAdvanced code generator.

At the end of the rewrite process, we obtain on the one hand, a number of state transition graphs representing the SDL processes and on the other hand, the corresponding C functions (i.e.; yPADs) that implement them. All SCEC has to do, is to compare the SDL processes and the yPADs pairwise.

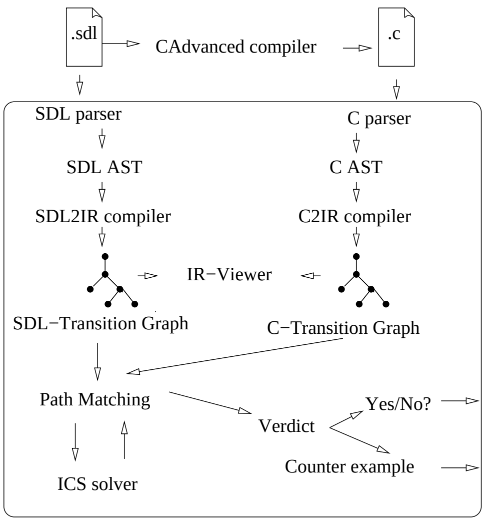


Fig. 1. SCEC Flow

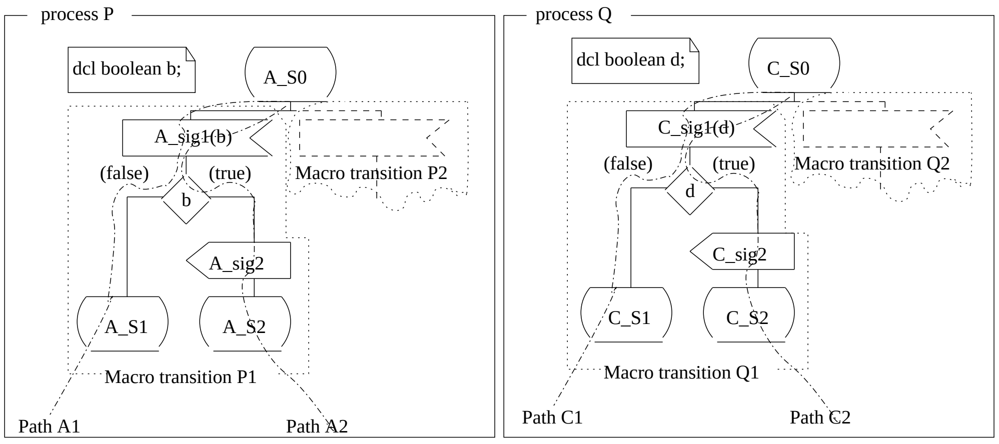


Fig. 2. Path matching

5 Process and yPAD correspondence

A yPAD (cf. figure 3) is a C function that defines all the transitions of the related SDL process. The yPAD is called by the Telelogic CAdvanced scheduler that controls the pseudo parallel execution of the communicating state machines. When the head of the signal queue contains a signal instance that can be consumed in the current state of the associated state machine, the scheduler fires the associated yPAD that will run one selected transition completely. The exe-

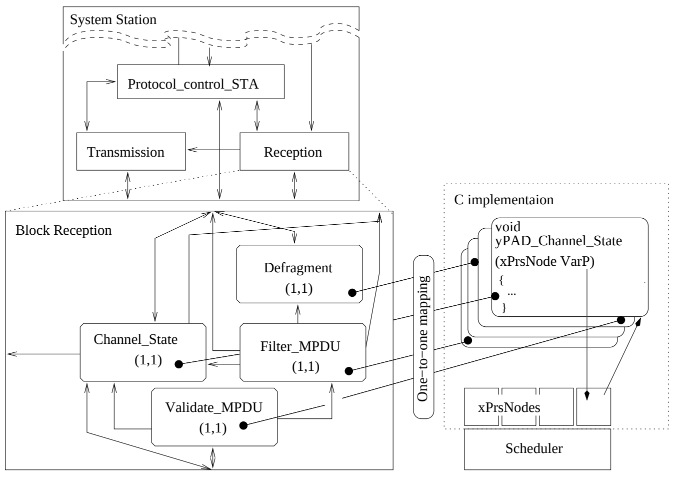


Fig. 3. Process yPAD correspondence

cution control returns back to the scheduler after changing the state of the last fired yPAD function.

6 Path matching concept

To grasp the concept of path matching, let us consider the two SDL processes P and Q that are depicted in figure 2. Starting from state A_S0, P can either execute the path A1: $state(A\_S0), input(A\_sig1(b)), guard(\neg b), state(A\_S1)$ or the path A2: $state(A\_S0), input(A\_sig1(b)), guard(b), output(A\_sig2), state(A\_S2)$. A path represents one transition from one state to its successor. We refer in the following to a *path* with the term *micro transition.* Moreover, a group of micro transitions under the same signal input are structured further to form a *macro transition.* Now, P and Q are considered to be equivalent if the paths A1 and A2 can be matched with the paths C1 and C2 respectively. Basically, if P and Q have an identical internal state and they both consume the same signal instance, then, at the end of the matching paths, they will have modified their internal state in the same way and they will have output the same signal instances. In our case, we need to compare an SDL process to an yPAD function that is why SCEC has to align the SDL and C internal representations by regenerating the original SDL process from the yPAD function. To cope with the combinatorial explosion of paths, *cut points* are introduced at four levels (cf. figure 4). These cut points are used by our approach to establish a formal correspondence between two descriptions. By restricting the type, abstraction level and number of cut points considered, we help the tool to establish the correspondence, since fewer pairs of cut points have to be checked. On the other hand this implies less verifiable but equivalent programs. However, no automatic tool can be expected to be able to check all equivalent programs completely, since in general translation validation and software equivalence checking are undecidable problems. One of our main contributions is to list those potential cut points that allow to verify equivalence in practice.

7 Cut points

We assume that the compiler or the developer respects some naming convention that will allow SCEC to establish correspondence between cut points in order to prove equivalence of the two descriptions. There are four levels of cut points:

1. process name versus yPAD function names.
2. state and connection names.
3. label names defining termination points of control edges.
4. Macro transition names.

A macro transition start with one of the following:

– an input signal.
– an enabling condition.
– a continuous signal.

A free action identified by the SDL keyword *connection* allows to split the graphical representation of an SDL process so that it can span over more than one page. We use the label present in the *in* and *out* connector as a cut point. In addition, we exploit the fact that any control edge that the user defines when drawing the SDL process will appear in the form of a *join* statement to a label defining the termination point of that control edge. This means that all the loops are cut allowing SCEC to reduce loop equivalence problems to path equivalence problems.

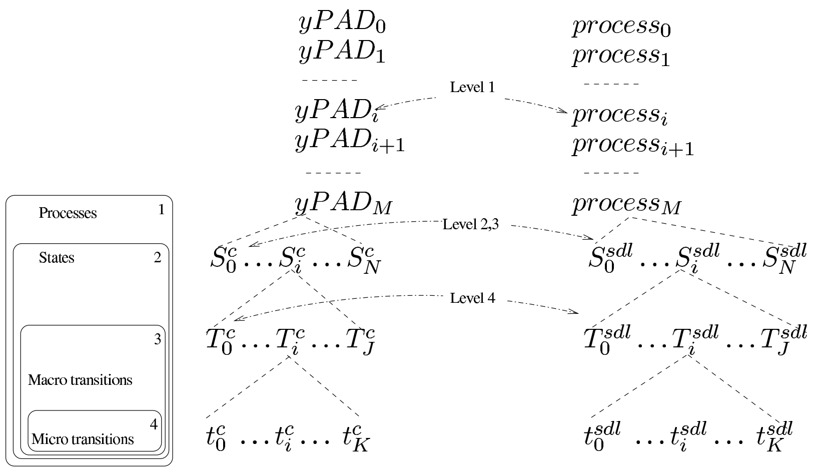
8 Code generator assumptions

A yPAD does not contain enough information needed for SCEC to regenerate the finite state machine. In fact, we still need to understand the interface between the yPAD and the scheduler. The scheduler needs to store the execution context information (i.e.; xPrsNode) and may shift some information that lies originally in the SDL process definition, out of the yPAD in order to avoid firing idle processes. Therefore, SCEC needs to analyze the xPrsNode structure as well. The xPrsNode contains:

– A list of input signals denoted xInputSignals.
– A list of states occurring in the SDL process denoted xStateIdStruct.

Each xStateIdStruct element contains the following:

– A macro transition type table yStaH.



$yPAD_0$
$yPAD_1$
$yPAD_i$
$yPAD_{i+1}$
$yPAD_M$
$process_0$
$process_1$
$process_i$
$process_{i+1}$
$process_M$
Level 1
Level 2,3
Level 4
$S_0^c \dots S_i^c \dots S_N^c$
$S_0^{sdl} \dots S_i^{sdl} \dots S_N^{sdl}$
$T_0^c \dots T_i^c \dots T_J^c$
$T_0^{sdl} \dots T_i^{sdl} \dots T_J^{sdl}$
$t_0^c \dots t_i^c \dots t_K^c$
$t_0^{sdl} \dots t_i^{sdl} \dots t_K^{sdl}$
Processes 1
States 2
3
Macro transitions
Micro transitions 4

Fig. 4. Modular verification using cut points

- A transition table called yStaI.
- A reference to enabling conditions denoted yEnab (optional).
- A reference to continuous signals denoted yCont (optional).

This concludes the list of information that has to be extracted by SCEC to regenerate the original SDL process. In the next section we precisely define the kind of equivalence we are referring to.

9 Equivalence Relation

We assume that both the SDL model and its C implementation can be compiled into a normal form that we call a *process network*. Process networks can be compared using an *equivalence* relation.

Proposition 1. *Any process (i.e.; extended finite state machine) can be transformed into a* state transition graph *such that each micro transition is represented by:*

- *a sequence of terms built over the local data.*
- *a path predicate defining under which control condition that path is followed.*

Proposition 2. *Each micro transition in the state transition graph is closed with either a nextstate statement or with a join statement referring to a connection name.*

9.1 Equivalence between two state transition graphs

Definition 1. *Let f and g two terms in a micro transition (i.e.; path). We say that f is equivalent to g written $f \equiv g$ iff f is structurally identical to g.*

Definition 2. *Let t_i and t_j two micro transitions. We say that t_i is equivalent to t_j written $t_i \approx t_j$ iff:*

- *t_i and t_j contain equivalent sequence of terms.*
- *the guards in t_i and t_j are logically equivalent.*
- *the data and control dependencies between terms and guards are preserved.*

Definition 3. *Let $G_{sdl} = \langle S^{sdl}, s_0^{sdl}, \longrightarrow \rangle$ and $G_c = \langle S^c, s_0^c, \longrightarrow \rangle$ two state transition graph. G_c simulates G_{sdl} if it exists a binary relation $\sim \subseteq S^{sdl} \times S^c$ such as:*

- $\forall s_{sdl} \in S^{sdl}, \exists s_c \in S^c : s_{sdl} \sim s_c$
- $s_{sdl} \sim s_c \wedge s_{sdl} \xrightarrow[t_{sdl}]{} s'_{sdl} \Rightarrow \exists s'_c \in S_c : s_c \xrightarrow[t_c]{} s'_c \wedge s'_{sdl} \sim s'_c \wedge t_{sdl} \approx t_c$

if G_{sdl} simulates G_c via $\sim^{-1}$ then $\sim$ is a bisimulation.

Proposition 3. *If two state transition graphs can be reduced to the same state transition graph S_3 then S_1 bisimulates S_2*

In fact, each rewrite rule performed by SCEC is a reduction. Therefore, if after composing a number of reductions on the C state transition graph and on the SDL state transition graph we can reach the same transition graph then we can conclude using proposition 2 that there is a bisimulation between the C and the SDL.

9.2 Process network equivalence

Definition 4. *A* process network *is a set of processes that communicate with each other and with the environment asynchronously using signals and queues.*

Assume we have two isomorphic process networks PN_{SDL} and PN_C such that related components are equivalent in the sense of definition 3. If we compose components of PN_{SDL} and PN_C with the same deterministic scheduler and with the same environment then we can conclude that PN_{SDL} and PN_C are also equivalent. We are definitely in the case of bottom up compositionality principle of *components-based design* defined in [5].

9.3 C to IR translation

Translating SDL into the intermediate form was straightforward, since by construction IR was built in such way, that it subsumes a low level representation of SDL models. For the C part it was less obvious. As a general principle, we have chosen to unify the concepts of both languages instead of reducing them to atomic statements that would have made the correspondence almost infeasible [6].

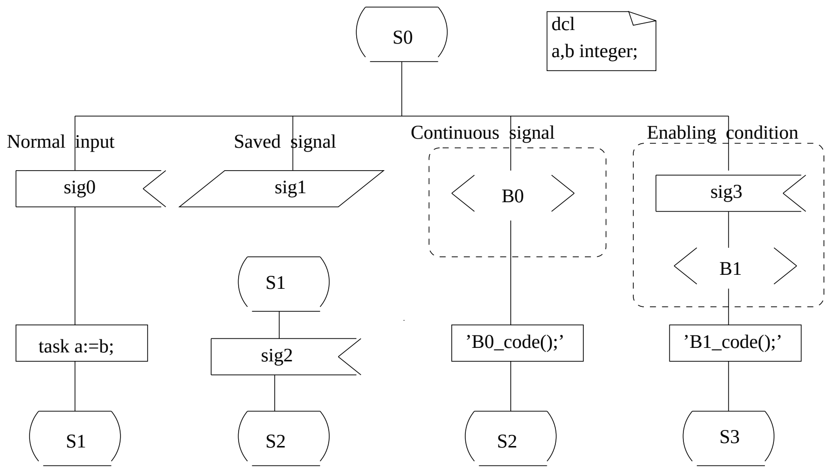


Fig. 5. SDL program fragment

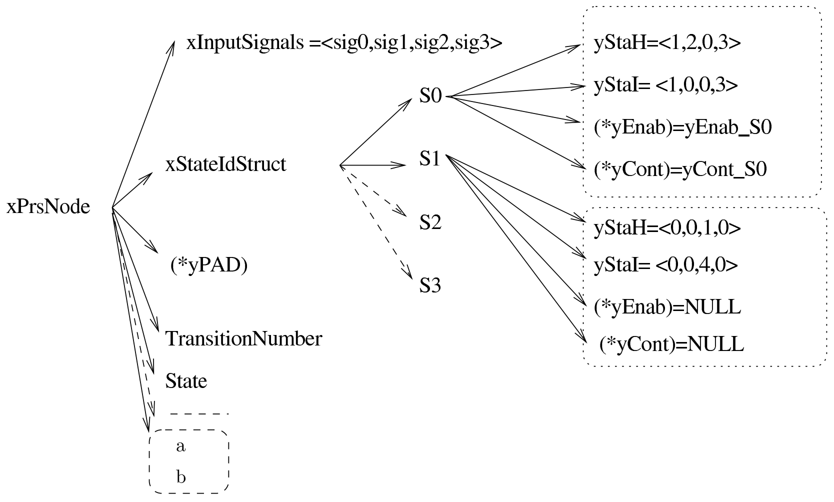
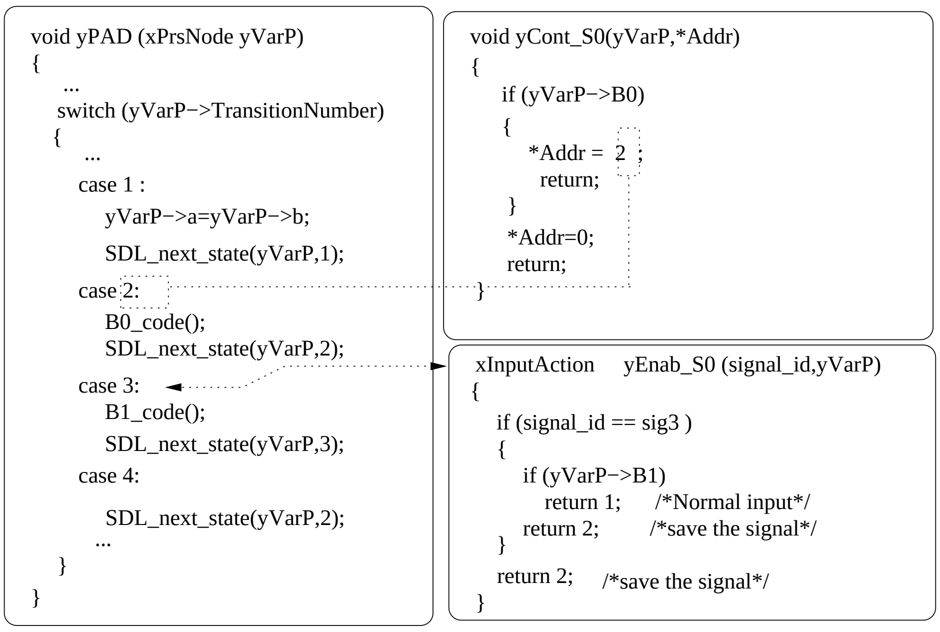


Fig. 6. xPrsNode structure (cf. figure 5)

yStaH value	Interpretation
0	unexpected signal
1	normal input
2	saved input
3	enabling condition

Table 1: yStaH interpretation



```
void yPAD (xPrsNode yVarP)
{
    ...
   switch (yVarP−>TransitionNumber)
   {
      ...
     case 1 :
        yVarP−>a=yVarP−>b;
        SDL_next_state(yVarP,1);
     case 2:
        B0_code();
        SDL_next_state(yVarP,2);
     case 3:
        B1_code();
        SDL_next_state(yVarP,3);
     case 4:

        SDL_next_state(yVarP,2);
       ...
   }
}
```

```
void yCont_S0(yVarP,*Addr)
{
    if (yVarP−>B0)
    {
       *Addr =  2 ;
         return;
    }
    *Addr=0;
    return;
}
```

```
xInputAction     yEnab_S0 (signal_id,yVarP)
{
    if (signal_id == sig3 )
    {
       if (yVarP−>B1)
          return 1;     /*Normal input*/
       return 2;        /*save the signal*/
    }
    return 2;     /*save the signal*/
}
```

Fig. 7. yPAD, yEnab and yCont correspondence

state	event	transition
S0	sig0	1
S0	B0	2
S0	sig3	3
S1	sig2	4

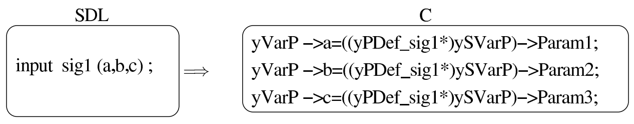
Table 2: Transition matrix

In the following, we present some elements describing how the SDL process represented in figure 5 is regenerated from the components depicted in figure 6 and figure 7.

- **Local data retrieval**: SCEC dereferences a pointer to an xPrsNode (cf. figure 6) passed as parameter to the yPAD and then extracts the integer fields a and b representing local data definitions.
- **Transition number resolution**: This is done by looking up the yStaI tables to determine to which state and macro transition it corresponds. For example, the transition number 4 (cf. figure 6) occurs in the yStaI list that belongs to the state S1. Moreover, the position of the transition number 4 in yStaI list corresponds to the position of sig2 in the xInputSignals list. At last, to determine the macro transition type related to sig2, SCEC looks up yStaH (cf. table 1) at the position of sig2 in xInputSignals list and infers that

it is a normal signal input. The complete transition matrix of the process depicted in figure 5 is given in table 2.

- **Next state name regeneration** The name corresponds to the element of xStateIdStruct list that is indexed by the second parameter passed to the SDL_next_state function. For instance, SDL_next_state(yVarP,1) corresponds to nextstate(S1).
- **Input reconstruction**: Figure 8 illustrates how an SDL input statement is translated to C. In fact, the scheduler pass to the yPAD a pointer to the received signal (i.e.; ySVarP) before firing the transition. This pointer is converted to the type of the signal corresponding to the selected transition. Signal parameters are then stored to the local data of the SDL process.

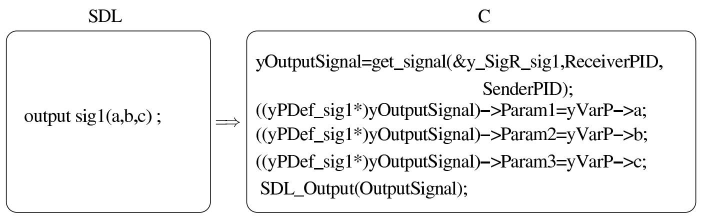


```
input sig1 (a,b,c) ;
```

```
yVarP –>a=((yPDef_sig1*)ySVarP)–>Param1;
yVarP –>b=((yPDef_sig1*)ySVarP)–>Param2;
yVarP –>c=((yPDef_sig1*)ySVarP)–>Param3;
```

Fig. 8. SDL input statement translation

- **Output reconstruction** The sending process allocates the necessary storage to hold the signal instance at the receiver input queue using the get_signal function. The returned pointer yOutputSignal is used then to build the actual parameters of the signal from the local data fields (cf. figure 9).



```
output sig1(a,b,c) ;
```

```
yOutputSignal=get_signal(&y_SigR_sig1,ReceiverPID,
                                        SenderPID);
((yPDef_sig1*)yOutputSignal)–>Param1=yVarP–>a;
((yPDef_sig1*)yOutputSignal)–>Param2=yVarP–>b;
((yPDef_sig1*)yOutputSignal)–>Param3=yVarP–>c;
 SDL_Output(OutputSignal);
```

Fig. 9. SDL output statement translation

- **Saved signal set reconstruction**: Basically all the signals that have the value 2 in the yStaH list are saved in the context state in which they appear. For example, sig1 which is located at the second position in the xInputSignals list is saved in state S0 (cf. figure 6).
- **Enabling condition regeneration**: By parsing the yEnab function referenced in xPrsNode, SCEC extracts the guard associated to the signal. For instance, sig3 is guarded by the expression B1 at state S0. When the guard evaluates to false, the signal is saved.

– **Continuous signal regeneration**: The yCont function body referenced in xPrsNode contains the boolean condition B0 and the transition number to be fired in case the condition is fulfilled.

10 A concrete example

In the following subsections, we show how a concrete SDL example is translated into C in order to figure out the kind of transformations that are applied by SCEC to align the two internal representations.

10.1 SDL transition definition

The SDL process is represented in the yPAD function by a switch case statement over the transition number. For instance, the state From_LLC together with the signal input MaUnitdata.request (i.e.; lines 1 and 2) is mapped onto transition number 1(i.e.; line 102). The case statement is immediately followed by the process local data update statements. In fact, all the parameters conveyed in the input signal pointer are copied into the corresponding local data using the reference yVarP that points to the xPrsNode structure (cf. figure 8).

```
1  state From_LLC;
2  input MaUnitdata.request(sa, da, rt, LLCdata, cf, srv);
```

```
100     switch(yVarP->TransitionNumber)
101     {
102       case 1:
103         yAss_z0A_octetstring (&(yVarP->z0017_sa),
104                              ((yPDef_z02_MaUnitdatarequest *)
105                               ySVarP)->Param1,0);
106         yAss_z0A_octetstring   (&(yVarP->z0018_da),
107                             ((yPDef_z02_MaUnitdatarequest*)ySVarP)->Param2,0);
108         yVarP->z0016_rt = ((yPDef_z02_MaUnitdatarequest *)ySVarP)->Param3;
109         yAss_z0A_octetstring (&(yVarP->z0015_LLCdata),
110                             ((yPDef_z02_MaUnitdatarequest*)ySVarP)->Param4,0);
111         yVarP->z0014_cf = ((yPDef_z02_MaUnitdatarequest*)ySVarP)->Param5;
112         yVarP->z001A_srv = ((yPDef_z02_MaUnitdatarequest*)ySVarP)->Param6;
```

10.2 SDL conditional assignment

A transition contains typically a sequence of actions to be performed when it is fired. The transition presented in subsection 8.1 is followed by a conditional SDL assignment (i.e.; lines 3 to 9).

```
3      task stat :=
4          if rt /= null_rt then
5            nonNullSourceRouting
6          else if (length(LLCdata) > sMsduMaxLng)
7                   or (length(LLCdata) < 0) then
8            excessiveDataLength
9          else successful fi fi;
```

The CAdvanced code generator preserves the structure of the assignment (i.e.; lines 113 to 116) which allows SCEC to do a simple structural term comparison instead of adding another factor in the number of paths to be matched. Note in passing that the SDL synonyms resolution rewrite is necessary before unifying the two assignment terms.

```
113  yVarP->z001B_stat = ((yVarP->z0016_rt) != (0) ? 5 :
114    (((((z0M1M_length (yVarP->z0015_LLCdata)) > (5678)))
115     ||
116    ((((z0M1M_length (yVarP->z0015_LLCdata))<(0)))))?4:0));
```

10.3 SDL decision

The SDL decision (i.e.; lines 10 to 29) comprises four micro transitions closed with a *join* statement.

```
10      decision stat = successful;
11      (true) :
12        decision srv;
13        (strictlyOrdered) :
14          decision
15            import(dot11PowerManagementMode);
16          (sta_active) :
17          else :
18            task stat := unavailableServiceClass;
19            join grst29;
20          enddecision;
21        (reorderable) :
22          join grst28;
23        else :
24          task stat := unsupportedServiceClass;
25          join grst29;
26        enddecision;
27      (false) :
28        join grst29;
29      enddecision;
```

In the generated C code, the translation reflects the same branching structure as in the SDL code and simply converts *join* statements into *goto* statements. In a manual translation, we would rather find a function call when the label is referring to a free action or the introduction of an equivalent C iteration statement in case of looping. In both cases, the cut points could still be derived automatically.

```
117      if ((yVarP->z001B_stat) == (0))
118        {
119          yVarP->yDcn_z08_ServiceClass = yVarP->z001A_srv;
120          if ((yVarP->yDcn_z08_ServiceClass) == (1))
121            {
122              if (((*(z00_PwrSave*)
       xGetExportAddr(
        &yReVR_z001H_dot11PowerManagementMode,
        xSysD.SDL_NULL_Var, (int) 0,
        VarP))) == (0))
    {
    }
   else
    {
     yVarP->z001B_stat = 9;
     goto L_grst29;
    }
   }
  else if ((yVarP->yDcn_z08_ServiceClass) == (0))
    goto L_grst28;
  else
   {
    yVarP->z001B_stat = 8;
    goto L_grst29;
   }
 }
else
  goto L_grst29;
```

11 Path matching

To establish the correspondence between SDL and C paths, SCEC needs to match terms structurally. For example, the SDL terms defined between line 10 and 12 can be matched with their corresponding C terms defined between line 23 and 25. A path contains also guards representing the chosen alternatives when conditional statements are met along the macro transition. To cope with guards and auxiliary variables, SCEC relies on an external solver to verify that the conjunction of guards on both sides are indeed equivalent. An example of query is given in the next subsection.

```
1    SDL_path_ns_RXC_Idle(
2     guard(or(ftype(pdu)=reasoc_rsp,ftype(pdu)=asoc_rsp,
3             ftype(pdu)=reasoc_req,ftype(pdu)=asoc_req,
4             ftype(pdu)=disasoc,ftype(pdu)=null_frame)),
5     guard(or(sau=1,sau=2)),
6     guard(or(ftype(pdu)=reasoc_rsp,ftype(pdu)=asoc_rsp,
7             ftype(pdu)=reasoc_req,ftype(pdu)=asoc_req,
8             ftype(pdu)=disasoc,ftype(pdu)=null_frame)),
9     guard(not(ftype(pdu)=null_frame)),
10    output(MmIndicate(pdu,endRx,strTs,0)),
11    label(grst50),
12    nextstate(RXC_Idle))

13   C_path_ns_RXC_Idle(
 assign(z13_TypeSubtype,ftype(pdu)),
 guard(or(or(or(or(or(z13_TypeSubtype=null_frame
                     ,z13_TypeSubtype=disasoc)
                     ,z13_TypeSubtype=asoc_req)
                     ,z13_TypeSubtype=reasoc_req)
                     ,z13_TypeSubtype=asoc_rsp)
                     ,z13_TypeSubtype=reasoc_rsp)),
 guard(or(sau=1,sau=2)),
 guard(not(ftype(pdu)=null_frame)),
 output(MmIndicate(pdu,endRx,strTs,0)),
 label(grst50),
 nextstate(RXC_Idle))
```

12 Solver Invocation

The ICS [7] solver is particularly useful to cope with auxiliary variables added in the generated C code(i.e.; SDL decision) and also to remove the redundant clauses that are added by the path extractor algorithm. Basically, if $\neg(Path_{sdl} \iff Path_C)$ is unsatisfiable then $Path_{sdl} \iff Path_C$ is valid. The following ICS code represents an SCEC satisfiability query.

```
def z13_TypeSubtype := ftype(pdu).
prop c_path :=    [[[[[z13_TypeSubtype=null_frame
                       |z13_TypeSubtype=disasoc]
                       |[z13_TypeSubtype=asoc_req]]
                       |[z13_TypeSubtype=reasoc_req]]
                       |[z13_TypeSubtype=asoc_rsp]]
                       |[z13_TypeSubtype=reasoc_rsp]]
                       & ~[z13_TypeSubtype=null_frame]
                  & [sau=1 | sau=2].
prop sdl_path :=  [ftype(pdu)=reasoc_rsp|ftype(pdu)=asoc_rsp
                   |ftype(pdu)=reasoc_req |ftype(pdu)=asoc_req
                   |ftype(pdu)=disasoc|ftype(pdu)=null_frame]
                   &~[ftype(pdu)=null_frame]
                   & [sau=1 | sau=2]
                   & [ftype(pdu)=reasoc_rsp |ftype(pdu)=asoc_rsp
                      |ftype(pdu)=reasoc_req |ftype(pdu)=asoc_req
                      |ftype(pdu)=disasoc  |ftype(pdu)=null_frame]
                   & ~[ftype(pdu)=null_frame].
prop path_eq:=[~c_path|sdl_path]&[~sdl_path|c_path].
sat ~path_eq.
```

13 Results

The 802.11 MAC layer is IEEE standardized. The original SDL diagrams (i.e.; 4'000 lines) came from the specification [1] and were automatically translated to C (i.e.;17'000 lines) using the CAdvanced 3.5 compiler. Using SCEC and ICS, the whole verification process takes less than one minute on an Intel Centrino 1.5 GHz, since most of the verification conditions turn out to be trivial after extracting the proper cut points. Figure 10 shows statistics extracted from the intermediate representation. We can see clearly that the number of micro transitions does not exceed one hundred paths for the biggest process (i.e.; TX coordination). This is due to the fact that free action cut points allowed to factor out all the paths that precedes the *join* statements and therefore reduce drastically the number of paths to be matched. To check the soundness of SCEC, we injected random defects into the correctly generated C code. Our tool found these inconsistency instantly.

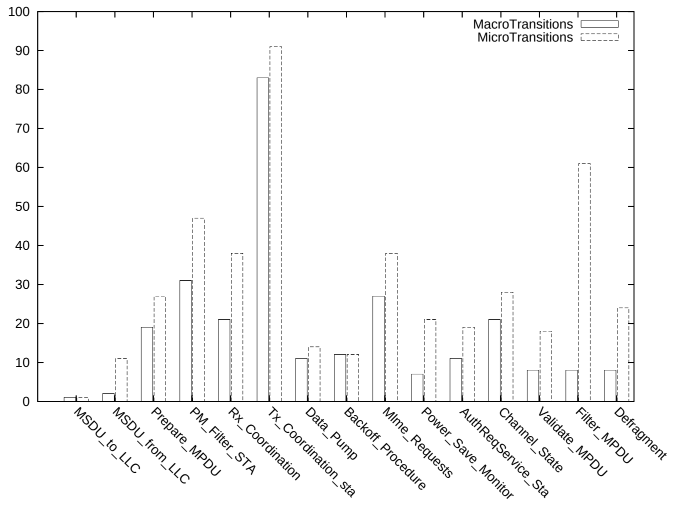


Fig. 10. Macro and micro transition statistics

14 Conclusion

We have described a practical method to check the equivalence between real world SDL programs and their corresponding auto-generated C code. One key feature is the full automation of the process. The SDL and C programs are translated into a common intermediate representation for which we presented a bisimulation equivalence argument. The translation into the intermediate form is done by applying specific rewrite rules that capture the FSM encoding method and the optimizations done by the compiler. Our method was successful in validating the translation of a commercial compiler and should be certainly very useful for checking manually translated code. Our plans for future include the integration of the Telelogic CMicro compiler which targets embedded applications and also to provide the user with a better diagnosis capability for failed proof attempts.

References

1. Wireless LAN Medium Access Control and Physical Layer specifications High-speed Physical Layer in the 5 GHz band. *IEEE specification*, 1999.
2. A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Bounded Model Checking*, volume 58 of *Advances in Computers*. Academic Press, 2003.
3. E. Clarke, A. Beire, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design (FMSD)*, 19(1), 2001.
4. E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and verilog programs using bounded model checking. In *Proc. 40th Conf. on Design Automation (DAC'03)*. ACM, 2003.
5. L. de Alfaro and T. Henzinger. Interface theories for component-based design. In *EMSOFT 01: Embedded Software*, Lecture Notes in Computer Science 2211, pages 148–165. Springer-Verlag, 2001.
6. P. Ellervee, S. Kumar, A. Jantsch, B. Svantesson, T. Meincke, and A. Hemani. IRSYD: An internal representation for heterogeneous embedded systems. In *Proc. 16th NORCHIP Conference*, 1998.
7. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonizer and Solver. *Computer Aided Verification Conference*, 2001.
8. M. C. Hai. Bisimulation Analysis of SDL-Expressed Protocols: A Case Study. *CASCON Conference*, 2004.
9. Haroud, Blažević, and Biere. HW accelerated Ultra Wide Band MAC protocol using SDL and SystemC. *Radio And Wireless Conference*, 2004.
10. S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proc. of 15th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'88)*. ACM, 1988.
11. M. Hnnikinen, A. Takko, J. Knuutila, T. Hmlinen, and J. Saarinen. SDL-to-C conversion for implementing embedded WLAN protocols. In *Intl. Conf. on Industrial Electronics, Control, and Instrumentation (IECON'00)*. IEEE, 2000.
12. R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.
13. R. Namballa, N. Ranganathan, and A. Ejnioui. Control and data flow graph extraction for high-level synthesis. In *Proc. IEEE Computer Society Annual Symp on VLSI Emerging Trends in VLSI Systems Design (ISVLSI'04)*. IEEE Computer, 2004.
14. G. C. Necula. Translation validation for an optimizing compiler. In *Proc. ACM SIGPLAN 2000 Conf. on Programming Language Design and Implementation (PLDI'00)*. ACM, 2000.
15. A. Olsen. *Systems Engineering Using SDL-92*. Elsevier, 1994.
16. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, 1998.
17. A. Pnueli, O. Strichman, and M. Siegel. Translation validation: From SIGNAL to C. In *Correct System Design, Recent Insight and Advances*, volume 1710 of *Lecture Notes in Computer Science*, pages 231–255. Springer, 1999.
18. N. Sidorova and M. Steffen. Verifying large SDL-specifications using model checking. In *Proc. 10th Intl. SDL-Forum (SDL'01)*, volume 2078 of *Lecture Notes in Computer Science*. Springer, 2001.
19. J. Sipilä and V. Luukkala. An SDL implementation framework for third generation mobile communications system. In *Proc. 10th Intl. SDL-Forum (SDL'01)*, volume 2078 of *Lecture Notes in Computer Science*. Springer, 2001.
20. J. A. Whittaker. What is software testing, and why is it so hard. *IEEE Software*, 17(1), 2000.