SDLcheck: A Model Checking Tool

Vladimir Levin and Hüsnü Yenigün

Bell Laboratories, Lucent Technologies, Murray Hill, NJ 07974
{levin,husnu}@research.bell-labs.com

Introduction

SDLcheck is a verification tool developed to support model checking for asynchronous (concurrent) programs written in SDL [1,2]. Given an SDL program and a specification of a desired behavior of the program, SDLcheck generates a verification model that consists of two ω-automata, P and T: P models the program and T the specification. Then, the automaton language containment, $L(P) \subset L(T)$, is tested by model checking with Cospan [3].

The majority of model checking tools designed for asynchronous program verification make use of interleaving systems as a model platform. In contrast, SDLcheck translates asynchronous SDL programs into synchronous ω-automata. Concurrent execution (interleaving) of SDL processes is modeled using a simple technique described below in the paper. The reason for this choice is in order to efficiently combine partial order reduction, which is known to be useful for asynchronous programs, with BDD-based symbolic verification, which is known to be useful for large synchronous models. For this, SDLcheck implements the algorithm described in [4] that realizes partial order reduction through modifying a program model P prior to model checking. Although model checking tools for SDL and other programming and design languages are being intensively developed in research, in a practical sense, they mostly remain prototypes lacking optimizations necessary to cope with large programs. There are several advanced model checking tools that mainly relate to hardware verification, where synchronous ω-automata, on one hand, naturally match synthesizable hardware designs and, on the other hand, support symbolic verification. For software verification, combining IF [5] and SPIN [6], as reported in [7], supports complementary sets of model checking optimizations. This combination nonetheless lacks symbolic verification, as do all other SDL verification tools of which we are aware.

SDLcheck is also capable of supporting software/hardware co-design verification. This is realized through Cospan, which is also used as the model checker in hardware verification, namely, in the commercial tool FormalCheck$^{\text{TM}}$. [1] Through the synchronous ω-automaton model platform, SDLcheck combined with Cospan supports both software specific and hardware specific model checking optimizations.

SDL Subset and Co-design Extensions

SDLcheck accepts the SDL'96 standard [1,2] including ASN.1 related features, however, without axiomatic data definitions, services and OO features. It also requires the SDL program model to be finite state — so no unbounded recursion.

[1] licensed by Lucent Technologies to Cadence Design Systems.

To support co-verification, SDLcheck implements extensions to SDL (suggested in [8]) that allow description of a software process interfacing to a hardware module. The hardware part of a co-design is expressed in a hardware description language, Verilog or VHDL. On the software side written in SDL, SDLcheck supports read/write access to a hardware variable (wire or port) through the declaration of an associated *interface* variable. The interface variable is either sourced from, or feeds the hardware variable. SDLcheck also supports a `none` input action. It does not read the process buffer and only triggers a transition from the current state of the process when the enabling condition which guards this action evaluates to true. A `none` input action matches well the concept of a hardware transition triggered by an event, such as clock rising (or falling) or signal reset. Being associated with an interface variable value, a hardware event, say, value 1 on wire $A.B.y$, may be tested in the enabling condition and trigger a transition in the corresponding software process. Once triggered, this transition executes like a hardware transition: synchronously (simultaneously) with all enabled transitions of the co-design hardware part. Other (*software*) transitions of software processes execute asynchronously according to usual SDL rules.

Verification Technology and the Tool Architecture
SDLcheck performs three steps:

1. The compiler `sdl2sr` translates both an SDL program and a behavior specification into S/R, the input language of Cospan. The specification is expressed using macro notations $always(x)$, $eventually(x)$, etc. that reflect linear temporal logic operators and useful combinations of those, with arguments being SDL boolean expressions over the program variables. The specification language is similar to that used in FormalCheck^{TM} . In a co-design case, S/R code generated by `sdl2sr` is mechanically concatenated with S/R code produced by the FormalCheck^{TM} compiler for hardware modules.
2. Cospan performs model checking on this S/R code, with any valid combination of its options, including symbolic verification and localization reduction. If it detects that the program model fails to satisfy the specification, it produces an error track demonstrating one of the failure scenarios.
3. The tool `T2sdl` extracts from the error track pieces related to the SDL program and prints those with back referencing S/R names to SDL sources. In a co-design case, the remaining pieces are back referenced to the HDL.

Translation into S/R ω-Automata
In S/R, an ω-automaton that models an SDL program is described as a synchronous product of primitive ω-automata, each being represented by a distinct state variable whose transitions are defined by a single if-then-else constructor:
`asgn` $x- > a_1 \,?\, g_1 \mid a_2 \,?\, g_2 \mid \ldots \mid a_{n-1} \,?\, g_{n-1} \mid a_n$
where the omitted guard of the default alternative a_n is $true$ and the complete guard for alternative a_i, $1 < i \leq n$, is $\neg g_1 \wedge \ldots \neg g_{i-1} \wedge g_i$.

After flattening complicated data (structures, arrays, etc.), each variable of an SDL program is translated into a separate state variable.

The sequentiality of process actions is implemented by designating one state variable per process to encode the process control flow graph, say, variable C_Q

for process Q. It works like a program counter: it is assigned to labels of a process' input states and statements, assuming that all statements have been labeled. Transitions between values of C_Q mimic the control flow in the process Q. The variable C_Q is then used in transition guards of other variables owned by the process, including its local and shared variables, and buffer. Since, the buffer queue is updated by both the owner process and a sender process, a buffer transition guard may also test whether the sender process program counter points to the corresponding output action.

In S/R, non-determinism may be captured and controlled using *selection* variables [3] that are assigned to sets of values rather than to distinct values. Selection variables do not contribute to the state space. The concurrency (interleaving) of actions executed by different SDL processes is implemented by designating a special selection variable, say, S, which is non-deterministically assigned to any one of the SDL program processes:
`asgn` $S\ :=\ \{Q_1, \dots Q_k\}$.
Then, each normal transition of the program counter C_Q is guarded by the condition $S = Q$. If the condition evaluates to $false$, the program counter C_Q self-loops at its current point. For example, let the SDL process Q consist of only one statement which is a two branch decision (i.e. if-then-else) and variable x be assigned, respectively, to y_1 and y_2 in its branches. Then, S/R code for this process will have these two assignments:
`asgn` $C_Q -{>}$
 $L_{then}\,?\,(S = Q) \wedge (C_Q = L_{start}) \wedge d_{if} \mid L_{else}\,?\,(S = Q) \wedge (C_Q = L_{start}) \wedge \neg d_{if} \mid$
 $L_{stop}\,?\,(S = Q) \wedge (C_Q = L_{then} \vee C_Q = L_{else}) \wedge true \mid C_Q$
`asgn` $x -{>} y_1\,?\,(S = Q) \wedge (C_Q = L_{then}) \mid y_2\,?\,(S = Q) \wedge (C_Q = L_{else}) \mid x$
where d_{if} is the decision condition and $L_{start}, L_{then}, L_{else}, L_{stop}$ are labels of nodes in the process control flow graph. Thus, the variable S models the interleaving of the processes $Q_1, \dots Q_k$ and C_Q the control flow in the process Q. Note the regular structure of the C_Q alternatives: in each alternative guard, its rightmost conjunction factor expresses the condition under which the process control flow (whenever allowed to move by $(S = Q)$) moves from its current point, which is captured by the middle conjunction factor, to its next point, which is the alternative's value.

Optimizations

On the top of this method, SDLcheck implements partial order reduction, which optimizes model checking by selecting only one of all possible interleavings between independent actions provided that others have the same verification effect on the behavior specification. This optimization is implemented by modifying the original ω-automaton model of an SDL program to restrict its transition relation. For this, SDLcheck imposes a control over the selection variable S. Namely, if an action of process Q may be selected to execute with ignoring other possible interleavings, it is marked by the SDLcheck compiler as *ample*. In the optimized model, the selection variable S is forced to be assigned to process Q, if the current action of this process is ample. If there are several such processes, only one of them is chosen: this is a deterministic though arbitrary choice, made in advance by compiler. Only when no ample actions are enabled, S remains non-deterministically assigned by the model to any one of the program processes $\{Q_1, \dots Q_k\}$. This technique may significantly reduce the original non-determinism in the state space exploration. The objective is to have more ample actions. As explained in [4], non-ample actions appear, in particular, because of the neccesisty to break global cycles in the state space exploration by allowing the complete non-determinism at least at one point in each global cycle. To statically deal with this problem, we might mark one action as non-ample in every local loop in each process control flow graph. However, SDLcheck performs better. It statically analyzes control flow loops that belong to different processes but semantically compensate each other: for example, a loop with output of signal z is compensated by a loop (in a different process) with an input action for the same signal z. As shown in [4], to break a global cycle that executes along compensated control flow loops, it is sufficient to have a non-ample action in only one of those loops. It is how SDLcheck implements partial order reduction. As an option, SDLcheck strengthens this optimization more by forcing to execute simultaneously "by a parallel leap" (instead of sequentially) all current actions that have been marked ample.

Applications

[9] reports on verification of a robot control system developed in an UML-like graphical notation. The verification has been supported by translating the robot control system into an internal representation of SDL used by SDLcheck and then applying SDLcheck and Cospan for model checking. SDLcheck is also applied for debugging an SDL description of the H.248 gateway control protocol issued by ITU–T in 2000.

References

1. *ITU–T Recommendation Z.100 (03/93) — Specification and Description Language (SDL)* , Geneva, 1993.
2. *ITU–T Recommendation Z.100 (10/96) — Specification and Description Language (SDL)* , Addendum 1, Geneva, 1996.
3. R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, Princeton University Press, 1994.
4. R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Static Partial Order Reduction, *Proc. of 4th International Conference Tools and Algorithms for the Construction and Analysis of Systems*, *LNCS* no. 1384, pp. 345-357, 1998.
5. M. Bozga, J. C. Fernandez, L. Ghirvu, S. Graf, J. P. Krimm, L. Mounier, J. Sifakis, IF: An Intermediate Representation for SDL and its Applications. *Proc. of the SDL Forum*, Montreal, Canada, 1999.
6. G. J. Holzmann, The Model Checker Spin, *IEEE Trans. on Software Engineering* Vol. 23, No. 5, 1997.
7. D. Bosnacki, D. Damm, L. Holenderski, N. Sidorova, Model checking SDL with Spin, *Proc. of the Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, 2000.
8. Levin, V., E. Bounimova, O. Başbuğoğlu, and K. İnan, A Verifiable Software/Hardware Co-design Using SDL and COSPAN, *Proceedings of the COST 247 International Workshop on Applied Formal Methods In System Design.* Maribor, Slovenia, pp. 6–16, 1996.
9. N. Sharygina, R. P. Kurshan, J. C. Browne, A Formal Object-oriented Analysis for Software Reliability, To appear at FASE 2001.