Verifying Hardware in its Software Context

R. Kurshan V. Levin M. Minea D. Peled H. Yenigün

Bell Laboratories
Lucent Technologies
Murray Hill, NJ 07974

Abstract

We describe a method for verifying hardware whose correct behavior depends upon its software interface. It is presumed that the hardware is presented as a synchronous RTL model whereas the software is presented as an asynchronous abstraction. Our methodology incorporates partial order reduction on the software side, and localization reduction, to deal with the computational complexity of the verification. The partial order reduction is implemented as a constraint on the transition relation of a synchronous transformation of the software model. The reduced transformed model then may be verified using a verification algorithm whose scope is purely synchronous models, without modification. Thus, independent of the interface verification problem, this gives a general method for combining partial order reduction with symbolic model-checking.

1 Introduction

One facet of hardware/software co-design is dealing with verification at the hardware/software interface. In the typical case that the hardware and software are designed separately, a common problem is that the expectations of each design group do not match the implementation of the other. This can lead to logical failures and costly delays in production. The familiar presumption that the hardware and the software each can be designed successfully with only a sketchy understanding of the other side may be unwarranted. Each design team often needs a precise description of its "environment" as defined by the other design. If formal models of the hardware design and the software design are used in support of this need, the problem of interface mismatch is moved to a problem of how to develop a design in the context of a formal environment description.

Conceptually, formal verification could offer a solution to this new problem. Direct application of model-checking in this context is mostly infeasible, however, since each side individually is often near the limit of feasible verification. Moreover, there is an intrinsic difficulty in applying finite-state verification to software, which often is viewed as an infinite-state system on account of its dependence on memory.

We describe a methodology which seeks to overcome these obstacles. An environment description need not capture the entire design, but should be precise for those functions on which the other side depends for its correct operation. For the software model, we use an asynchronous abstraction which captures the level at which the software and hardware interact. In our case, this software model is written in the international standard protocol specification language SDL [12]. The SDL model can be formally refined into an implementation whose consistency with the SDL model can be verified. The hardware is expressed in a synthesizable subset of VHDL or Verilog. Both the software and hardware models are compiled together into S/R, the input language of the verification tool COSPAN [4]. The two models coordinate through interfaces each of which looks like an SDL process to the software side, and an HDL module to the hardware side.

To deal with the computational complexity of the interface verification, a partial order reduction [11] is applied to the software model. The combined hardware/software model is localized [7] relative to each respective property to be checked. In order to have the partial order reduction of the asynchronous software model be compatible with the synchronous hardware model, we transform the asynchronous model to an equivalent synchronous model [9] and implement the partial order reduction in terms of constraints defined by automata. These automata are defined at compile time. The synchronous transformation uses nondeterminism to simulate asynchrony and the constraints remove transitions which are redundant with respect to the partial order reduction. Thus, an implicit form of the entire partial order reduction is cre-

1092-3152/97 $10.00 © 1997 IEEE

ated statically by the compiler, without any need to modify the COSPAN verification tool. The reduced model is checked by COSPAN in the same manner as any synchronous model. Independent of the interface verification problem, this gives a general method for combining partial order reduction with symbolic model-checking.

1.1 Previous Work

While hardware/software co-design has been treated extensively in terms of methodology, little has been published on formal co-verification. The co-verification problem is treated mainly as a testing problem, and a number of commercial tools have appeared for testing software in the context of hardware, and vice versa.

Partial order reduction and symbolic state space search have been applied widely to cope with the intrinsic computational complexity of model-checking. Typically, these techniques are applied separately – only one or the other – since they appear to be incompatible. Partial order reduction algorithms in the literature [3, 6, 11, 13] are based intrinsically upon explicit state enumeration in the context of a depth-first search. The partial order reduction is derived dynamically in the course of enumerating the states. Symbolic verification deals only with sets of states defined by respective Boolean functions. State reachability is expressed in terms of the convergence of a monotone state set operator which implements a breadth-first search of the state space. What makes partial order reduction problematic in this context has to do with guaranteeing that certain transitions which may be deferred in the course of the state space search (thus giving rise to the desired reduction), are not deferred indefinitely (resulting in an incomplete search). The usual way this is implemented in the context of explicit-state enumeration, is to explore all such deferred transitions in the course of closing any cycle during a depth-first search.

Nonetheless, there is no intrinsic reason that partial order reduction and symbolic verification cannot be combined. Perhaps the first published proposal for such a combination is [1]. In that paper, the cycle-closing condition is replaced by some additional steps in the course of the symbolic breadth-first search, requiring the normal model-checking algorithm to be altered.

In contrast, the algorithm presented here constructs an automaton which implements the partial order reduction implicitly. The reduction is completely isolated from the model-checking algorithm and it therefore does not depend on a specific search technique.

2 Hardware/Software Co-verification

One could envision a design methodology which starts from an abstract behavioral design model. This model may not distinguish hardware from software. If it does, it need not be reflected syntactically and need not be evident architecturally either, appearing perhaps only as an annotation. Such a unified abstract behavioral model could be formally verified and then formally refined (in a verifiable manner) to separate hardware and software components. The verification of the refinement would guarantee that the properties verified in the abstract design model are inherited by the coordinated behavior of the hardware and software implementation models [7], relieving the need to further check these properties in the refinement.

An advantage of this approach is that functional verification which today is conducted at the synthesizable (RTL) level, instead could be conducted at an abstract behavioral level, where the design model is simpler. This not only aids the verification process, but also simplifies and accelerates the functional debugging itself, as there are fewer constructs that need to be altered to fix a design error. At this earlier stage of design development, the verification can be used to support architectural design evolution by spotting logical weaknesses in a preliminary architecture. Moreover, supporting debugging at an earlier stage in the design cycle is a well-known accelerant of the entire design development. In contrast, conventional design methodologies normally support testing and functional debugging only at a much later stage, once the synthesizable model is complete. Thus, the successive refinements design development methodology both accelerates the design process and produces more reliable designs.

Although the benefits of design though formal stepwise refinement have been corroborated in pilot projects and reported for some time [5], there has been no evident surge in the CAD industry to embrace this apparently promising methodology. Reasons for this may lie with the significant methodological changes it would entail and perhaps even more, with a lack of commercial technology available to support this methodology. Nonetheless, now that formal verification is a commercial reality [8], users are beginning to awaken to the benefits of abstraction and refinement. In the case of the Lucent Technologies verification tool FormalCheck^{TM}, this is the single most sought-after advanced enhancement requested by users.

There are a number of ways such a feature could be introduced and supported. One good way is as a solution to a well-known important problem such as

dealing with design complexity.

In this article, we suppose that the technology for abstraction and refinement already has been introduced into a verification tool such as FormalCheck, and we explore how this technology could be extended to solve the titled problem. At some future further in the distance, we may imagine a unified representation of hardware and software. Pragmatically, we believe such a unified representation will need to be approached incrementally, and only after proving the over-all methodology using contemporary frameworks. Thus, we have developed a system for co-verification based on existing programming environments developed for VHDL, Verilog and SDL as we will describe presently. However, the methodology described here is not strongly tied to a particular representation of hardware or software, except with regard to the granularity of time as a mediating design factor. For hardware, it is assumed that the design is (clock) cycle dependent and thus we use a synchronous model of coordination. By contrast, software is presumed to operate under synchronization conditions guaranteed by its host, and thus coordination among software design components is modelled as asynchronous, and in fact as interleaving. (There may be occasion to model asynchronous hardware. For this, interleaving semantics may not be an appropriate model, if the possibility of two simultaneous asynchronous events needs to be considered.)

At least for the present, since hardware and software are viewed at different levels of abstraction and embrace conceptually different aspects of a design, it is not unnatural to represent them with different languages tailored to their respective applications. The choice of a hardware description language within this context is strongly dictated by current use: it must be VHDL or Verilog. While these languages are closer in their use to low-level programming languages than to formal description techniques, the strong semantics which adheres to them from synthesis, as well as the perceived support for more structural constructs, make them suitable formal description languages. Our platform supports both. Unlike with hardware, there is no widely accepted formal description language for software. As the size of the software projects makes their management increasingly difficult, the value of using formal languages and formal verification in the software development has been better understood. Some companies have designed their own formal languages and tools, and other languages have been promoted by international standardization. The Specification and Description Language (SDL), a standard language of ITU-T (formerly CCITT), is one of the most promising of these languages. It is regularly updated by ITU-T and has already found some applications in software projects. Likewise, we have focused on the use of SDL for software description, in order to advance software programming to a more abstract level, as it has been achieved to a certain extent with hardware level through the use of the hardware description languages VHDL and Verilog. Once it has been verified, an SDL description of a system can serve as a formal documentation and be used in the development phase. With the support of formal verification, a refinement may be checked against an abstract SDL model, and then synthesized automatically by generated C code as is supported through COSPAN.

Our approach takes advantage of an existing compiler [2] from SDL to S/R (the native language of COSPAN), which has been implemented in order to verify software systems written in SDL using COSPAN. A modified version of this compiler, together with COSPAN, form the tool basis for our co-verification method. We assume that the hardware part of the co-design is described either in VHDL or Verilog, for which a translator to S/R is supported by the commercial tool FormalCheck, and the software part is described in SDL. The interface between hardware and software is also described in SDL, but with a slight modification in semantics, described in [10]. In essence, it allows an interface process to read and write (combinational) signals from the hardware part. The coordination of an interface process with software processes and with other interface processes is handled completely by the SDL communication mechanism, namely the signal exchange through buffers. Therefore an interface process looks like another software process from the view point of the software part. An interface process interacts with the hardware using shared variables called *interface variables* in the synchronous fashion of the hardware. Therefore, it appears like another hardware module to the hardware part of the system. The hardware, software and the interface parts of a co-design system are translated into S/R. This enables us to treat the entire co-design system having different parts with different natures, as a single synchronous system.

We distinguish between two types of global transitions in a co-design system. The transition of a single SDL process is called a *software transition*. The other kind of global transition is a *hardware transition*. Since hardware has a synchronous nature, a hardware transition corresponds to simultaneous transitions of all the hardware modules in a synchronous hardware

component. An interface process can have both kinds of transitions. A transition of an interface process in which an interface variable is referenced counts as a hardware transition. Otherwise, it counts as a software transition. Since different abstraction levels can be used for hardware and software, we do not make an assumption about the relative speeds of the software and hardware.

Conventional verification of purely hardware or purely software systems already faces great problems in dealing with complexity. In order to be able to deal with the far greater complexity when both a hardware and a software system are considered together, we use two different approaches to co-verification, assuming that the properties to be verified refer mainly either to the hardware part or to the software part.

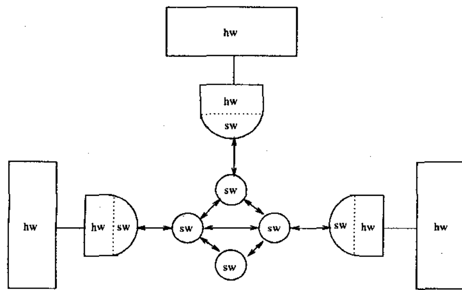

Figure 1: Software-centric view

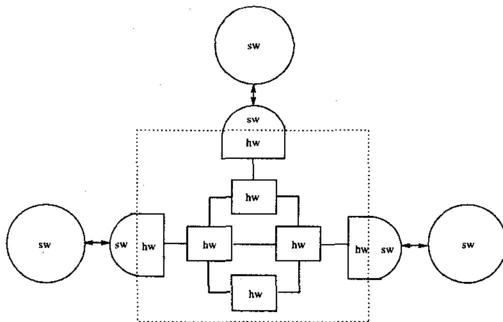

Figure 2: Hardware-centric view

2.1 Software-Centric Verification

To verify software properties of a co-design, we use a software-centric approach. While verifying the software part may not at first appear to be a fundamental part of the hardware verification, in fact it is. The reason is that it is not useful to verify the hardware in the context of a faulty software model. The software model must also be debugged. Of course the same argument applies to verifying the software model in the context of a buggy hardware model. Thus, a few software-hardware-software iterations of verification centricity may be necessary before the co-design stabilizes, and both sides verify.

In the software-centric view of a co-design system, each interface process encapsulates a set of hardware modules, which may be common for two or more interface processes. In other words, several interface processes may coordinate with one hardware subsystem which forms their shared resource.

Figure 1 illustrates the structure of the software-centric view of a system. The software system represented by the circles form a pure SDL system, with all processes coordinating asynchronously. The interface processes together with the hardware components form an environment for this SDL system. Therefore, the concept of environment, which is considered to be completely non-deterministic in [12], becomes a structured world which enforces constraints on the behavior of the SDL system. Since the aim of the software-centric approach is to verify software properties, the hardware part typically can be reduced automatically to a high level of abstraction through localization reduction (see below). The role of the hardware part is to adequately constrain the software part, in support of its verification.

In order to further cope with the complexity of the software-centric verification, we incorporate partial order reduction to reduce complexity of the SDL model. Unlike other applications of this reduction technique which implement the reduction dynamically in the course of the state reachability, we needed an algorithm which would be compatible with the synchronous nature of the companion hardware parts. Moreover, on account of the large state space contributed by the hardware, we needed to be able to apply the reachability analysis symbolically. These needs led us to search for a way to implement the partial order reduction statically (during compile time) in a fashion that would be compatible with symbolic search. We started with the idea that a partial order reduction can be viewed as a constraint on the transition relation of the unreduced system, and that such a constraint could be represented synchronously as an automaton, compatible with COSPAN's underlying automata-theoretic basis. Furthermore, COSPAN efficiently implements safety constraints in a fashion that prevents the exploration of unaccepted traces. Although we originally envisioned that we would need to augment COSPAN's core algorithms to implement the cycle-closing condition discussed above, it clearly is of great advantage that we have discovered a way

to avoid this. Instead, the cycle-closing condition is implemented into the statically defined constraint automaton as well. No change to COSPAN was necessary – the partial order reduction is encoded completely into the input program by the SDL compiler.

The SDL model is transformed to an equivalent synchronous model using nondeterminism to simulate asynchrony, as described in [9]. The statically derived automaton which implements the partial order reduction is then composed with this synchronous transformation of the SDL model, and the (unreduced) synchronous hardware model. Localization reduction [7] is then applied to this combined synchronous model, in order to reduce it relative to the property being checked. When the property is implemented mainly through control structures in the SDL model, it can be anticipated that much of the hardware will be abstracted by this automatic procedure.

2.2 Hardware-Centric Verification

Hardware-centric verification is used to check properties relating mainly to the hardware side of the co-design. Figure 2 shows an example of a a hardware-centric view of a co-design system. The software modules in Figure 2 correspond to distinct SDL systems which can coordinate only through the hardware. Each SDL system has only one interface process. Unlike the dual software-centric view, in the hardware-centric view, only hardware modules which are part of the same subsystem are considered at any one time. Thus, within the dashed box of Figure 2, all the modules coordinate synchronously. If there are other hardware components in the greater system, they must be treated in a separate hardware-centric model. The interface processes together with the pure SDL systems form an environment for the hardware part and establish constraints on the primary inputs of the hardware.

Just as with the software-centric system, partial order reduction followed by localization reduct may be applied. However, if little of the software side gets entailed in the hardware-centric verification, localization reduction may suffice, and be faster than running both algorithms in succession.

3 Partial Order Reduction

There is an extensive literature on partial order reduction techniques (see, for example, [3, 11, 13]). Here, we use the ample set method of Peled [11]. Partial order reduction in general exploits the observation that in models of coordination with the semantics of interleaved events, concurrent events are modelled by executing the events in all possible orders. The reduction exploits the fact that properties for such systems cannot distinguish among the different orders, producing a representation of the checked system which contains only a subset of the states and transitions of the original system. This representation is a quotient of the unreduced model, being equivalent to the unreduced model with respect to the property. The subset of the behaviors in the quotient, which are the paths in a *reduced state graph*, need only preserve the checked property. Namely, when the model-checking algorithm is applied to the reduced state graph, it would result in a positive answer when the property holds, and a counter example when it does not hold.

3.1 The Ample Sets Method

For each state s of the modelled system, there is a set of (atomic) transitions that can be executed next, or are *enabled*. These will be denoted by $enabled(s)$. The simple model-checking algorithm constructs the state graph of the program by performing a search, typically, a depth or breadth first search. The search starts with an initial state and progresses to explore its immediate successors, generated by applying the enabled transitions, then their successors, and so on. Different search strategies vary from each other by the order in which the successors are searched.

From a given state s, the partial order reduction explores only a subset of the enabled transitions: the set $ample(s)$. Such an *ample set* is chosen to enforce some conditions that guarantee the preservation of the checked property in the resulting quotient.

We assume that every state is labeled by some subset of the atomic propositions that appear in the formulation of the checked property. We denote by $L(s)$ the set of propositions that hold in s. Before presenting the conditions we use for choosing $ample(s)$, we define two more concepts. A pair of transitions is said to be *independent*, if executing one and then the other starting at any state s results in the same subsequent state s', regardless of the order of execution. We require moreover that if one of them is enabled, then after executing the other, the first remains enabled. The second concept is that of *invisible transitions*. A transition is invisible if when executed, it does not change the state labeling. That is, for each pair of states s and s', if executing the transition from s results in s', then $L(s) = L(s')$.

We now give sufficient conditions on the sets $ample(s)$ for the resulting partial order reduction to preserve the given property. The explanation of these conditions and a proof that they guarantee a reduced state graph that preserves the given property appears in [11]. A state s is said to be *fully expanded* when $ample(s) = enabled(s)$, *i.e.*, all the transitions are se-

lected.

- **C0** [Non-emptiness condition] *ample*(s) is empty if and only if *enabled*(s) is empty.

- **C1** [Ample decomposition] For every behavior of the system, starting from the state s, a transition that is dependent on some transition in *ample*(s) never appears (in the unreduced model) before a transition from *ample*(s).

- **C2** [Non-visibility condition] If s is not fully expanded then none of the transitions in it is visible.

- **C3** [Cycle closing condition] At least one state along each cycle of the reduced state graph is fully expanded.[1]

The first model-checking system to implement these conditions was SPIN [6]. In order to guarantee these conditions, the search mechanism of SPIN was changed and specialized for the reduction. Conditions **C0** and **C2** are trivial to check. **C0** is a local syntactic check and **C2** is guaranteed through scoping rules imposed syntactically on the design. Condition **C1** was translated into a set of checks on the transitions enabled at the current state s. For example, if some process can execute from its current program counter only local transitions that do not involve global variables or exchanging messages, then its set of transitions are guaranteed to satisfy Condition **C1**. Condition **C3** was handled by checking whether a selected ample set closed a cycle (in depth-first search this happens when a state that is already on the search stack is reached). If so, the currently selected ample set was discarded in favor of an alternative selection.

In translating SDL to COSPAN, we wanted to use a partial order reduction. As already discussed, several goals dictated a solution different than the one used in [6]. Most particularly, in order to be able to use a symbolic state space search, we wanted a reduction algorithm which was not dependent on the use of depth-first search. The reduction described next has the added benefit that it can be used with any model-checking algorithm for synchronous models, without any modification of the model-checking algorithm whatsoever. Moreover, this new partial order reduction algorithm is very general, being insensitive to the mode of state space search employed.

[1] There are other stronger and weaker conditions that can be used instead of Condition **C3**. This particular version fits well with our framework.

3.2 A Generic Partial Order Reduction

The main obstacle to implementing the reduction in the context of a breadth-first search is that we need to avoid the possibility of a cycle which does not contain any state which is fully expanded. At first sight, Condition **C3** suggests a substantial change to the model-checking algorithm would be required to achieve this in the context of a breadth-first search (*cf.* [1]). We show how this may be accomplished in fact very simply and generally, without modification to the model-checking algorithm. In principle, it should be capable of producing at least as good a reduction (measured in terms of reached states) as [6] or [1], although with symbolic analysis, the number of reached states is a poor complexity measure. Moreover, there are trade-offs between the size of the reduction and the cost of producing the reduction, which are hard to compare across algorithms, in general.

For our purposes we strengthen Condition **C3** as follows:

- **C3'** There is a set of transitions T such that each cycle in the reduced state space includes at least one transition from T. If *ample*(s) includes a transition from T, then s is fully expanded.

We call the set of transitions T *sticky transitions*, since intuitively, they stick to all other enabled transitions.

An easy way to find such a set T to look at the static control flow graph of each process of the checked system. Any cycle in the global state space projects to a cycle (or possibly a self-loop) in each component process. By breaking each local cycle, we are guaranteed to break each global cycle. This suggests the following further strengthening, which guarantees the above Condition **C3'**:

- **C3"** Each cycle in the static control flow of a process of the modelled system contains at least one sticky transition, and if *ample*(s) includes a sticky transition, then s is fully expanded.

Condition **C3"** can be satisfied by using a static analysis algorithm on the (small) component processes, to find a set of sticky transitions T, lifted to the global state space from each respective component process. While finding a minimal sticky set is NP hard, (it is the same as removing transitions – the sticky ones – until there are no cycles left in the static control flow graph), it is NP hard in the size of the local control flow graph (*i.e.*, each process) which is small, not the much bigger global state graph. Moreover, one need not find a minimal T. One way to find a set of sticky transitions in a local control graph is to

choose all the back-edges in the course of a depth-first search, *i.e.*, an edge from the currently searched state to a state currently in the search stack. The resulting T is the lifting of each local back-edge. Since the local control graphs are small, they can be searched at compile time, at no perceptible cost. (Any syntactic transition is assumed to be an edge in the local control graph – while this heuristic could be foiled by a process with many unsatisfiable transitions, this is not commonly the case, and we envision a process as not only small, but as having a sparse set of transitions).

Note that priority is given to transitions that are not sticky. Sticky transitions may be deferred, although no cycle can be closed without having passed through at least one sticky transition. It is possible that a global cycle may include more than one fully expanded state, due to sticky transitions lifted from different processes. However, since sticky transitions have low priority, and the reduction algorithm tries to select the other transitions first, it may be worthwhile to defer several of them to a point where they all can be taken from the same state. This is possible because the expansion for sticky transitions may occur anywhere along a cycle.

3.3 The COSPAN Implementation

In practice, one may weaken **C3"**, as follows. A cycle in the reduced state graph cannot consist of only receiving messages, since all message queues would then become eventually empty, and some sends would be required to close the cycle. Thus, local control flow cycles with only message receives do not have to include a sticky transition; it must be part of another cycle projected from a cycle in the global state graph. (While that this is true also for sending messages, – the queues will eventually get filled – if the same relaxation is allowed at the same time to both sending and receiving messages, they can cancel the effect of one another and be part of a global state graph cycle.) This observation can be used to design an algorithm that is linear in the size of the control graphs of each process. After removing the receiving transitions from the control graph of a process, a linear time search can be performed on the remaining transitions to identify any back-edges and mark them as the sticky transitions.

There are many ways to further relax **C3"**. One is to involve the user to some extent in further marking sticky transitions. This would reduce the more arbitrary assignment which would result from choosing all back-edges as above. Another way is to perform the back-edge analysis after combining a few processes – only the cycles of the larger structure need be broken.

In order to achieve in COSPAN a partial order reduction that is independent of the search control, we exploit the selection mechanism of S/R. This mechanism consists of using *selection variables*. These variables are combinational, not part of the state, and thus do not incur any memory overhead. When deciding on the successor state, each process chooses non-deterministically among some possible values of its selection variables. The choice of any process can be dependent on the choice of the selections of the other processes (as long as this relationship is acyclic). In particular, we heuristically optimize the size of $ample(s)$ by selecting the process with $|ample(s)|$ as samll as possible, *i.e.*, with the fewest number of enabled transitions.

At each local state of a process, we calculate whether the currently enabled transitions satisfy Condition **C1**. If so, we say that the process is *ample* at this state. In the current version of our compiler, a process is considered to be ample at a state if it does not have an enabled sticky transition and all its enabled transitions either are receiving or internal transitions. If the process has only internal transitions (the transitions in which only the local variables are referred), then it is clear that the enabled transitions of the process satisfy **C1** since no other process can refer to those variables. Similarly, when the process has only receiving transitions, the enabled transitions of the process again satisfies **C1**. Although the send transition of another process can change the same message queue from which the receiving transition reads, their execution orders do not matter. The problem is that several processes may have enabled transitions that are potentially ample sets. Selecting all of these sets would be counter to our goal to minimize the selected transitions. We can use the selection mechanism to resolve this. When a process is ample, it can use global selection variables to inform the other processes about this fact. If several processes are ample at the same time, they choose one of them through a static (index number) priority. The other ample processes are informed via the selection variables that they are not the chosen candidate for providing an ample set, and do not contribute transitions to this set.

4 Summary and Conclusions

We have described a methodology for hardware/software co-verification which confronts the computational complexity issues directly. Our current implementation supports VHDL or Verilog for hardware and SDL for software. However, the methodology is general and could be applied to any finite-state

modelling paradigm in which the hardware is represented synchronously and the software is represented in terms of interleaving semantics. The general approach to complexity reduction lies with a combination of partial order reduction and localization reduction. The partial order reduction is introduced as a constraint on a transformation of the software model, and as such supports symbolic model-checking without need to alter the model-checking algorithm. In particular, this gives a method for combining partial order reduction with symbolic search.

By introducing the methodology in terms of currently accepted paradigms, it may be easier to introduce this technology into the commercial CAD arena. If this turns out to show signs of success, one may wish to consider supporting a larger subset of SDL, other languages in place of SDL, and eventually, a unified formalism which embraces both hardware and software.

References

[1] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, S.K. Rajamani, Partial order reduction in symbolic state space exploration, *Conference on Computer Aided Verification (CAV 97)*, LNCS **1254**, Springer-Verlag, (1997) pp 340 - 351.

[2] O. Başbuğoğlu, K. İnan, Compiling SDL into the finite state specification language COSPAN. *Proceedings of the Tenth International Symposium on Computer and Information Sciences(ISCIS X)*, Kuşadası, Turkey, 1995.

[3] P. Godefroid, P. Wolper, A Partial Approach to Model Checking, *6th Annual IEEE Symposium on Logic in Computer Science*, 1991, Amsterdam, pp 406–415.

[4] R. H. Hardin, Z. Har'El, R. P. Kurshan, COSPAN, *Conference on Computer Aided Verification (CAV 96)*, LNCS **1102**, Springer-Verlag, (1996) pp 423 - 427.

[5] Z. Har'El, R. P. Kurshan, Software for Analytical Development of Communication Protocols, AT&T Tech. J. **69** (1) (1990), pp 45 - 59.

[6] G.J. Holzmann, D. Peled, An Improvement in Formal Verification, *7th International Conference on Formal Description Techniques*, Berne, Switzerland, 1994, pp 177–194.

[7] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, Princeton Univ. Press, 1994.

[8] R. P. Kurshan, Formal Verification in a Commercial Setting, *Proc. 34th Design Automation Conf. (DAC'97)*, ACM, 1997, pp 258 - 262.

[9] R. P. Kurshan, M. Merritt, A. Orda, S. Sachs, Modelling Asynchrony with a Synchronous Model, *Conference on Computer Aided Verification (CAV 95)*, LNCS **939**, Springer-Verlag, (1995) pp 339 - 352.

[10] V. Levin, E. Bounimova, O. Başbuğoğlu and K. İnan, A Verifiable Software/Hardware Codesign Using SDL and COSPAN, *Proceedings of the COST 247 International Workshop on Applied Formal Methods In System Design*, Maribor, Slovenia, 1996, pp 6–16.

[11] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design* **8** (1996), pp 39–64.

[12] Functional Specification and Description Language (SDL), CCITT Blue Book, Recommendation Z.100, Geneva, 1992.

[13] A. Valmari, Stubborn sets for reduced state space generation, *10th International Conference on Application and Theory of Petri Nets*, Bonn, Germany, LNCS **483**, Springer-Verlag, (1989) pp 491–515.