An Overview of Some Automated Tools for Formal Analysis and Verification*

Tomáš Vojnar†
vojnar@dcse.fee.vutbr.cz

Abstract: *The article presents several automated tools for performing formal analysis and verification. The choice of the tools is restricted to the ones working on concurrent untimed discrete-event systems and exploiting state space searching.*

1 Introduction

For many years computer-aided simulation has been an important technique widely used in science as well as industry. What is more, this is still true and it will obviously remain so even in the future. However, there has appeared also another area of interest — the area of formal analysis and verification. Although capabilities of current formal methods are still restricted when compared to simulation, they have already attracted significant attention of not only researchers but also of many leading industrial companies (at least in the area of computer engineering). This indicates that there is a big potential behind these methods and that they seem to be promising for the industry. To support this idea we can present a partial list of sponsors of the Computer-Aided Verification 1998 conference in Vancouver: Fujitsu, Hewlett-Packard, IBM, Intel, Lucent Technologies, Motorola, NEC, Siemens, Sun, and others.

With respect to the previous paragraph, the aim of this article is to provide a brief description of several automated verification and analysis tools. We focus on tools working with concurrent discrete-event untimed systems and using state space methods. Tools of this category are probably the most common and most elaborated. However, there are also many other categories of tools, such as theorem provers, real-time verification tools, or tools for performance analysis, but those are beyond the scope of this tiny article. We are even unable to present all tools from the chosen class and so we select only some of them.

2 The Analysis/Verification Problem

This section presents some basic principles which the tools described in the following section are built on. The section is mostly based on the excellent tutorial on the state space explosion problem presented by prof. Antti Valmari within the 19th international conference on Petri nets ICATPN'98 in Lisbon on June 1998.

Firstly we should clarify what we mean by the terms formal analysis and verification. *Verification* is an algorithmic check that a given formally stated property holds for a given formally described system. *Analysis*, on the other hand, denotes the process of answering more general questions which are usually not of the "yes—no" form or both "yes" and "no" are of the same relevance. Answers obtained from formal analysis must usually be further processed in order to obtain answers to the questions modellers

*This is a slightly modified version of the article published in the Proceedings of ASIS'98, Krnov, Czech Republic in 1998.
†Department of Computer Science and Engineering, Technical University of Brno, Božetěchova 2, CZ–612 66 Brno, Czech Republic.

are really interested in. So it is possible to say that most thinking is done *before* running a formal verification tool but *after* using a formal analysis tool.

A formal analysis/verification method requires the following items: (1) a formalism for modelling systems (a *modelling language*), (2) a formalism for expressing questions to be answered or properties to be proved (a *property language*), (3) a formal specification of what it means that a system has a property, and (4) an algorithm for answering analysis/verification questions.

There are many different modelling languages, such as various specialised parallel languages, process algebras, Petri nets, parallel finite automata, etc. It is also frequent that modellers use a different notation (such as SDL) for creating models than the appropriate automated tools use for the actual verification. This allows modellers to use what they are familiar with.

As for property languages, it is possible to divide them into *query* formalisms for expressing analysis questions and *specification* formalisms for describing properties to be proved or disproved by formal verification. A query formalism can have the form of a state space query language or simply a set of items present in menus of a given analysis tool. The role of a specification formalism can be played by different kinds of temporal logics, process algebras (in their case, models and properties are typically expressed in the same way), and many other more specialised notations. Here, it is essential to distinguish between *safety* and *liveness* properties. Safety properties can be proved wrong on finite behaviours, whereas we need to examine infinite behaviours to disprove liveness. When proving liveness we often have to specify the so-called fairness assumptions which typically state something like that it is impossible to for ever omit performing some firable actions. We further distinguish between linear-time and branching time-properties (expressed e.g. in LTL or CTL, respectively).

Formal analysis/verification methods most commonly work on state spaces which can be understood as (labelled) oriented graph structures consisting of vertices (states) and edges (transitions). State spaces are also often represented as the so-called (labelled) transition systems. They provide complete information on the interleaving semantics of modelled systems which is usually enough. Sometimes true concurrency semantics is combined with the interleaving one in order to reduce the state space explosion which we will mention later on.

The reason why state spaces are so popular is that they are quite general, they allow for totally automatic analysis, and are easy to be used. This is usually not true for other methods, such as Petri net invariants, Petri net structural theory, or theorem proving. However, this does not at all mean that these methods are useless. They can be used in some specialised cases (such as Petri net structural theory for analysing soundness of workflow nets), as parts of other analysing techniques (such as invariants in tools transforming Petri nets to Markov chains), as a support for human driven proofs (as e.g. theorem provers like HOL or PVS), etc.

The main disadvantage of state space methods is the state space explosion problem. It is the exponential growth of the state space size when compared to the growth of model sizes. This problem stems from the nature of interleaving. Imagine having n totally independent processes each with k different states. In this case, the size of the total state space would be k^n...

Although it seems that the state space explosion problem will never be totally cured, there have appeared many more or less powerful techniques for alleviating it. We can divide them into three basic categories: (1) *Packed state spaces* which include binary decision diagrams (BDDs), symmetries, Petri net coverability graphs, various hash compression techniques, and others. (2) *Methods based on abstractions* as for example data abstraction, structural reductions, process algebraic abstractions, etc. (3) *Methods based on commutativity* (also *partial order methods*) which contain stubborn sets (also persistent or ample sets), sleep sets, and others.

It is far beyond the scope of this article to go into details of the particular techniques, but we will mention some of them in the context of the tools described in the next section.

3 Some of the Automated Analysis/Verification Tools

3.1 Spin

Spin [3] is a tool determined primarily for analysis and verification of concurrent systems, especially communication protocols. The authors of this tool emphasise that is intended to be used in the area of software rather than hardware development.

The input language of Spin is called Promela. It allows modellers to describe systems in terms of process templates from which concurrent interleaving processes are then instantiated dynamically at run time. The processes can communicate asynchronously via shared memory and finite capacity channels or synchronously via zero capacity channels. Promela allows one to use common integer types, boolean variables, static arrays, and finite capacity channels.

During simulation and verification Spin looks for deadlocks, unspecified receptions, and unexecutable code. Furthermore, it can check the absence of livelocks and verify the correctness of system invariants (expressed as assertions in the model). Finally, Spin can also verify invalidity of undesired properties expressed by negative formulae of the next-time free version of LTL. As for fairness assumptions, Spin can automatically take into account weak fairness and other assumptions can be made a part of formulae to be verified.

The Spin verifier is based on translating processes into parallel augmented automata and building their asynchronous product representing the state space. A negative LTL formula whose invalidity is to be proved is translated into a Büchi automaton and Spin builds on-thy-fly its synchronous product with the state space and checks for the emptiness of the resulting language, i.e. for the absence of accepting cycles.

The attitude of Spin to alleviating the state space explosion problem is using a *partial order* method and some *state compression* methods. The Spin's partial order method is a member of a broader family of similar methods which are based on the observation that if several enabled actions are independent in some way it is not always necessary to build all of their interleavings. These methods usually save both time and space. As for the state compression methods, Spin uses the recursive indexing method and the two phase compression based on Huffman's encoding. These methods yield big memory savings but we have to pay a lot of time for that.

The last method which can be applied in Spin is the *supertrace* or *bit state* technique which uses states as keys for hashing into a bit array. In the bit array there is stored only information about whether a state with a given value of the hash function has already been found or not. This method does not lead to classical verification because it can merge different states with the same value of the hash function. However, it can say at least something about so big systems that other attitudes could not manage at all under given memory restrictions.

3.2 Murφ

The tool Murφ [4] was primarily designed for verifying communication protocols and hardware controllers and protocols (such as cache coherence protocols).

The input language of Murφ is based on creating sets of guarded commands (condition/action rules) which are then executed repeatedly in a loop. As for data types, subranges, enumerations, arrays, records, multisets, and some special types allowing state space reductions are offered.

Murφ allows one to verify absence of deadlocks, error statements, and validity of invariants and assertions specified within system descriptions. Liveness properties are not supported because of problems with combining them with using symmetries for reducing state spaces[1].

[1]An older version of Murφ (without symmetries) used to offer a subset of LTL for specifying properties including liveness.

Murφ works in such a way that it generates reduced state spaces according to the appropriate rules described in the input language and on-the fly checks the validity of given invariants. The main reduction method employed in Murφ is based on *symmetries*, i.e. finding automorphisms on state spaces. It would be too time consuming to look for general symmetries and therefore Murφ looks only for permutations of values of the special data type called "scalarset". Values of this data type are typically used for describing multiple similar system components and thus the identity of these components can be neglected. An extension of this method is the method based on *replicating components* which tries not to store exact numbers of replicated components which are currently in the same local state. Instead, it stores only whether there are 0, 1, or ≥ 1 of the appropriate components in the appropriate local state. Finally, these methods can be combined with using *reversible rules*. In this case, we store only a single state instead of a set of states provided that all these states can be obtained from their representative by correct reversible rules. All these methods in combination can significantly reduce memory as well as time requirements of verification.

Apart from the above described methods, Murφ can also use hash compression which stores only small signatures of particular states. This leads to a loss of information and to the so-called probabilistic verification, i.e. there are some probabilistic bounds on missing errors in models.

3.3 SMV

The SMV system [5] can be especially useful for verifying synchronous as well asynchronous hardware.

The SMV's input language is based on describing changes of state variables by means of parallel conditional assignments. All these assignments are then repeatedly performed in a synchronous way. System descriptions can be broken down to modules. Instead of modules, it is also possible to use processes. The execution of models is then asynchronous on the level of processes, but synchronous within particular processes. The offered data types are: boolean, enumerations, static arrays, and structures.

Properties to be verified by SMV are specified in CTL. Fairness assumptions can be specified by defining processes which must keep on doing some steps and then of course by means of suitable forms of CTL formulae to be proved.

SMV uses an on-the-fly symbolic method based on *OBDDs*. OBDDs (Ordered Binary Decision Diagrams) are directed acyclic graphs whose every node has zero or two successors. Internal nodes correspond to binary system variables and their outgoing edges to the two possible values of these variables — 0 or 1. Leaves are labelled by `true` or `false` according to the fact that the appropriate state vector has been or has not been included into the state space. This representation allows for reducing state space sizes by replacing subtrees of OBDDs by single `true` leaves as soon as all their original leaves become `true`. Symbolic model checking on OBDDs is then based on interpreting both states and transitions as formulae, representing both of them by OBDDs, and using special fixed point algorithms over OBDDs. This method has proved especially useful in the area of hardware. However, it seems not to work that well in the area of software. Its main disadvantage is that it heavily depends on the chosen order of state variables which is typically established by using some heuristic algorithms.

3.4 PEP and PROD — Two Petri Net-Based Tools

PEP [1] is an open platform which uses parallel languages (B(PN)^2 and restricted SDL), interacting finite automata, process algebras, and high-level as well as low-level Petri nets as its input languages. Models described in these languages are automatically compiled into 1-safe low-level Petri nets. PEP can then verify for example reachability properties, deadlock-freeness, or properties expressed in a simple branching-time logic exploiting finite prefixes of branching processes of the low-level nets (or some other methods). A branching process represents an unfolding of a net which is again a net but without cycles. Apart from that, PEP allows models to be translated into input languages of many other tools, such as

Spin, INA (a Petri net reachability analyser), SMV, etc. A disadvantage of PEP is that all models it can handle must be transformable into low-level Petri nets which implies that all system variables must have *in advance* restricted ranges.

PROD [7] uses Pr/T nets as its input language. From among data types, PROD allows one to use only tuples of integers. PROD implements reachability graph generation which can be combined with the following reduction methods: symmetries (low-level net symmetries computed via unfolding), the stubborn set and the sleep set method (both of which are partial order methods). PROD also provides on-the-fly verification of testers and LTL formulae using stubborn sets and off-line CTL verification on in advance generated reachability graphs. Further, there exist the Emma system which allows (almost unrestricted) TN-SDL programs to be translated into PROD nets and verified. However, a big disadvantage of PROD is that it also performs unfolding to low-level nets to be able to use its reduction algorithms. This disadvantage should be overcome within the Maria system whose development started recently and which should work directly on high-level nets.

3.5 Two Process Algebraic Tools: Ara and NCSU-CWB

Process algebras provide different calculi of processes. They usually contain two typical operators, namely parallel composition and hiding. Process algebraic analysis is based on different kinds of equivalences which allow us to compare systems of processes taking into account only some of their aspects. Thus we can describe both the model of a system and a specification of its desired properties as systems of processes and compare them using a suitable equivalence relation. Process algebraic equivalences have yet another interesting feature — they allow us to replace processes by simpler, but from some point of view equivalent, ones. What is more, process algebraic reductions can be performed in a compositional way and they also allow for on-the-fly model checking. General drawbacks of process algebraic techniques are connected to problems with expressing fairness and working with bigger amount of data inside models.

Ara [6] uses a restricted variant of Lotos as its input language. It uses the abstract CFFD (Chaos-Free Failure Divergence) equivalence and implements a compositional state space construction method with on-the-fly checking for the absence of situations which have been marked as dangerous in an input system description. Ara further offers tools for normalising labeled transition systems (LTS) representing state spaces of systems of processes and comparing them under the CFFD equivalence. Finally, Ara offers a visualisation tool which produces pictures of LTSs resulting from composition, hiding, and applying the CFFD equivalence.

The NCSU Concurrency Workbench (NCSU-CWB) [2] uses several different input languages, such as CCS or CSP. It can check many equivalences and preorders (such as bisimulation or trace equivalence) and it can also verify whether a given system satisfies a given μ-calculus formula. New input languages and equivalences can easily be added into the tool.

4 Conclusions

We have presented several automated tools for performing formal analysis and verification of concurrent untimed discrete-event systems. Of course, it was impossible to describe all tools from this class. We have not mentioned for example Cospan (parallel ω-automata and BDDs), VeriSoft (C as the input language and state-less partial order validation), and many many others. What is more, we have not covered other classes of tools at all. At least now we should mention that there also exist many *real-time verification* tools (Uppaal, Kronos, HyTech, ...), *performance analysis* tools (e.g. GreatSPN), hardware *synthesis* tools (Petrify), tools based on different principles than on state spaces (theorem provers and others), etc.

It is clearly visible that the number of formal analysis/verification tools is tremendous. So, how to choose the suitable one? We can for example consider answering the following questions: What sort of systems are we dealing with? (In the following, we suppose the concurrent untimed discrete-event ones only.) Do we need synchronous or asynchronous semantics? What kind of input language do we prefer? What data types do we need? And what sort of inter-process communication? What properties do we need to verify and with what sort of fairness? Is the nature of the systems to be modelled consistent with the needs of the reduction methods in the selected tools?

In the article, we have tried to answer the above questions for several well-known tools. However, interested readers are encouraged to perform their own experiments — all mentioned tools are freely accessible via Internet despite the effort invested in them. And, after all, it can also turn out to be advantageous to use a combination of several different tools for verifying different aspects of systems under investigation.

Acknowledgment. This work was supported by the grant of the Czech Grant Agency under the contract 102/98/0552 "Research and Applications of Heterogeneous Models".

References

[1] E. Best. Partial Order Verification with PEP. In *Proceedings of the POMIV'96*, 1996.

[2] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. Henzinger, editors, *Proceedings of the International Conference on Computer-Aided Verification, CAV'96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, 1996. Springer-Verlag.

[3] G.J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[4] C. Norris Ip and D.L. Dill. Efficient Verification of Symmetric Concurrent Systems. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Science*, 1993.

[5] K.L. McMillan. The SMV System. Technical report, Carnegie-Mellon University, 1992.

[6] A. Valmari and R. Savola. Verification of the Behaviour of Reactive Software with CFFD-semantics and ARA Tools. In *Proceedings of the International Symposium on On-Board Real-Time Software, ESTEC*, pages 173–180, Noordwijk, The Netherlands, November 1996.

[7] K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. PROD Reference Manual. Technical Report B13, Helsinki University of Technology, August 1995.