THE EFFICIENT SYMBOLIC TOOLS PACKAGE

Robert Meolic, Tatjana Kapus, Zmago Brezočnik
Faculty of Electrical Engineering and Computer Science
University of Maribor
Smetanova ul. 17, SI-2000 Maribor, Slovenia
E-mail: {meolic,kapus,brezocnik}@uni-mb.si

Abstract: *Efficient Symbolic Tools* (EST) *is a software package for formal verification of concurrent systems. It appears as an educational project and has been entirely written in the Laboratory of Microcomputer Systems at the Faculty of Electrical Engineering and Computer Science in Maribor. The main purpose of our work was a study of algorithms that could serve for formal verification of complex protocols, which are used in computer and telecommunication networks.*

KEYWORDS: concurrent systems, **process algebra**, **symbolic verification**

1 INTRODUCTION

Verifying concurrent systems (e.g. communication protocols) is known to be a difficult challenge for verification technologies. Concurrent system is a system composed of two or more components which can be concurrently executed and communicate with each other. There are many research groups across the world providing their own tools, commercial or non-commercial, which perform formal verification of such systems. Inspired by the reported work, we started in 1992 to build our own tool that we now refer to as *Efficient Symbolic Tools package* (EST) [12, 8, 9, 11]. EST distinguishes itself as a small and efficient package with an easily readable source code and well implemented algorithms. It runs on many different computers with different operating systems, including HP-UX, Linux, and Windows 95/98/NT.

A verification problem consists of formally establishing a relationship between a design specification and a requirements specification of a system. The design specification describes how the system is implemented and the requirement specification describes how the system should behave. The system being verified with EST is described by a set of communicating processes. Processes are represented using a formalism derived from CSS and CSP. The syntax used for describing processes is similar to the syntax of the CCS/MEIJE process algebra [1].

EST has a simple user interface. The user enters commands either textually or choses them from menus. A description of processes is read from textual files. Processes are then composed together to represent the behaviour of the whole

system. Formal verification can be done by equivalence checking or by model checking.

EST uses symbolic methods to represent and manipulate processes. States and transitions are encoded by Boolean functions rather that being explicitly enumerated. Further, Boolean functions are represented with binary decision diagrams (BDDs). Thus, operations on processes are performed as operations with Boolean functions, which are actually performed as manipulations with BDDs (Figure 1).

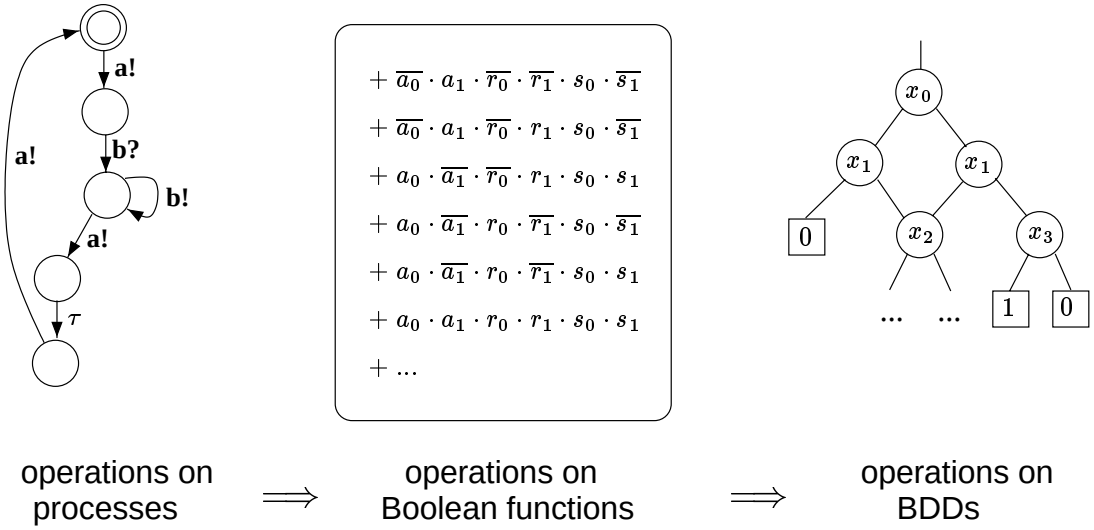

Figure 1: A schematic view of symbolic verification

Further, the paper is organised as follows. The next section gives an overview of EST. The main characteristics of the package are given. An exact description of used algorithms and mathematical background are not given in this paper. An interested reader may find them in [2, 3, 4, 5, 10]. Section 3 describes how to interact with EST during the verification process. Also, this section gives an insight into the capabilities of the tool. Section 4 presents an example of verification of a simple concurrent system. In the conclusion we give some directions for future work.

2 AN OVERVIEW OF EST

EST is a modularized package and thus easy to extend. Currently, there are four main modules: the Binary Decision Diagrams module, Process Algebra module, Versis module, and Model Checking module. All modules, except parsers, are written in C and compiled with `gcc`, while the user interface is realized with Tcl/Tk scripts. This enables us to run EST on many different platforms. The flexibility of EST is supported by efficient memory management. Many interesting systems can be verified with as few as 8 MB of available RAM.

The **Binary Decision Diagrams** module is a general purpose BDD package for the manipulation of Boolean functions. Boolean functions are represented as *reduced ordered* BDDs with complemented edges. The BDD package includes a simple parser, cache tables for basic operations, and automatic garbage collection. All standard operations on Boolean operations are implemented: conjunction, disjunction, negation, ITE-operation, existential and universal quantification of variables, restriction of Boolean functions, and composition of Boolean functions. A lot of statistics are available about the size of represented Boolean functions and the efficiency of BDD representation.

The **Process Algebra** module is a framework for representing processes. Each process executes input and output actions and transits between states. The concurrent execution of two or more processes is asynchronous. An action is an input action if its name terminates by '?', e.g. $a?, b?, ...$, and it is an output action if its name terminates by '!', e.g. $a!, b!, ...$ Two actions whose names differ only in the last character, e.g. $a?$ and $a!$, are called complementary actions. Two processes synchronise with each other by simultaneously performing complementary actions. Processes may include nondeterminism and there is also the special action τ, which is used to model internal communications, not visible to an external observer.

The most important operations performed by the Process Algebra module are *parsing* of an input file and *encoding* of processes. The parser is written using `Flex/Bison` and it allows comments in the input file. An example of input file describing the process from Figure 1 is in Figure 2.

```
/* sort is a list of action names without '!' and '?' */
SORT sortExample a,b

/* process is described by its transitions */
PROCESS P
SORT sortExample
ACTIONS a!,b!,b?
INITIAL STATE s0
TRANSITIONS s0 = a!.s1
            s1 = b?.s2
            s2 = b!.s2 + a!.s3
            s3 = TAU.s4
            s4 = a!.s0
```

Figure 2: Specification of a process

The **Versis** module implements operations on processes. The most important operations are composition of processes and different kinds of equivalence checking. Currently, the Versis module is capable of efficient *parallel composition* of two or more processes, checking *strong* and *weak observational equivalence* and

checking *testing equivalence*. Parallel composition is used to compose processes together. A parallel composition of processes represents a common behaviour of these processes when executed concurrently, where the internal communication between processes is hidden. Equivalence checking is used for studying the relationship between processes' behaviour. Different systems or different levels of the system abstraction can be compared. Checking of observational equivalences and checking of testing equivalence are two verification approaches that supplement each other. The observational equivalences take into account the branching structure of processes, while the testing equivalence treats processes with regard to the external observers.

The **Model Checking** module provides a parser for action computation tree logic (ACTL) and functions for ACTL model checking. ACTL is used for describing system properties. It is a propositional branching-time temporal logic, which is very suitable for specifying properties of a system described with a process algebra. It is similar to the popular CTL and it has all the nice characteristics of it, including the feasibility of efficient symbolic model checking.

The syntax of ACTL formulas consists of constants *true* and *false*, *action variables*, standard Boolean operators, *path quantifiers* **E** ("there exists a path") and **A** ("for all paths"), and *temporal operators* **U** ("until"), **U̲** or **UU** ("unless"), **X** ("for the next transition"), **F** ("for some transition in the future"), and **G** ("for all transitions in the future"). With ACTL formulas one can easily express properties like: "action $a!$ will be executed in the future" (**AF** $\{a!\}$), "it is always possible that action $a!$ will be executed" (**AG EF** $\{a!\}$), "action $a!$ will be executed infinitely often" (**AG AF** $\{a!\}$), "after action $a!$ has been executed, action $b!$ will never be executed" (**AG** $[a!]$ **NOT EF** $\{b!\}$), and many others.

3 INTERACTING WITH EST

`EST` is started by running the user interface called **My Interface**, which is implemented as a set of `Tcl/Tk` scripts and located in subdirectory `mi`. The main script is called `mish.tcl` (`mi` shell). After the user interface is initialized, `mish` executes commands from file `miStartUp.tcl`. Usually, this initializes all other modules. At this point, `EST` is ready for work.

My Interface consists of three parts: menu, output window and input window (Figure 3). Commands are entered in the input window and responses from the tool appear in the output window. Alternatively, commands can also be chosen from the menu or read from a user's `Tcl` script. In the last case, the user has two different ways to execute the script. With the command `source` the commands from the script are executed as if they were sequentially entered in the input window. If the command `xsource` is used, after the execution of each command the information about the time used for the completion of the command is reported.

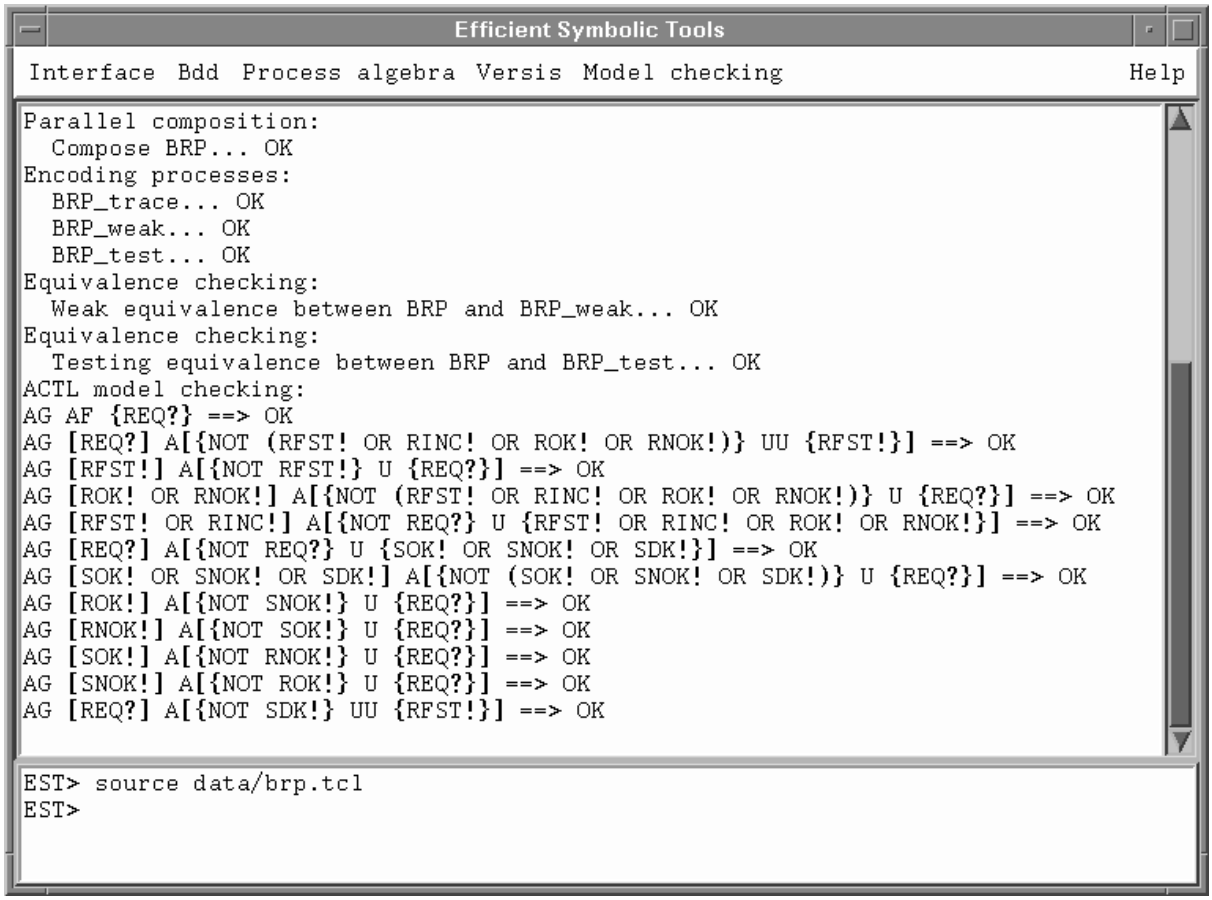

Figure 3: My Interface — a simple user interface of EST

Descriptions of processes are loaded with the `pa_read_process` command. The descriptions are not automatically encoded with Boolean functions. This has to be done with the `pa_encode_sort` and `pa_encode_process` commands. An encoded process can be studied in many ways. Its set of states and transition relation are encoded with Boolean function `S_process_name` and `D_process_name`, respectively. Boolean functions in CNF can be obtained with the `bdd_out_function` command. Moreover, a BDD representing a Boolean function can be written with the `bdd_out_BDD` command. Note that the transition relation of a process can be encoded with a Boolean function which contains thousands of minterms and therefore a request for writing the Boolean function or BDD has sense only for small processes. BDD nodes required for the representation of Boolean function can be counted with the `bdd_node_number` command. An encoded process can be decoded with the `pa_decode_process` command.

Two or more processes are composed together with the `versis_compose` command. The set of states and the transition relation of the resulting process are encoded with Boolean function `S_composition_name` and `D_composition_name`, respectively. The composed process can be decoded with the `pa_decode_composition` command.

Currently, the user can perform formal verification by the following commands:

- `versis_strong_equivalence`,
- `versis_weak_equivalence`,
- `versis_test_equivalence`, and
- `mc_check_ACTL`.

In the case of equivalence checking, two encoded processes or an encoded process and a composition of processes are compared. In the case of model checking, ACTL formulas written in a separate file are checked for their validity in the initial state of an encoded process or in the initial state of a composition of processes.

4 AN EXAMPLE OF VERIFICATION

EST has already been successfully used for the verification of some larger concurrent systems, for example Bounded Retransmission Protocol [9], but the description of that work exceeds the scope of this paper. Here, we will present another interesting verification, namely that of *Milner's simple distributed scheduler*, which is a standard scaleable benchmark for process algebra tools [5, 6, 7]. Informally, the system consists of k processes which are scheduled. The processes are organized in a ring. Each process starts the next process in the ring. Process k reactivates process 1. A process must never be reactivated before it has terminated.

The scheduler is constructed of one starter process S and k cyclers $C_1, ..., C_k$, where cycler C_i takes care of process i (Figure 4). The starter is only needed to start the first cycler at the beginning. Cycler C_i first receives signal $c_i?$ which indicates that it may start. It then activates process i via action $a_i!$. Next, it waits for termination of process i (action τ) and in parallel, it informs the next cycler via

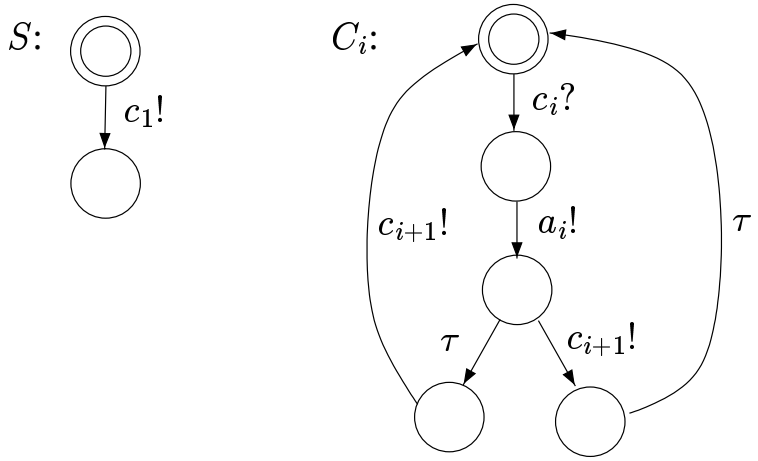

Figure 4: A description of starter S and cycler C_i

an action $c_{i+1}!$ that it may start. Finally, the cycler returns to its initial state. As usual in process algebras, a parallel execution of two or more actions is modelled as a possibility of process to perform all permutations of these actions.

The behaviour of the whole system is obtained by the parallel composition of process S and processes $C_1, ..., C_k$. The composition is weak observational equivalent and also testing equivalent to the process presented in Figure 5.

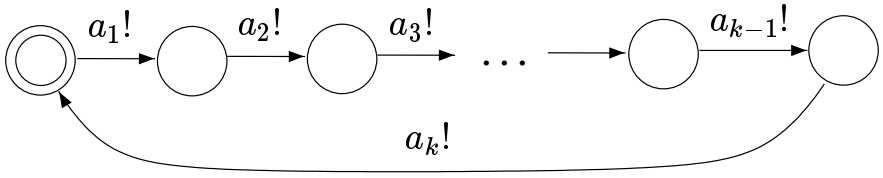

Figure 5: External behaviour of the system with k cyclers

Our results of the verification of simple distributed scheduler are shown in Figure 6. The states and transitions in the composition were counted. Because most transitions in the composition are transitions with action τ, the information about the number of transitions with actions other than τ is also presented. Currently, our program uses 32-bit numbers for counting and therefore we could not complete this counting for larger processes.

k	**states**	**transitions (without τ)**	**nodes in BDD**	**parallel composition**	**weak obs. equivalence**	**testing equivalence**
4	97	241 (32)	272	0.5s	0.3s	0.4s
8	3073	13825 (1024)	707	2.1s	0.8s	1.3s
12	73729	479233 (24576)	1245	7.1s	1.6s	2.9s
16	1572865	13369345 (524288)	1912	19.5s	2.8s	4.9s
20	31457281	(10485760)	2674	47.5s	4.1s	8.2s
24	603979777	-	3548	97.8s	5.8s	12.8s
28	-	-	4534	205.1s	8.0s	18.7s
32	-	-	5665	452.8s	10.6s	26.9s

Figure 6: The results of verifying simple distributed scheduler

The time required for the completion of parallel composition, weak observational equivalence and testing equivalence functions was measured on HP 715/100 with 128 MB RAM. The program was allowed to have at most 500000 BDD nodes at once, so that the total memory consumption never exceeded 32 MB. To get more

realistic times for equivalence checking, we always performed garbage collection after obtaining a parallel composition of processes.

The data presented in Figure 6 show the capability of symbolic methods to efficiently represent and manipulate processes containing millions of states and transitions. It is worth to say that EST is also able to verify a system with up to 28 cyclers with considerably stronger limitations: maximum 100000 BDD nodes at once and the memory consumption limited to 8 MB RAM, which includes the memory requirements of the user interface, BDD nodes and all neccessary cache tables. With such restrictions, EST needed 412.6s to obtain parallel compositon and then additionaly 9.9s to check weak observational equivalence.

Figure 7 compares our results with the reported results of other authors. Our times were obtained using the same computer and restrictions as stated before. It is hard to compare results of different tools because various computers were used. The results for tools named BB [7] and BDD [5] were obtained on much slower platforms, while the results for tool Severo [6] were measured on only a little bit slower computer than ours. Note that the computer we used for testing is relatively slow compared to the systems popular these days. For example, we tried EST with the same restrictions on an overclocked Pentium II 266 with Linux and got 7 times better results! EST completed parallel composition and equivalence checking for the system with 32 cyclers in 60.9s.

k	**BB [7] only weak obs. eq.**	**BDD [5] compos. & weak o. eq.**	**Severo [6] compos. & weak o. eq.**	EST **parallel composition**	EST **weak obs. equivalence**	EST **compos. & weak o. eq.**
4	0.02s	177s	-	0.5s	0.3s	0.8s
6	0.2s	345s	-	0.9s	0.5s	1.4s
8	1.2s	665s	-	2.1s	0.8s	2.9s
10	7.4s	1147s	-	4.1s	1.1s	5.2s
12	53s	1928s	-	7.1s	1.6s	8.7s
14	-	-	30.25s	11.7s	2.1s	13.8s
16	-	2972s	40.19s	19.5s	2.8s	22.3s
20	-	3259s	75.61s	47.5s	4.1s	51.6s
24	-	4927s	-	97.8s	5.8s	103.6s
28	-	-	-	205.1s	8.0s	213.1s
32	-	-	-	452.8s	10.6s	463.4s

Figure 7: Benchmarks for the problem of simple distributed scheduler

5 CONCLUSION

EST is a new tool for the verification of concurrent systems, which has not been widely presented yet. Its main advantages are the flexibility, portability and an efficient memory management. The latter is mainly the consequence of a well implemented BDD package with good garbage collection functions. EST is a relatively small package and thus easy understandable. In the future, we wish to add to EST a lot of additional capabilities, for example: introduction of data passing, introduction of other popular formalisms for describing concurrent systems, abstraction and minimization of processes, generation of diagnostics, e.g. counterexamples, ACTL model checking with fairness constraints, simulation of processes etc.

REFERENCES

[1] C. Bernardeschi et al. A Formal Verification Environment for Railway Signaling System Design. *Formal Methods In System Design*, 12(2):139–161, 1998.

[2] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient Implementation of a BDD Package. In *The Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[3] R. Cleaveland and M. Hennessy. Testing Equivalence as a Bisimulation Equivalence. *Formal Aspects of Computing*, 5:1–20, 1993.

[4] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[5] Reinhard Enders, Thomas Filkorn, and Dirk Taubner. Generating BDDs for symbolic model checking in CCS. *Distributed Computing*, 6(3):155–164, 1993.

[6] M. Ferrero and M. Cusinato. Severo: A symbolic equivalence verification tool. Tesi di Laurea. Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy, October 1994.

[7] Jan Friso Groote and Frits Vaandrager. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In *Automata, Languages and Programming, 17th International Colloquium*, pages 626–638, 1990. LNCS 443.

[8] Robert Meolic. EST Home Page. `http://www.el.feri.uni-mb.si/est/`.

[9] Robert Meolic. Checking correctness of concurrent systems behaviour. Master's thesis, Faculty of Electrical Engineering and Computer Science, Maribor, November 1999. In Slovene.

[10] Robert Meolic, Tatjana Kapus, and Zmago Brezočnik. Verification of concurrent systems using ACTL. In *Proceedings of the IASTED International Conference on Applied Informatics AI'2000*, pages 663–669, Innsbruck, Austria, February 2000.

[11] M. Sepešy, T. Kapus, and B. Horvat. Verifikacija komunikacijskega obnašanja sistemov z uporabo binarnih odločitvenih grafov. In *Proceedings of the Fifth Electrotechnical and Computer Science Conference ERK'96, Portorož, Slovenia*, volume B, pages 23–26, 1996.

[12] A. Časar, R. Meolic, Z. Brezočnik, and B. Horvat. Predstavitev logičnih funkcij z minimalnimi urejenimi binarnimi odločitvenimi grafi. *Elektrotehniški vestnik*, 59(5):299–307, December 1992.