SEQ.OPEN: A Tool for Efficient Trace-Based Verification*

Hubert Garavel and Radu Mateescu

Inria Rhône-Alpes / Vasy
655, avenue de l'Europe, F-38330 Montbonnot Saint Martin, France
{Hubert.Garavel,Radu.Mateescu}@inria.fr

Abstract. We report about recent enhancements of the Cadp verification tool set that allow to check the correctness of event traces obtained by simulating or executing complex, industrial-size systems. Correctness properties are expressed using either regular expressions or modal μ-calculus formulas, and verified efficiently on very large traces.

1 Introduction

Trace-based verification [3, 10, 15, 16] consists in assessing the correctness of a (software, hardware, telecommunication...) system by checking a set of event traces, i.e., chronological lists of inputs/outputs events sent/received by this system. Although trace-based verification is more limited than general verification on state graphs or Labelled Transition Systems (Ltss), it might be the only option for "real" systems that run as "black boxes", disclose none or little information about their internal state, and provide no means for an external observer to control or simply know about their branching structure (i.e., the list of possible transitions permitted in a given state). This is particularly true when the source code of these systems is not available, or cannot be instrumented easily.

The importance of trace-based verification is widely recognized in the hardware community, where traces might be the only information available during the execution of a circuit or the simulation of an Hdl model. In particular, there are recent efforts to standardize the use of temporal logics (e.g., Sugar [4] and ForSpec [2]) for trace-based verification.

Trace-based verification can be either *on-line* (i.e., verification is done at the same time the trace is generated) or *off-line* (i.e., the trace is generated first, stored in a file and verified afterwards). On-line verification avoids to store the trace in a file, but gets potentially slower if several correctness properties must be checked on the same trace, in which case it might be faster to generate the trace only once and perform all verifications off-line.

In this paper, we present a general solution for off-line trace-based verification, which is easily applicable to the traces generated by virtually any system. Traces are encoded on a very simple, line-based format. Correctness properties are specified using either regular expressions or μ-calculus formulas, and model

* This research was partially funded by Bull S.A. and the European Ist -2001-32360 project "ArchWare".

checked using dedicated tools of the widespread CADP verification tool set [9]. Notice that on-line trace-based verification could also be addressed by the CADP tools supporting on the fly verification, although this is beyond the scope of this paper.

2 Assumptions on Trace Structure and Representation

Following the "black box" testing paradigm, we assume that the internal state of the system is not available for inspection. Thus, a *trace* is defined as a (degenerated) LTS (S, A, T, s_0) consisting of a set S of *states*, a set A of *actions* (transition labels corresponding to the input/output communications of the system), a *transition relation* $T \subseteq S \times A \times S$, and an *initial state* $s_0 \in S$. Should (a part of) the internal state be observable, then this information could be encoded in the actions without loss of generality [14].

We then assume that the *length*, i.e., the number of states (and transitions) in a trace can be large (e.g., several millions), since traces can be produced by hour- or day-long simulation/execution of the system. In fact, the number of states can be as large as for classical explicit-state verification (with the difference that traces are particular LTSs with a tiny breadth and a huge depth).

We make no special assumption regarding actions. Their contents are unrestricted and may include any sequence of data, including variable-length values such as lists, sets, etc. We therefore represent actions as arbitrary-length character strings. As a consequence, the set A of all possible actions may be very large (or even unbounded), so that it might be prohibitive (or even infeasible) to enumerate its elements. Finally, we make no assumption of regularity or locality in the occurrence of actions. In the worst-case, a trace might contain as many different actions as it contains transitions.

In general, traces might be too large to fit into main memory entirely and must be stored in computer files instead. The CADP tool set provides a textual file format (the SEQ format) for representing traces. So far, this format was mostly used to display the counter-examples generated by CADP model-checkers, but, since this format satisfies the above assumptions, we decided to adopt it for trace-based verification as well. In practice, it is often convenient to store in the same file several traces issued from the same initial state. For this reason, a SEQ file consists of a set of finite traces, separated by the choice symbol "`[]`", which indicates the existence of several branches starting at the initial state. Each trace consists of a list of character strings (one string per line) enclosed between double quotes, each representing one action in the trace. The SEQ format also admits comments (enclosed between the special characters "`\001`" and "`\002`").

3 Principles of the Seq.Open Tool

All the CADP tools that operate on the fly (i.e., execution, simulation, test generation, and verification tools) rely upon the OPEN/CÆSAR [8] software framework. Due to the modularity and reusability brought by OPEN/CÆSAR, it was not needed to develop yet another model checker dedicated to traces encoded in the SEQ format. The proper approach was to design a new tool (named SEQ.OPEN) that connects the SEQ format to OPEN/CÆSAR, thus allowing all the OPEN/CÆSAR tools (including model checkers) to be applied to traces without any modification.

A central feature of OPEN/CÆSAR is its generic API (*Application Programming Interface*) providing an abstract representation for on the fly LTSs. This API clearly separates language-dependent aspects (translation of source languages into LTS models, which is done by OPEN/CÆSAR-*compliant compilers* implementing the API) from the language-independent aspects (on the fly LTS exploration algorithms built on top of the API). In a nutshell, the API consists of two types "LTS state" and "LTS label", equipped with comparison, hash, and print functions, and two operations computing the initial state of the LTS and the transitions going out of a given state.

SEQ.OPEN is an OPEN/CÆSAR-compliant compiler that maps a SEQ file onto the aforementioned API (see Figure 1). A set of n traces contained in a SEQ file can be viewed as an LTS with three types of states: *deadlock states* (terminating states, with 0 successors[1]); *normal states* (intermediate states, with 1 successor); and the *initial state* (common to all traces, with n successors). The user of SEQ.OPEN may decide to explore all the n traces, or only the i-th one ($1 \leq i \leq n$).

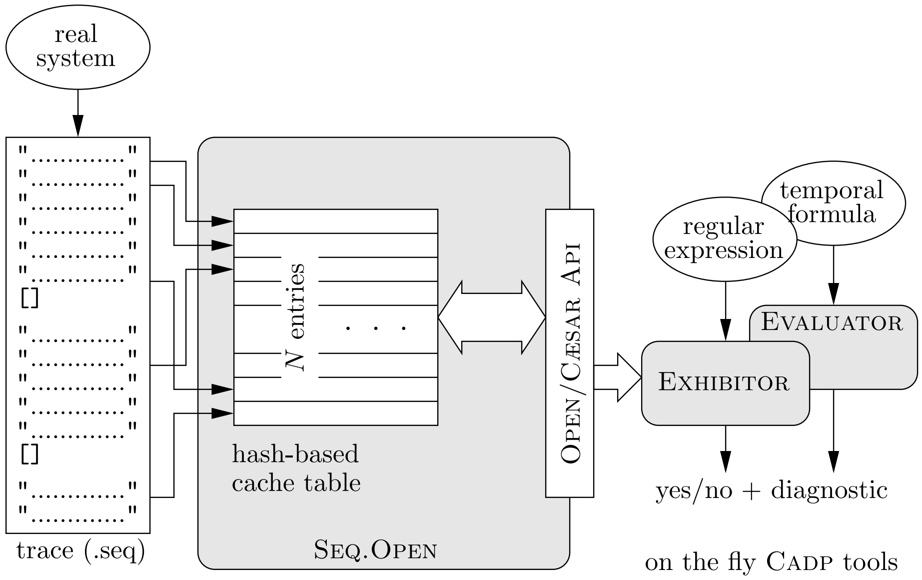

Fig. 1. The SEQ.OPEN Tool

[1] All traces end in a deadlock state. If necessary, a distinction can be made between *successful* and *abnormal* termination by considering the action of the last transition preceding the deadlock state.

An LTS label is implemented by SEQ.OPEN as an offset in the SEQ file (the offset returned by the POSIX function `ftell()` for the double quote opening the label character string). This representation is not canonical: The same label occurring at different places in the file is represented by different offsets.

States also are implemented as offsets. Each deadlock state is represented by the special offset -1. Each normal state s is represented by the offset of the label of the transition going out of s. The initial state is represented by the offset of the first label of the first trace to be considered. Contrary to labels, the state representation is canonical (up to graph isomorphism).

A transition (s_1, a, s_2) of the LTS is encoded by a couple (o_1, o_2), where o_1 is the offset of state s_1 (equal to the offset of label a) and o_2 is the offset of state s_2. The transition relation is implemented as follows. Deadlock states have no successors. For a normal state s with offset o_1, the offset o_2 of its successor is computed by positioning the file cursor at o_1 using the `fseek()` function, reading the character string of the transition label going out of s (possibly skipping comments), then taking for o_2 the offset returned by the `ftell()` function. For the initial state, the successors are computed only once at initialization. This is done using a preliminary scan of the SEQ file, unless the user wants to consider the first trace only (a frequent situation for which no preliminary scan is needed).

To reduce the time overhead induced by calls to `fseek()`, i.e., back and forth skips inside the SEQ file, we introduced a hash-based cache table similar to those used in BDD implementations. This table has a prime number N of entries, which is chosen by the user and remains constant regardless of the number of visited states/transitions. The table speeds up both (1) for a label a known by its offset, the computation of the character string of a, and (2) for a normal state s_1 known by its offset, the computation of the outgoing transition (s_1, a, s_2). Precisely, for a label (resp. normal state) with offset o_1, the table entry of index $o_1 \, mod \, N$ may contain the character string of label o_1 (resp. the character string of the transition label going out of state o_1, and the successor state offset o_2); if this entry is already occupied by another label (resp. state), its contents will be erased and replaced with information corresponding to the label (resp. state) with offset o_1 (this information will be computed using `fseek()` and `ftell()` as explained above).

The other operations of the OPEN/CÆSAR API are performed as follows. Comparison of states is simply an equality test of their offsets (since state offsets are canonical). Comparison of labels is done by comparing their character strings (since label offsets are not canonical), a comparison that is sped up when labels are already present in the cache.

4 Verification of Trace Properties

CADP provides two different tools for checking general properties over traces on the fly[2].

[2] For very specific, application-dependent properties, additional tools could be developed using the OPEN/CÆSAR environment.

Exhibitor allows to check linear-time properties expressed as regular expressions over traces. Individual actions in a trace are characterized by boolean formulas consisting of action predicates (plain character strings or Unix-like regular expressions matching several character strings) combined using boolean connectors such as negation ("~"), disjunction ("|"), and conjunction ("&"). Regular expressions over traces consist of these boolean formulas combined using regular operators such as concatenation (newline character "\n"), choice ("[]"), and iteration ("*" and "+"). A special "<deadlock>" operator characterizes deadlock states. Additional operators inspired from linear temporal logic are provided as shorthand notations: "<until> P" is equivalent to "(~P)*" and "<while> P <until> Q" is equivalent to "(P & ~Q)* \n Q".

Evaluator [13] allows to check branching-time properties expressed in alternation-free μ-calculus [7], a specification formalism for which efficient model checking algorithms exist [5] with a linear (time and space) complexity $O(|\varphi| \cdot (|S| + |T|))$, where $|\varphi|$ is the number of operators in the formula φ to be checked, and where $|S|$ and $|T|$ are the respective numbers of states and transitions in the Lts under verification. It was shown recently that on acyclic Ltss (which contain traces as a particular case), the alternation-free μ-calculus has the same expressive power as the full μ-calculus [11]. This result allows, in the case of acyclic Ltss, to benefit from the expressiveness of the full μ-calculus (which subsumes most temporal logics, including Ctl and Pdl [7], as well as Ltl and Ctl* [6]) still keeping model checking algorithms with a linear (rather than exponential) complexity. Furthermore, space complexity can be reduced down to $O(|\varphi| \cdot |S|)$ (still maintaining a linear complexity in time) by using specialized algorithms for checking alternation-free μ-calculus formulas on acyclic Ltss [11, 12].

Both Exhibitor and Evaluator provide diagnostic generation features, allowing to exhibit the prefix of the trace illustrating the truth value of a property.

5 Conclusion

We presented a working solution for model checking large event traces. Based upon the generic Open/Cæsar [8] framework, this solution relies on a new software tool, Seq.Open, which enables fast, cache-based handling of large traces stored in computer files. Combined with already existing components of the Cadp tool set (such as Exhibitor and Evaluator), Seq.Open allows to verify trace properties efficiently.

Due to the extreme simplicity of Seq.Open's line-based trace format, our solution is not "intrusive", in the sense that it is easily applicable to most existing systems without heavy reengineering.

In the setting of hardware systems, our solution was chosen by Bull for validating the traces produced by the Verilog simulation of the cache coherency protocol used in Bull's Novascale multiprocessor servers. This validation task, previously done by human reviewers, is now fully automated with good performances (7.4 million model checking jobs in 23 hours using a standard 700 MHz Pentium PC).

In the setting of software systems, our solution is used to analyze the traces produced by a multi-threaded virtual machine, which provides the runtime environment for executing the ArchWare description language for mobile software architectures.

As for future work, we plan to study trace-based verification algorithms that improve "locality", i.e., produce less faults in the Seq.Open cache table.

Acknowledgements The authors are grateful to Bruno Ondet (Inria/Vasy) for his contribution to the implementation of Seq.Open, and to Nicolas Zuanon and Solofo Ramangalahy (Bull) for their industrial feedback.

References

1. H. R. Andersen. Model checking and boolean graphs. *TCS*, 126(1):3–30, 1994.
2. R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In *TACAS'2002*, LNCS vol. 2280, pp. 296–311.
3. T. Arts and L-A. Fredlund. Trace Analysis of Erlang Programs. In *ACM SIGPLAN Erlang Workshop (Pittsburgh, PA, USA)*, 2002.
4. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The Temporal Logic Sugar. In *CAV'2001*, LNCS vol. 2102, pp. 363–367.
5. R. Cleaveland and B. Steffen. A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. *FMSD*, 2:121–147, 1993.
6. M. Dam. Ctl* and Ectl* as fragments of the modal μ-calculus. *TCS*, 126(1):77–96, 1994.
7. E. A. Emerson and C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *LICS'86*, pp. 267–278.
8. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *TACAS'98*, LNCS vol. 1384, pp. 68–84. Full version available as INRIA Research Report RR-3352.
9. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, 2002. Also available as INRIA Technical Report RT-0254.
10. K. Havelund, A. Goldberg, R. Filman, and G. Rosu. Program Instrumentation and Trace Analysis. In *Monterey Workshop (Venice, Italy)*, 2002.
11. R. Mateescu. Local Model-Checking of Modal Mu-Calculus on Acyclic Labeled Transition Systems. In *TACAS'2002*, LNCS vol. 2280, pp. 281–295. Full version available as INRIA Research Report RR-4430.
12. R. Mateescu. A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems. In *TACAS'2003*, LNCS vol. 2619, pp. 81–96.
13. R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *SCP*, 46(3):255–281, 2003.
14. R. De Nicola and F. W. Vaandrager. *Action versus State based Logics for Transition Systems*. In *Semantics of Concurrency*, LNCS vol. 469, pp. 407–419, 1990.
15. S. Vloavic and E. Davidson. TAXI: Trace Analysis for X86 Implementation. In *ICCD'2002 (Freiburg, Germany)*.
16. A. Ziv. Using Temporal Checkers for Functional Coverage. In *MTV'2002 (Austin, Texas, USA)*.