SPLIT: A Compositional LTL Verifier

Ariel Cohen[1], Kedar S. Namjoshi[2], and Yaniv Sa'ar[3]

[1] New York University, New York, NY. Email: arielc@cs.nyu.edu
[2] Bell Labs, Alcatel-Lucent, Murray Hill, NJ. Email: kedar@research.bell-labs.com
[3] Weizmann Institute of Science, Rehovot, Israel. Email: yaniv.saar@weizmann.ac.il

Abstract. This paper describes SPLIT, a compositional verifier for safety and general LTL properties of shared-variable, multi-threaded programs. The foundation is a computation of compact local invariants, one for each process, which are used for constructing a proof for the property. An automatic refinement procedure gradually exposes more local information, until a decisive result (proof/disproof) is obtained.

1 Introduction

Standard model checking algorithms prove safety properties through a reachability computation, computing an inductive assertion (the reachable states) that is defined over the full state vector. They often suffer from the state explosion problem [2]; for concurrent programs, this is manifested as an exponential growth of the state space with increasing number of components.

SPLIT is a new tool for the verification of shared-variable, asynchronous concurrent programs, which ameliorates state explosion through assertional (i.e., state-based) compositional reasoning, based on the classical Owicki-Gries method [13]. The foundation is a construction of a *vector* of local (i.e., per-process) inductive invariants, $\theta = (\theta_1, \theta_2, \ldots, \theta_N)$. The invariants are mutually interference-free—i.e., a move by one process does not violate the local invariant of another. Such a vector is called a *split-invariant*, as the conjunction of its components, $(\bigwedge_i \theta_i)$, is always a *globally inductive* invariant. Locality is enforced by syntactically limiting each process assertion to the variables visible to that process—i.e., the globally shared and process-local variables.

SPLIT implements a number of algorithms; together, they result in a fully automatic compositional model checker for general LTL properties.

1. A simultaneous least fixpoint algorithm [12], which computes the *strongest* split invariant vector (A split invariant is usually weaker than the set of reachable states.)
2. A safety refinement method [4], which achieves completeness by gradually "exposing" local predicates (i.e., encoding them as shared variables)
3. A compositional algorithm which verifies arbitrary LTL properties [5], based on a split invariance computation and a counter-example based refinement scheme
4. A recently developed compositional algorithm [6], for the verification of progress properties under general fairness assumptions

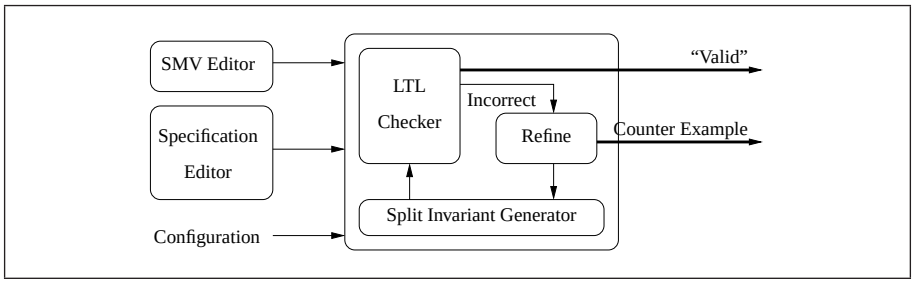

Fig. 1. The architecture of SPLIT.

Experimental results support the hypothesis that local reasoning allows verifying significantly larger systems without running into state explosion, and can result in order-of-magnitude improvements in run-time over monolithic model checking. It is interesting that basic local reasoning suffices for the proofs for many protocols, without a need for refinement. In many other cases, a proof/disproof is obtained by exposing a limited amount of local state, validating the basic argument for compositional verification. SPLIT has been used to verify protocols for cache coherence and mutual exclusion. To the best of our knowledge, this is the first tool to implement a fully automated compositional method for both safety and liveness properties.

2 Architecture and Selected Features

SPLIT is built using JTLV [14] – a BDD-based framework for developing temporal verification algorithms. Fig. 1 sketches the architecture of SPLIT. It takes three inputs: an SMV [11] program, an LTL specification, and a configuration. The main part of SPLIT is built up from three components: a unit that generates the split invariant, a verifier for LTL properties, and a unit to compute refinements.

Verification is implemented differently for safety and liveness properties. For a safety property, the algorithm (from [12,4]) first checks if the split invariant implies the property. If it does, then the property is valid; otherwise, the refinement unit heuristically selects local predicates and "exposes" them. (A local predicate p is exposed by adding an auxiliary shared variable, say x_p, in such a manner that the invariant $(x_p \equiv p)$ is maintained.) Exposing local state strengthens the split invariant in the next iteration; the process is repeated until the property is proved or no additional refinements can be performed. In the latter case, SPLIT generates a valid counter-example trace.

For a liveness property, the algorithm (from [5,6]) uses the computed split invariant to construct abstract forms of each process. It checks if the liveness property is satisfied by *all* abstract processes, using a standard LTL checker from JTLV. If all checks succeed, the property is valid; otherwise, a counter example trace is extracted. If the trace is spurious, it is used by a refinement procedure to expose local predicates. This process is repeated until either the property is proved or a valid counter-example trace is found.

Example	Property	N	JTLV		SPLIT	
			Nodes	Time	Nodes	Time
SEMAPHORE (+COUNT)	mutual exclusion	10	1.2M	10.4	160k	0.3
	– valid –	12	1.8M	440	252k	0.5
PETERSON's	mutual exclusion	5	6.9M	16	3.7M	8.1
	– valid –	6	91M	509	43.8M	172
BAKERY	mutual exclusion	7	2.9M	65	7.8M	20
	– valid –	8	11M	844	27M	97
SZYMANSKI	mutual exclusion	3	68k	0.1	788k	2.4
	– valid –	4	395k	0.6	3.8M	10
SEMAPHORE	individual starvation-freedom	10	21M	24	371k	1.1
	– Counter example –	20	over 20 minutes		2.1M	9
BAKERY	individual starvation-freedom	3	300k	0.3	1.2M	2.5
	– Valid –	4	11.6M	93	14.6M	52
DINING-PHIL	individual starvation-freedom	9	9.1M	63	4.1M	8.6
	– Valid –	10	25M	421	8.6M	18

Table 1. Characteristic experimental results. (More results are on the tool web page.)

The user interface for SPLIT allows the user to expose local variables, which can help reduce the number of refinement steps. The counter-examples produced are augmented with refinement predicates that express the changes to the state. SPLIT is implemented in about 9000 lines of Java, of which at least half is for the user interface. It relies on standard BDD libraries written in C and Java. More information, including a collection of examples, can be found at `www.wisdom.weizmann.ac.il/~saar/split`.

3 Experimental Results

We have used SPLIT to verify safety and liveness properties for a number of multi-threaded protocols for mutual exclusion and cache coherence. Table 1 presents characteristic results of comparing SPLIT with the (monolithic) LTL model checker in JTLV. Both were configured to use the CUDD BDD library. In the table, "N" is the number of processes, "Nodes" is the peak number of BDD nodes generated, and "Time" is the runtime in seconds.

In nearly all cases (SZYMANSKI being the exception) SPLIT obtained better run-times, sometimes showing as much as one or two orders of magnitude improvement. Improvement in memory consumption, which is proportional to the number of peak BDD nodes, is not as clear-cut: for BAKERY, for which it obtains better run-times, SPLIT requires more memory. SPLIT was also able to verify much larger systems than the monolithic model checker; for instance, it proves SEMAPHORE for $N = 64$ where JTLV ran out of memory already for $N = 24$. The performance of NUSMV [1] on most of these examples was inferior to that of JTLV and SPLIT even after disabling the conjunctive partitioning. This appears to be because NUSMV is optimized for verifying synchronous systems and we therefore do not include the results obtained by it.

4 Related Work and Conclusions

SPLIT mechanizes assertional (i.e., state-predicate based) compositional reasoning in the style of the seminal Owicki-Gries proof method. Thread-modular reasoning [8] computes the strongest split invariant with an explicit-state algorithm, but it does not include a refinement step and is therefore incomplete. An alternative automated compositional method is based on behavioral (i.e., path-based) reasoning, and uses automaton learning algorithms [10,9]. Experimental results with this method have been mixed [3]: in many cases, monolithic verification is faster; likely due to the complexity of the automaton learning algorithms. Assertional reasoning has a simple implementation, even for the analysis of general LTL properties, and the experiments with SPLIT show a clear advantage over monolithic verification on a number of protocols.

There are several potential improvements and extensions being investigated in current work. One focus is on coupling counter-example generation with refinement; the current implementation uses whichever trace is provided by the JTLV model checker. Another focus is on parallel and distributed implementations [7], as the compositional reasoning calculations can be easily parallelized.

References

1. Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: a new symbolic model verifier. In *CAV*, pages 495–499, 1999.
2. E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *PODC*, pages 294–303, 1987.
3. J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke. Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In *ISSTA*, pages 97–108, 2006.
4. A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. In *CAV*, volume 4590 of *LNCS*, pages 55–67. Springer, 2007.
5. A. Cohen and K. S. Namjoshi. Local proofs for linear-time properties of concurrent programs. In *CAV*, volume 5123 of *LNCS*, pages 149–161. Springer, 2008.
6. A. Cohen, K. S. Namjoshi, and Y. Sa'ar. A dash of fairness for compositional reasoning. submitted to CAV 2010.
7. Ariel Cohen, Kedar S. Namjoshi, Yaniv Sa'ar, and Lenore D. Zuck. Symbolic model checking on multi-core processors. Technical report, Bell Laboratories, 2009.
8. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, volume 2648 of *LNCS*, pages 213–224, 2003.
9. D. Giannakopoulou and C. S. Pasareanu. Learning-based assume-guarantee verification (tool paper). In *SPIN*, volume 3639 of *LNCS*, pages 282–287, 2005.
10. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE*, pages 3–12, 2002.
11. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
12. K. S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *VMCAI*, volume 4349 of *LNCS*, 2007.
13. S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
14. Amir Pnueli, Yaniv Sa'ar, and Lenore D. Zuck. JTLV: A framework for developing verification algorithms. web: `http://jtlv.sourceforge.net/`.