Model Checking in Bits and Pieces

Ariel Cohen[1], Kedar S. Namjoshi[2⋆], Yaniv Sa'ar[3],
Lenore D. Zuck[4⋆⋆], and Katya I. Kisyova[4]

[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
[4] University of Illinois at Chicago, Chicago, IL. Email:{lenore,kkisyo2}@cs.uic.edu

The central thesis explored in this paper is that parallel model checking is easily achieved through compositional methods. This may strike the reader as being self-evident. Nevertheless, it does not appear to have been considered before. The justification comes from the following observations: (1) the efficacy of a parallel program is inversely proportional to the amount of synchronization and communication across processors, and (2) compositional methods break up the analysis into nearly-independent parts; in each part, a program component is analyzed with limited knowledge of the internal state of other components. For loosely coupled programs, therefore, one may expect compositional methods to work well, and to further benefit from parallelization.

To test this thesis, we have implemented a parallel version of an assertional compositional model checking method. A brief introduction is necessary in order to explain the parallel algorithm. The basis is a computation of a vector of assertions, $\theta = (\theta_1, \theta_2, \ldots, \theta_n)$, with the following properties: (1) each θ_i is an assertion defined over the state visible to process i; (2) θ_i is a local inductive invariant for process i; and (3) the conjunction $(\bigwedge i : \theta_i)$ is a *globally inductive invariant*. We call such a vector a "split invariant". These conditions correspond to the interference-free proof outlines of Owicki and Gries [13], and the thread-modular safety proofs of Flanagan and Qadeer [7].

It was shown in [12] that there is a strongest split invariant, which can be computed symbolically (for instance, with BDDs) through a simultaneous fixpoint computation. It is well known (cf. [11, 13]) that a split invariant may be too weak to prove a safety property, and that this can be remedied by adding auxiliary shared variables to expose internal process state. Heuristics for discovering relevant auxiliary variables are developed in [2]; iterating these heuristics makes the method complete. Compositional methods have been developed to verify general liveness and fairness properties using split invariance [3, 4].

The fixpoint computation for the strongest split invariant is the basis for parallelization. In a fixpoint step, every component of θ is *simultaneously* updated as follows.

$$\theta'_i = I_i \ \vee \ sp_i(T_i, \theta_i) \ \vee \ (\vee k : k \neq i : sp_i(\overline{T}_k \ \wedge \ unchanged(L_i), \theta_i)) \qquad (1)$$

⋆ Kedar Namjoshi's research was supported, in part, by National Science Foundation grant CCR-0341658.

⋆⋆ This material was based on work supported by the National Science Foundation, while Lenore Zuck was working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this article are those of the author and do not necessarily reflect the views of the National Science Foundation.

The update for component i includes: (1) the initial condition I_i of process i; (2) closure under transitions T_i of process i, represented by the strongest post-condition term $sp_i(T_i, \circ)$; and (3) closure under interference from all other processes, given by the $sp(\overline{T_k} \wedge unchanged(L_i), \circ)$ terms. The interference from process k is represented by the *summary transition*, $\overline{T}_k$, defined as $(\exists L_k, L'_k : T_k \wedge \theta_k)$. It restricts process k's transition to states satisfying θ_k and quantifies out the local state variables L_k to obtain the net effect on the shared state of a move by process k.

The key to parallelization is the *chaotic iteration theorem* from [5]. It allows the simultaneous fixpoint to be computed lazily, via *any fair schedule* of the individual updates. To parallelize the computation, therefore, we simply distribute the fixpoint updates across processors, as shown in Figure 1, where processor i computes the component θ_i.

local variables $\theta_i, \overline{T}[1..n]$;

$\theta_i := I_i$; `// initialize` θ_i
forall $k : k \neq i : \overline{T}[k] := false$; `// set known summaries to empty`
while (not globally converged){
  while (θ_i does not stabilize){`// compute fixpoint update`
    $\theta_i := \theta_i \vee sp_i(T_i, \theta_i) \vee (\vee k : k \neq i : sp_i(\overline{T}[k] \wedge unchanged(L_i),\ \theta_i))$
  }
  asynchronously broadcast new summary $\overline{T}[i] = (\exists L_i, L'_i : T_i \wedge \theta_i)$;
}

Fig. 1. Outline of the computation for Processor i. Vector $\overline{T}$ represents summary transitions. A secondary thread receives updates for $\overline{T}$ entries asynchronously from other threads.

This is a general schema for parallelization, which can be implemented in many ways. One axis is symbolic (BDD-based, say) vs. explicit-state. A symbolic implementation would broadcast new summaries as symbolic terms. Another axis is shared-memory vs. distributed. The advantage of a distributed implementation is that each processor has its own memory space, while the overall memory is shared between processors in a multi-core implementation. A third axis is the organization of the broadcast, which is perhaps more of an issue in the distributed implementation.

For now, we have experimented with a symbolic (BDD-based), multi-core, shared memory implementation. The experiments use parameterized protocols, so as to easily vary the size of the state space. They also examine the role played by loose coupling and load balancing. MUXSEM (semaphore-based mutual exclusion) is uniform and balanced. SZYMANSKI is also uniform and balanced but more tightly coupled (summary BDDs are larger). German's cache coherence protocol has an unbalanced client-server nature, but could be rebalanced for analysis. PETERSON's exclusion protocol requires significantly complex summary transitions. Except PETERSON, other protocols need a small amount of auxiliary state to be exposed. These protocols are hard to handle by monolithic model checking for larger instances, as shown in [2]. Sequential and parallel algorithms are both implemented in Java using JTLV [14] and a JAVA BDD package based on BUDDY.

The experiments compare sequential and parallel calculations for split invariance. It should be noted that the sequential calculation is itself significantly faster than a non-compositional reachability calculation. The results from Table 1 show that parallelization is quite effective. The "efficiency" number (Eff.) is the ratio of the actual speed-up to the ideal speed-up. It is greater than 1 in some cases, possibly due to favorable cache usage. Those programs which are more loosely coupled than others have better parallel performance (e.g., MUXSEM and GERMAN over SZYMANSKI and PETERSON). The results support the main thesis, that parallelization of compositional methods can be very effective. Compositionality is not a panacea, however: tightly coupled programs are more difficult to analyze compositionally and may not show much improvement with parallelization. The expectation, though, is that most distributed protocols and shared-memory asynchronous programs are sufficiently loosely coupled.

Parallel model checking is an active research area for both explicit-state and symbolic algorithms ([9, 15] and [1, 6, 8, 10] are representative). Nonetheless, prior work has not explored the uses of compositionality for parallel model checking. Our early experience is that the parallel implementation is particularly simple, with significant benefit for loosely coupled programs.

References

1. G. Cabodi, P. Camurati, A. Lioy, M. Poncino, and S. Quer. A parallel approach to symbolic traversal based on set partitioning. In *CHARME*, pages 167–184, 1997.
2. A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. In *CAV*, volume 4590 of *LNCS*, pages 55–67. Springer, 2007.
3. 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.
4. A. Cohen, K. S. Namjoshi, and Y. Sa'ar. A dash of fairness for compositional reasoning. In *CAV*, 2010. (to appear).
5. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence & Programming Languages*, Rochester, NY, ACM SIGPLAN Not. 12(8):1–12, Aug. 1977.
6. J. Ezekiel, G. Lüttgen, and G. Ciardo. Parallelising symbolic state-space generators. In *CAV*, volume 4590 of *LNCS*, pages 268–280. Springer, 2007.
7. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, volume 2648 of *LNCS*, pages 213–224, 2003.
8. O. Grumberg, T. Heyman, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. *Formal Methods in System Design*, 29(2):157–175, 2006.
9. G. J. Holzmann and D. Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Trans. Software Eng.*, 33(10):659–674, 2007.
10. S. K. Iyer, D. Sahoo, E. A. Emerson, and J. Jain. On partitioning and symbolic model checking. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(5):780–788, 2006.
11. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2), 1977.
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.

	sequential	2 cores			4 cores			8 cores		
N	**Time**	**Time**	**Speedup**	**Eff.**	**Time**	**Speedup**	**Eff.**	**Time**	**Speedup**	**Eff.**
MUXSEM										
512	27	16	1.68	0.84	8.3	3.25	0.81	4.8	5.6	0.70
1024	117	65.8	1.77	0.88	34.8	3.3	0.82	19.2	6.1	0.76
1536	360	203	1.77	0.88	112	3.2	0.80	65	5.5	0.69
2048	561	314	1.80	0.90	165	3.4	0.85	92	6.1	0.76
SZYMANSKI										
5	3.1	2.4	1.29	0.64	1.6	1.93	0.48	1.1	2.81	0.35
6	20.5	11.6	1.76	0.88	6.5	3.15	0.78	4.4	4.65	0.78
7	130	73.5	1.76	0.88	41	3.17	0.79	23.7	5.48	0.78
8	564	302	1.87	0.93	163	3.46	0.86	93	6.06	0.76
9	2896	1362	2.12	1.06	739	3.91	0.97	492	5.88	0.73
GERMAN										
8	185	78	2.37	1.19	44	4.20	1.05	31	5.96	0.74
9	489	234	2.08	1.04	126	3.88	0.97	76	6.40	0.80
10	1076	511	2.10	1.05	268	4.00	1.00	164	6.56	0.82
11	2867	1310	2.18	1.09	691	4.14	1.03	385	7.44	0.93
12	over BDD limit	3505	-	-	1819	-	-	1013	-	-
PETERSON**'s**										
4	0.7	1.1	0.63	0.32	0.9	0.77	0.19	0.9	0.77	0.17
5	8.5	6.7	1.26	0.63	4	2.1	0.50	3.2	2.6	0.52
6	183	109	1.68	0.84	66	2.77	0.70	46	3.98	0.66

Table 1. Test results for MUXSEM, SZYMANSKI, GERMAN and PETERSON's. The machine is a dual-quad-core AMD Opteron (8 cores total), with $1.1GHz$ clock-speed, $512KB$ cache, and $32G$ RAM.

14. A. Pnueli, Y. Sa'ar, and L. D. Zuck. JTLV- a framework for developing temporal verification algorithms. In *CAV*, 2010. (to appear) `http://jtlv.sourceforge.net/`.
15. U. Stern and D. L. Dill. Parallelizing the Murφ verifier. *Formal Methods in System Design*, 18(2):117–129, 2001.