Parallelizing A Symbolic Compositional Model Checking Algorithm

Ariel Cohen

with Kedar Namjoshi (Bell Labs), Yaniv Sa'ar (Weizmann),
Lenore Zuck (UIC/NSF) and Katya Kisyova (UIC)

HVC – October 2010

Essentials

Symbolic BDD-based model checking revolutionized the field.
Parallelizing these algorithms, however, has proven difficult.

Essentials

Symbolic BDD-based model checking revolutionized the field. Parallelizing these algorithms, however, has proven difficult.

We show how a local reasoning method gives a rise to simple and effective algorithms for parallel, symbolic model checking.

Intuition:

- Many concurrent programs are formed of loosely coupled processes (this is a good design principal);
- Local (compositional) analysis works well for loosely-coupled processes. Proofs can be carried out with limited mutual knowledge of internal process state;

Old News

Let $P := P_1 \parallel P_2 \parallel ... \parallel P_N$ represents an asynchronous composition of N threads/processes communicating by shared memory.

- The reachable state space often grows as 2^N;
- Checking safety properties is PSPACE-complete in N;

Modular reasoning is essential for scalability.

Global Proofs of *Always*(φ)

- Let $P := P_1 \parallel P_2 \parallel \ldots \parallel P_N$;
- Proof-Theoretic Method
  - guess a state assertion θ;
  - check that θ is inductive;
  - check $\theta \longrightarrow \varphi$;
- How to find an appropriate θ?
  - Model Checking method
    - ⋆ compute reachable states, *Reach*, as a least fixpoint;
    - ⋆ take θ to be *Reach*;
    - ⋆ check subset relation *Reach* $\longrightarrow \varphi$;

Local Proofs of *Always*(φ)

Guess a vector of assertions, $(\theta_1, \theta_2, ..., \theta_N)$ such that

- Locality: θ_i is defined on variables visible to process P_i (i.e., the shared variables X and local variables L_i);
- Step: θ_i is an inductive invariant for process P_i;
- Non-interference: actions of P_j preserve θ_i, assuming θ_j;

This is an "assume-guarantee" system – a reformation of Owicki & Gries [1976]

Theorem: The last two constraints are equivalent to the statement that $\theta_1 \wedge \theta_2 \wedge ... \wedge \theta_N$ is an inductive invariant of the full program.

This motivates the name "split invariant" for $(\theta_1, \theta_2, ..., \theta_N)$.

Computing the Split Invariant

FOREACH i : initialize θ_i to the initial states of P_i;
REPEAT
    FOREACH i :
        increase θ_i *by computing successors in* P_i;
    FOREACH i :
        FOREACH $j \neq i$:
            increase θ_i *through* **interference** *from* θ_j;
UNTIL (convergence of all θ_i)

Interference:

(Intuitively) θ_j gives θ_i a summary of the states it found, in terms of the shared variables

Computing the Split Invariant

FOREACH i : initialize θ_i to the initial states of P_i;
REPEAT
    FOREACH i :
        increase θ_i *by computing successors in* P_i;
    FOREACH i :
        FOREACH $j \neq i$:
            increase θ_i *through* **interference** *from* θ_j;
UNTIL (convergence of all θ_i)

IF $(\wedge i : \theta_i \rightarrow \varphi)$
    THEN "φ **is an invariant**";
    ELSE "**unable to prove** φ";
END

Computing the Split Invariant

FOREACH i : initialize θ_i to the initial states of P_i;
REPEAT
    FOREACH i :
        increase θ_i *by computing successors in* P_i;
    FOREACH i :
        FOREACH $j \neq i$:
            increase θ_i *through* **interference** *from* θ_j;
UNTIL (convergence of all θ_i)

Does local reasoning always work?

We have an efficient method for exposing local state by adding auxiliary variables (CAV 07).

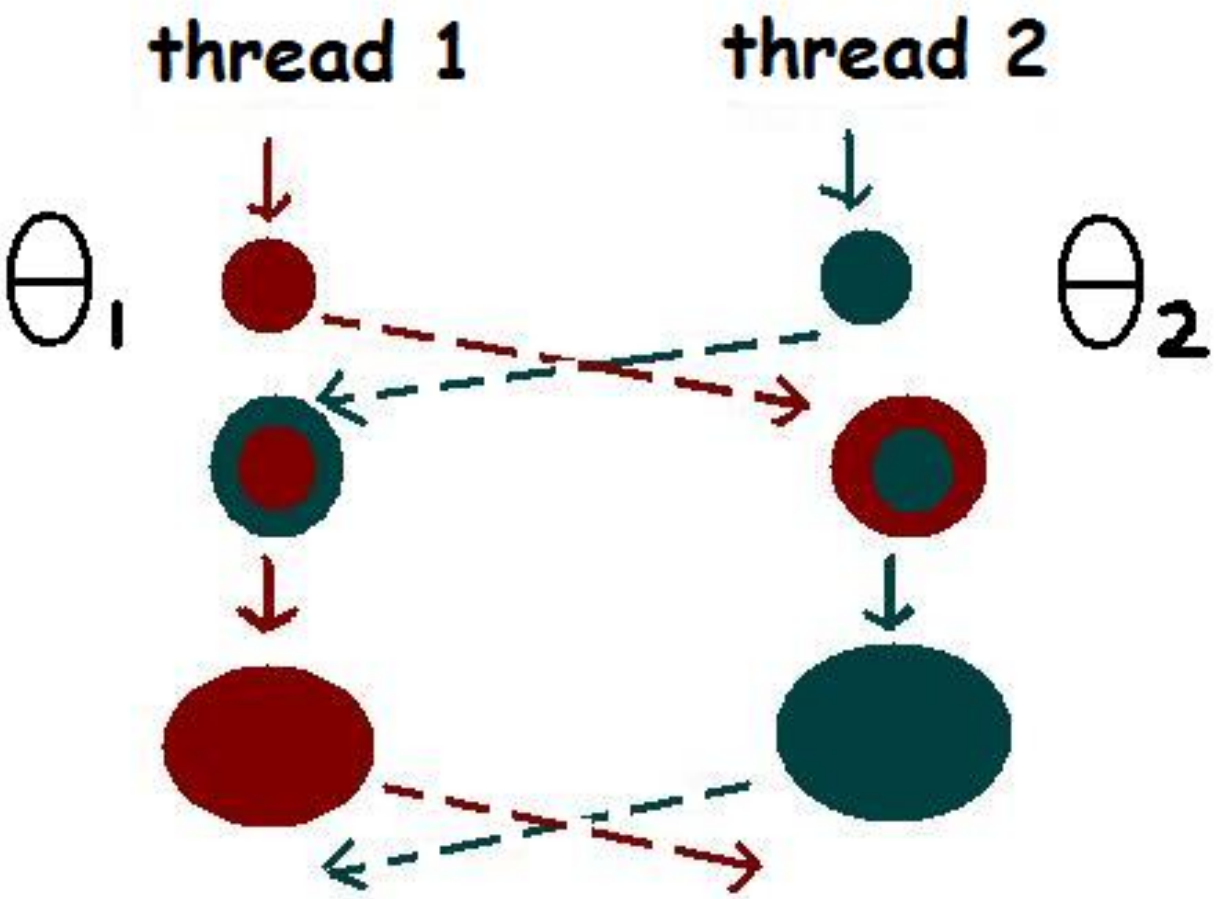
Local Reasoning Works Well

Example	N	Reachability		Local Reasoning	
		Nodes	Time (s)	Nodes	Time (s)
Semaphore (Safety)	10	1.2M	10.4	160k	**0.3**
	12	1.8M	440	252k	**0.5**
Peterson's (Safety)	5	6.9M	16	3.7M	**8.1**
	6	91M	509	43.8M	**172**
Bakery (Safety)	7	2.9M	65	7.8M	**20**
	8	11M	844	27M	**97**
Szymanski (Safety)	3	68k	**0.1**	788k	2.4
	4	395k	**0.6**	3.8M	10
Semaphore (Liveness)	10	21M	24	371k	**1.1**
	20	over 20 minutes		2.1M	**9**
Bakery (Liveness)	3	300k	**0.3**	1.2M	2.5
	4	11.6M	93	14.6M	**52**
Dining-Phil (Liveness)	9	9.1M	63	4.1M	**8.6**
	10	25M	421	8.6M	**18**

Introducing Parallelism

- The split invariance calculation is a simultaneous fixpoint over the vector $(\theta_1, \theta_2, ..., \theta_N)$;
- By the chaotic iteration theorem (Cousot & Cousot 1977), any evaluation schedule generates the final answer, so long as it is fair in the limit;
- This gives rise to a highly non-deterministic parallel algorithm:
  - each component θ_i is computed by a **separate thread** (i);
  - the effect of each process is broadcast to other processes;
  - repeated until global convergence;

Image of Parallel Algorithm



Problem: Synchronizing BDD Computations

- We use BDDs;
- Concurrent BDD access does not scale any better than the distributed scheme described next;

Problem: Synchronizing BDD Computations

- We use BDDs;
- Concurrent BDD access does not scale any better than the distributed scheme described next;
- In our implementation each machine (thread) i has **its own BDD stores** for computing θ_i;
- The only BDDs which must be exchanged between machines are "summary" transitions (only over shared variables) used for interference calculations;
- This results in replication of BDDs, and BDD copying when summaries are exchanged;

Copying BDDs

- BDD stores have to agree on the ordering of the shared variables;
- This might results in replication of common terms;
- Cost of copying summary transitions – depends on the amount of shared state;

Copying BDDs

- BDD stores have to agree on the ordering of the shared variables;
- This might results in replication of common terms;
- Cost of copying summary transitions – depends on the amount of shared state;

Experiments on a number of protocols show a speedup of roughly $5.5 - 7.5$x on a machine with 8 cores.

Experiments – Muxsem

	Seq.	4 cores			8 cores		
N	Time	Time	Speedup	Eff.	Time	Speedup	Eff.
512	27	8.3	3.25	0.81	4.8	5.6	0.70
1024	117	34.8	3.3	0.82	19.2	6.1	0.76
1536	360	112	3.2	0.80	65	5.5	0.69
2048	561	165	3.4	0.85	92	6.1	0.76

	Sequential	Parallel	
N	number of BDD nodes	number of BDDs nodes	BDD inc.
512	19.5M	19.8M	1%
1024	81.0M	82.0M	1%
1536	219.0M	221.0M	1%
2048	335.0M	342.0M	2%

Experiments – Szymanski

	sequential	4 cores			8 cores		
N	**Time**	**Time**	**Speedup**	**Eff.**	**Time**	**Speedup**	**Eff.**
6	20.5	6.5	3.15	0.78	4.4	4.65	0.78
7	130	41	3.17	0.79	23.7	5.48	0.78
8	564	163	3.46	0.86	93	6.06	0.76
9	2896	739	3.91	0.97	492	5.88	0.73

	Sequential	Parallel	
N	**number of BDD nodes**	**number of BDDs nodes**	**BDD inc.**
6	4.8M	6.9M	43%
7	16.1M	23M	42%
8	49M	73M	48%
9	141M	216M	53%

Experiments – German's

	sequential	4 cores			8 cores		
N	**Time**	**Time**	**Speedup**	**Eff.**	**Time**	**Speedup**	**Eff.**
8	185	44	4.20	1.05	31	5.96	0.74
9	489	126	3.88	0.97	76	6.40	0.80
10	1076	268	4.00	1.00	164	6.56	0.82
11	2867	691	4.14	1.03	385	7.44	0.93
12	over BDD limit	1819	-	-	1013	-	-

The number of BDD nodes is the same for sequential and parallel.

Finishing Up

- Local-reasoning is effective for loosely-coupled protocols. Sequentially, sometimes faster by a few orders of magnitude than reachability;
- Parallelism can be exploited for a further speed-up ($5.5 - 7.5$x on an 8 cores machine);
- Both single-machine (multi-core) and distributed implementations possible;
- The parallel implementation can be extended easily to check general temporal properties;

Open Questions

- We have not yet considered parallelizing refinement steps;
- A particularly challenging problem is the implementation of an efficient, thread-safe BDD package;
- Combine with other techniques;
- An open task: add parallelism to SPLIT;

Closely Related Work

- Owicki-Gries 1976, Lamport 1977 [local proof rules];
- Moon-Kukula-Shiple-Somenzi 1999 [local reasoning for synchronous models];
- Grumberg-Heyman-Ifergan-Schuster 2005, 2006 [distributed model checking for synchronous models];
- Sahoo-Jain-Iyer-Dill-Emerson 2005 [multi-core model checking for synchronous models];
- Ezekiel-Luettgen-Cardo 2008 [multi-core, symbolic model checking for asynchronous models];
- Stern-Dill 2001, Holzmann-Bosnacki 2007 [parallel, explicit-state model checking for asynchronous models];

(The parallel algorithms all compute exact reachability)