25 Years of Compositionality Issues in CADP: An Overview

Hubert Garavel *and colleagues*

Inria Grenoble – LIG

and Saarland University (part-time)

http://convecs.inria.fr

Outline

- 1. CADP in a nutshell
- 2. Compositionality issues:
  - 2.a. Types and data structures
  - 2.b. Concurrency I
  - 2.c. Concurrency II
- 3. Conclusion

1. CADP in a nutshell

CADP

- A modular toolbox for concurrent systems
- Research work at the crossroads between:
  - concurrency theory
  - formal methods
  - computer-aided verification
  - compiler construction
- A long-run effort:
  - development of CADP started in the mid 80s
  - initially: 2 tools
    - CAESAR: LOTOS → Petri nets with data → LTSs
    - ALDEBARAN: minimization and comparison of LTSs modulo bisimulations
  - today: 50 tools

Main features of CADP

- Formal specification languages
- Verification techniques:
  - Model checking (modal μ-calculus)
  - Equivalence checking (bisimulations)
  - Visual checking (graph drawing)

  using
  - Reachability analysis
  - On-the-fly verification
  - Compositional verification
  - Distributed verification
  - Static analysis
- Other features:
  - Rapid prototyping
  - Step-by-step simulation
  - Test-case generation
  - Performance evaluation

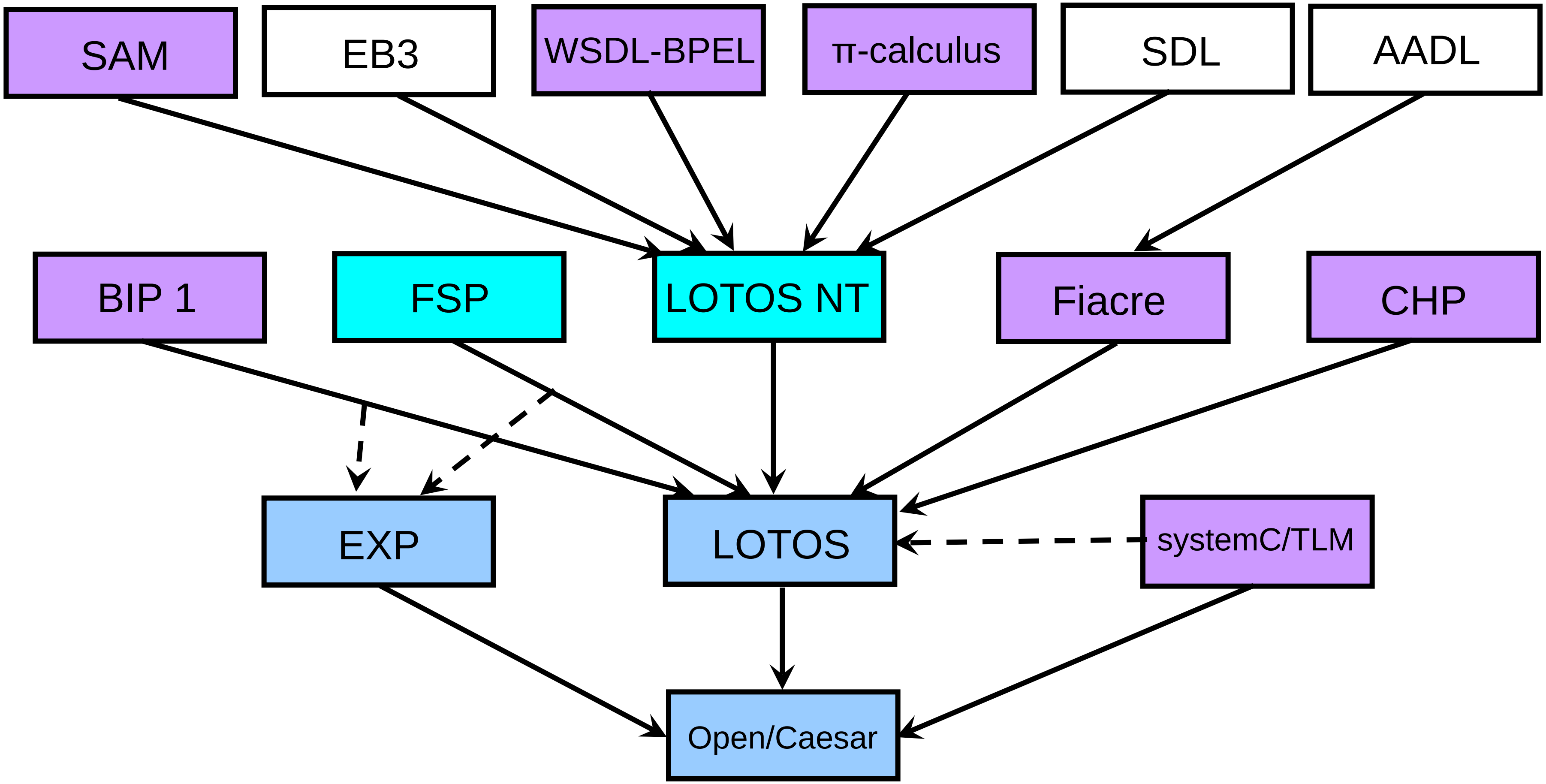
MCL (Model Checking Language)

- Transition labels carry data values "SEND !2 !true !3.14"
- The MCL temporal logic handles these values
  - Base = alternation-free modal μ-calculus + fairness PDL-Δ operators to express cyclic behaviour
  - Action formulas: value extraction, value matching
  - Path formulas: if-then-else, case, let, for, while, etc.
  - State formulas: fixed points parameterized with typed variables, if-then-else, case, let, quantifiers over finite domains
- MCL supported by the EVALUATOR 4.0 model checker of CADP

LNT (LOTOS New Technology)

- LOTOS NT:
  - a process calculus disguised as an imperative language
- Features:
  - typed variables, explicit assignment, pattern matching
  - symmetric sequential composition (≠ action prefix)
  - usual control structures: **if-then-else**, **case**, **while**, **for**
  - multiway rendezvous, choice, parallel composition
- Implemented by translation to LOTOS

Languages connected to CADP



2.a. Compositionality issues: Types and data structures

LOTOS abstract data types

```
type SimpleBoolean is
   sorts Bool
   opns false : → Bool
        true : → Bool
        not : Bool → Bool
   eqns
        not (false) = true;
        not (true) = false;
endtype
```

- based on the ACT-ONE language
- initial algebra semantics: $\Sigma_{bool} = \{\text{false}, \text{true}\}$

LOTOS type imports

```
type BasicBoolean is
  sorts Bool
  opns false : → Bool
         true : → Bool
endtype
```

+

```
type SimpleBoolean is BasicBoolean
  opns  not : Bool → Bool
  eqns
        not (false) = true;
        not (true) = false;
endtype
```

- Types can import other types
  - circular dependencies forbidden
  - DAG-like dependencies allowed
  - semantics: union of sorts, operations, and equations

Issue #1: Algebra expansion

```
type SimpleBoolean is
  sorts Bool
  opns false : → Bool
         true : → Bool
         not : Bool → Bool
  eqns
         not (false) = true;
         not (true) = false;
endtype
```

\+

```
type MyBoolean is SimpleBoolean
  opns other: → Bool
  eqns
         not (not (other)) = other;
endtype
```

- Σ_{bool} = {false, true, **other**, **not (other)**}
- MyBoolean "corrupts" SimpleBoolean
  - and all types and processes based on SimpleBoolean

Issue #2: Algebra collapse

```
type SimpleBoolean is
  sorts Bool
  opns false : → Bool
       true : → Bool
       not : Bool → Bool
  eqns
       not (false) = true;
       not (true) = false;
endtype
```

\+

```
type MyBoolean is SimpleBoolean
  opns fun : Bool → Bool
  eqns forall x, y : Bool
       fun (not (x)) = true;
       fun (x) = fun (y) => x = y;
endtype
```

- These equations imply true = false
- Σ_{bool} = {ω} where ω = true = false = fun (true) = …
- Again, MyBoolean "destroys" SimpleBoolean
  - and everything else based on SimpleBoolean

A way to avoid these issues

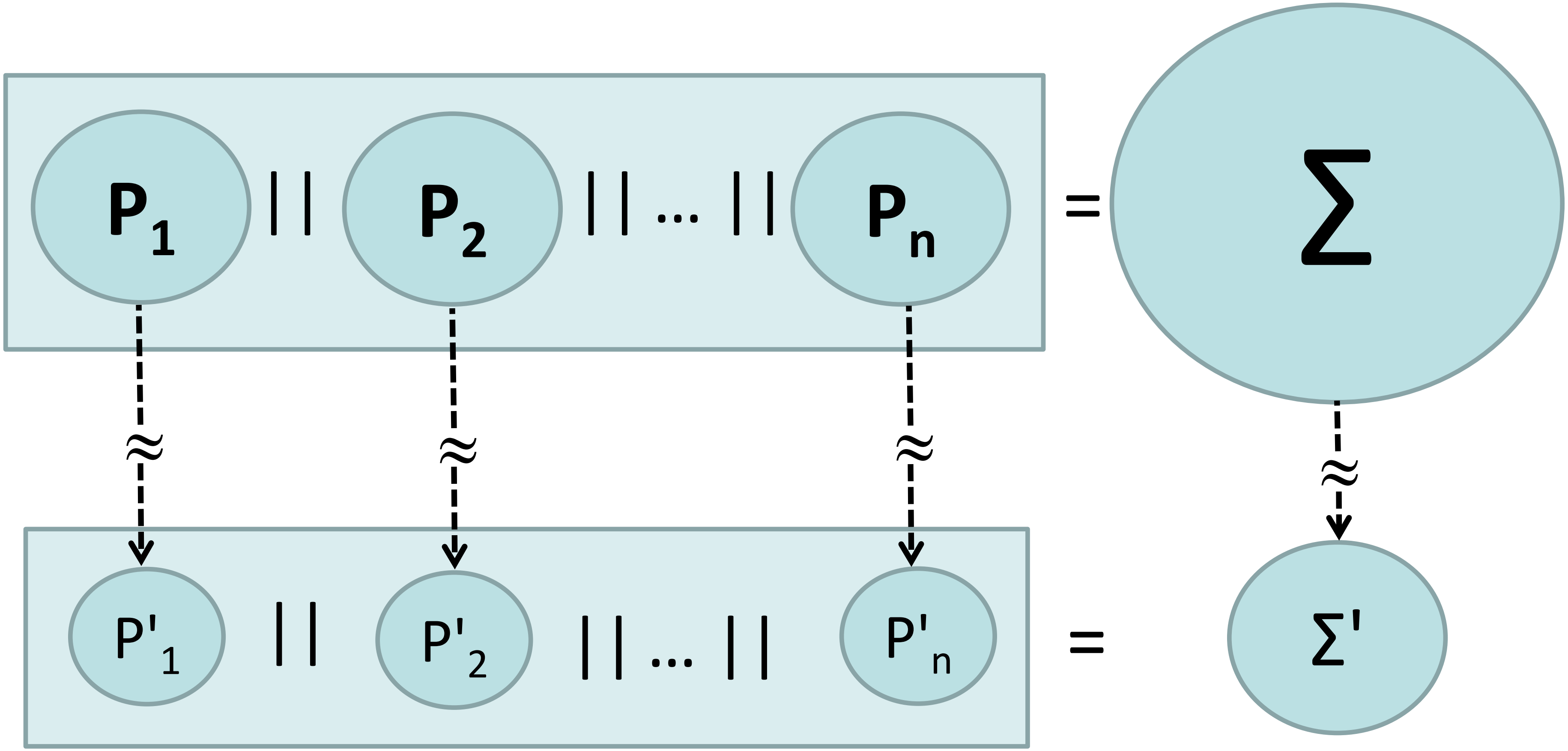
- When implementing LOTOS in CADP:
  - Replace initial algebras with term rewrite systems
  - Separate constructors from defined functions
  - No equation between constructors
  - Decreasing priorities between equations
  - Constructors for sort S defined in the same type as S
  - Equations for function F defined in the same type as F
- When defining E-LOTOS and LOTOS NT:
  - One step further: use a functional language (≈ ML without first-order, OPAL, etc.)

Impact on compositionality

- "Fully flattened" semantics is insecure:
    - Any local change may corrupt the global meaning
    - Not acceptable from an engineering point of view
    - Kind of "butterfly effect"
- Solution: "frontiers" (inside, outside, interface)
    - Defined things can be used everywhere
    - but can only be modified at controlled locations
- Many examples:
    - Encapsulation: modules, classes, objects
    - Monitors / rendezvous rather than shared variables

2.b. Compositionality issues: Concurrency I

Compositional model generation



- only valid if ≈ is a congruence wrt ||
- can/should be applied recursively

Compositional LTS generation using CADP

- Parallel components are (explicit or implicit) LTSs
- This approach is heavily implemented in CADP
  - LTSs are generated from high-level languages
  - BCG_MIN: minimization of LTSs modulo strong or branching minimization
  - REDUCTOR: on-the-fly reduction of LTSs modulo 8 equivalence relations
  - EXP.OPEN: composition of LTSs using many parallel composition operators (+ hiding, renaming, cut)

Compositional IMC generation using CADP

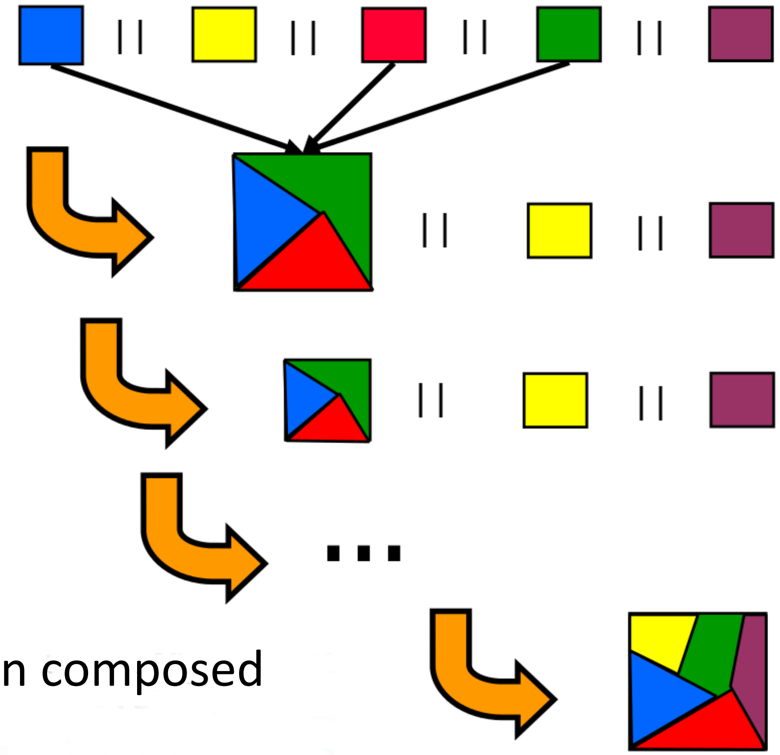
Hermanns, LNCS 2428 Garavel-Hermanns, FME 2002

- Parallel components are IMCs (Interactive Markov Chains)
  - normal transitions + stochastic ("rate") transitions
- Parallel composition is similar to interleaving
  - implemented in the EXP.OPEN tool of CADP
- Minimization combines lumpability on Markov chains with strong/branching bisimulation on LTSs
  - implemented in the BCG_MIN tool of CADP
- Additional tools: steady-state / transient solvers

Smart reduction

Crouzen-Lang, FASE 2011

- use metrics that suggest a "good" composition order
- rather than leaving the decision to the user

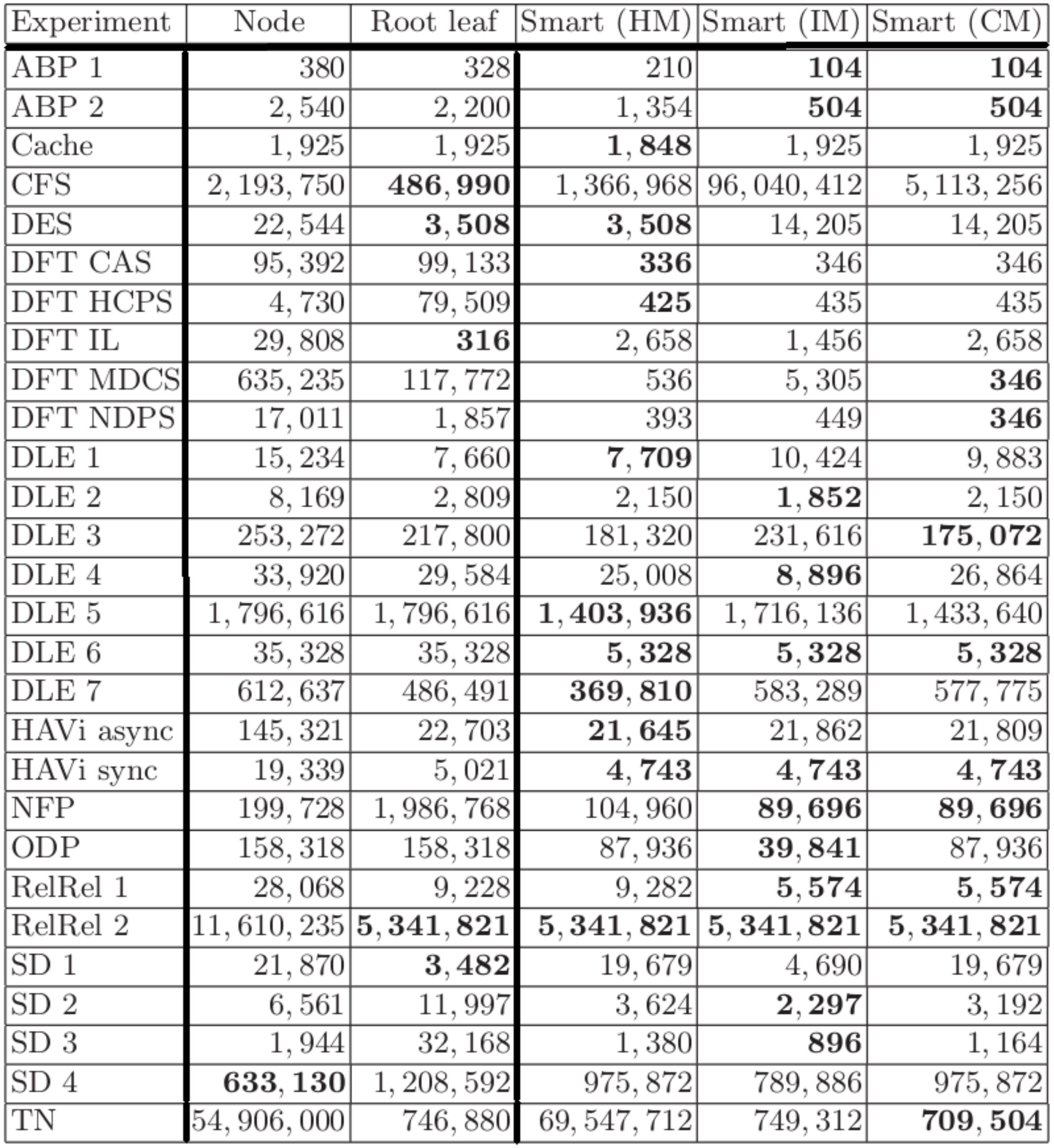


1. Select a subset of the individual processes
2. Compose this subset in parallel, hiding the internal labels
3. Minimize the resulting parallel composition modulo some equivalence (congruence)

Repeat until all individual processes have been composed

Smart reduction: Experimental results

number of transitions

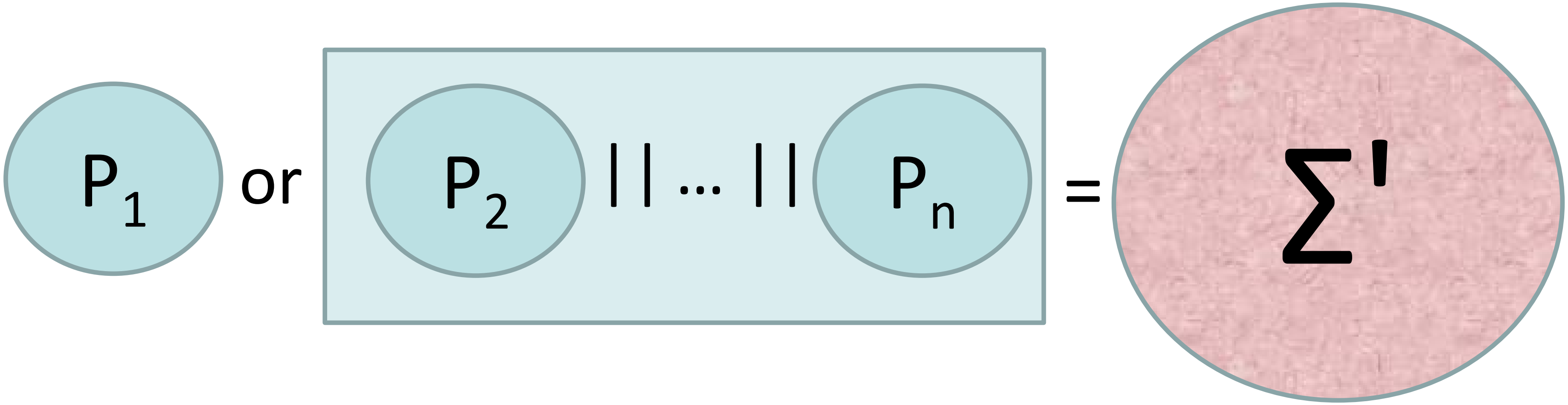


Experiment	Node	Root leaf	Smart (HM)	Smart (IM)	Smart (CM)
ABP 1	380	328	210	**104**	**104**
ABP 2	2, 540	2, 200	1, 354	**504**	**504**
Cache	1, 925	1, 925	**1, 848**	1, 925	1, 925
CFS	2, 193, 750	**486, 990**	1, 366, 968	96, 040, 412	5, 113, 256
DES	22, 544	**3, 508**	**3, 508**	14, 205	14, 205
DFT CAS	95, 392	99, 133	**336**	346	346
DFT HCPS	4, 730	79, 509	**425**	435	435
DFT IL	29, 808	**316**	2, 658	1, 456	2, 658
DFT MDCS	635, 235	117, 772	536	5, 305	**346**
DFT NDPS	17, 011	1, 857	393	449	**346**
DLE 1	15, 234	7, 660	**7, 709**	10, 424	9, 883
DLE 2	8, 169	2, 809	2, 150	**1, 852**	2, 150
DLE 3	253, 272	217, 800	181, 320	231, 616	**175, 072**
DLE 4	33, 920	29, 584	25, 008	**8, 896**	26, 864
DLE 5	1, 796, 616	1, 796, 616	**1, 403, 936**	1, 716, 136	1, 433, 640
DLE 6	35, 328	35, 328	**5, 328**	**5, 328**	**5, 328**
DLE 7	612, 637	486, 491	**369, 810**	583, 289	577, 775
HAVi async	145, 321	22, 703	**21, 645**	21, 862	21, 809
HAVi sync	19, 339	5, 021	**4, 743**	**4, 743**	**4, 743**
NFP	199, 728	1, 986, 768	104, 960	**89, 696**	**89, 696**
ODP	158, 318	158, 318	87, 936	**39, 841**	87, 936
RelRel 1	28, 068	9, 228	9, 282	**5, 574**	**5, 574**
RelRel 2	11, 610, 235	**5, 341, 821**	**5, 341, 821**	**5, 341, 821**	**5, 341, 821**
SD 1	21, 870	**3, 482**	19, 679	4, 690	19, 679
SD 2	6, 561	11, 997	3, 624	**2, 297**	3, 192
SD 3	1, 944	32, 168	1, 380	**896**	1, 164
SD 4	**633, 130**	1, 208, 592	975, 872	789, 886	975, 872
TN	54, 906, 000	746, 880	69, 547, 712	749, 312	**709, 504**

Interfaces and projections (1)

$P_1 \;||\; P_2 \;||\; \ldots \;||\; P_n = \Sigma$

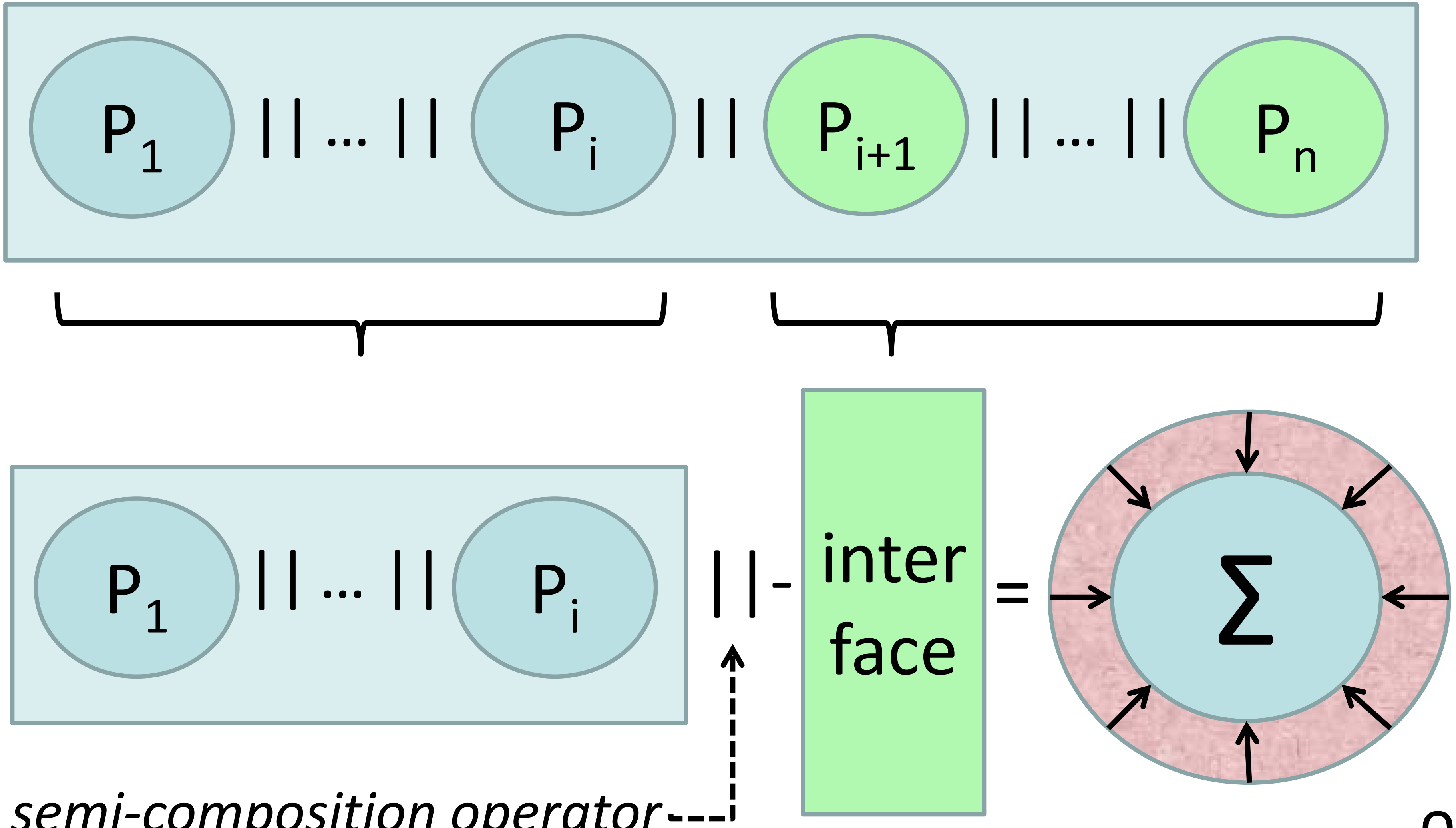
Sometimes splitting generates larger LTSs:



P_1 or $P_2 \;||\; \ldots \;||\; P_n = \Sigma'$

because splitted processes constrained each other

Interfaces and projections (2)



P_1 || … || P_i || P_{i+1} || … || P_n

P_1 || … || P_i ||- inter face = Σ

semi-composition operator

or

Interfaces and projections

Graf-Steffen, CAV 1990 Krimm-Mounier, TACAS 1997

- Interfaces L
  - Finite-state automata (trace acceptors)
  - Interfaces must be suggested by the user
  - Warning messages if interfaces are too restrictive
- Semi-composition operator P_i ||- L
  - Not a parallel composition!
  - P_i ||- L has no more states than P_i
  - Implemented by the PROJECTOR tool of CADP
- A working approach to fight state explosion

Automatic generation of interfaces

Lang, FORTE 2006

- Computed for one process P_1 wrt to P_2 … P_n
- Better reductions than using Krimm-Mounier-97
- Safety minimization and partial order reductions can be used:
- Experimental results on large processes
  - Philips' HAVi protocol: 365,923 → 645 states
  - ODP trader: 1 million states → 256 states
  - Cache coherency: 1 million states → 60 states

The SVL scripting language (1)

Garavel-Lang, FORTE 2001

- Verification scenarios are complex and repetitive
- Many tools and techniques:
  - enumerative, on-the-fly, compositional, interfaces…
  - verification **and** performance evaluation
- Many files (and formats) to handle:
  - concurrent descriptions: LOTOS, LOTOS NT, EXP, FSP…
  - explicit and implicit LTSs, CTMCs, DTMCs, IMCs…
  - interfaces, logic formulas, probability vectors…

The SVL scripting language (2)

- Many operations to perform:
  - LTS/IMC generation and projection
  - Label hiding, renaming, cut
  - Minimization and comparison modulo equivalences
  - Model checking, deadlock and livelock detection
- SVL:
  - a language to specify scenarios (+ Unix shell)
  - a compiler to execute them
  - provides a unified view of CADP tools
  - implement expert verification strategies

2.c. Compositionality issues: Concurrency II

Difference between parts I and II

- Part I
  - Only equivalences are considered
  - State space reduction must preserve an equivalence
  - Goal: generate a reduced/minimal state space
- Part II
  - A set of logical formula $\{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ is considered
  - State space reduction must preserve the truth values of these formulas
  - Goal: evaluate these formulas on a reduced/minimal state space

Decomposition wrt the formula set

$\Sigma \models \varphi_1, \varphi_2, \ldots, \varphi_n$

The unique Σ is replaced by several state spaces Σ_i

Each Σ_i is specialized/reduced wrt a given formula φ_i

$\Sigma_1 \models \varphi_1 \quad \Sigma_2 \models \varphi_2 \quad \ldots \quad \Sigma_n \models \varphi_n$

Approach 1: strong equivalence

Mateescu-Wijs, SPIN 2011

- $\varphi_1, \varphi_2, \ldots, \varphi_n$ are written in modal μ-calculus
- For each φ_i one computes a set of actions A_i such that: $\Sigma \models \varphi_i \Leftrightarrow (\text{hide } A_i \text{ in } \Sigma) \models \varphi_i$
- Basically, Ai gathers actions not occurring in φ_i
- A_i should be as large as possible (maximal hiding) to enable the greatest possible reduction
- $(\text{hide } A_i \text{ in } \Sigma)$ is reduced wrt strong bisimulation before evaluating φ_i *(global model checking)* or on-the-fly while evaluating φ_i *(local model checking)*

Approach 2: diverg. branching equiv.

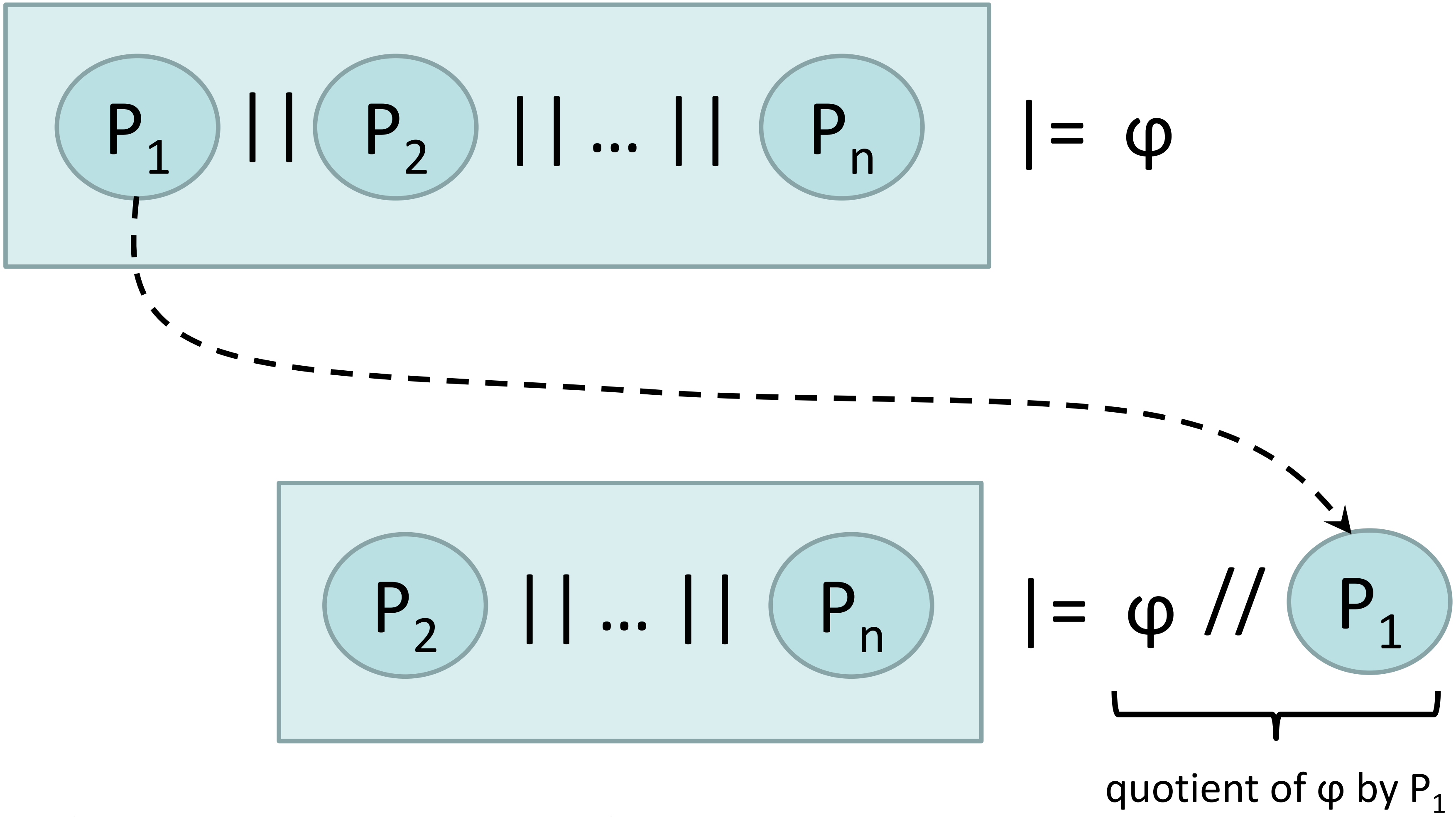
Mateescu-Wijs, SPIN 2011

- φ_1, φ_2, …, φ_n are written in a subset of the modal μ-calculus compatible with divergence-sensitive branching bisimulation
- For each φ_i one computes a set of actions A_i such that: $\Sigma \models \varphi_i \Leftrightarrow (\text{hide } A_i \text{ in } \Sigma) \models \varphi_i$
- $(\text{hide } A_i \text{ in } \Sigma)$ is reduced with divergence-sensitive branching bisimulation (enabling greater reductions than using strong bisimulation) or τ-confluence reduction (done on the fly)

Experimental results using CADP

- Using strong bisimulation
  - alternating bit (12 M states, 46 M transitions): speedup × 4, memory / 2
  - token ring (53 M states, 214 M transitions): speedup × 2.8, memory / 2.5
- Using divergence-sensitive branching bisimulation
  - Philips BRP (12 M states, 14 M transitions): memory / 1.6
- Using τ-confluence reduction
  - Erathosthene sieve: speedup × 10

Partial model checking [Andersen, LICS 95]



$$P_1 \,||\, P_2 \,||\, \dots \,||\, P_n \models \varphi$$

$$P_2 \,||\, \dots \,||\, P_n \models \underbrace{\varphi \,/\!/\, P_1}_{\text{quotient of } \varphi \text{ by } P_1}$$

(to be applied recursively)

Three issues with partial model checking

- The left-hand side should decrease a lot
    - $P_2 \; || \ldots || \; P_n$ should be much smaller than $P_1 \; || \ldots || \; P_n$
    - Not necessarily the case if P_1 constrains the others
- The right-hand side should not increase too much
    - Quotienting removes modalities, but adds variables
    - Quotiented formulas $\varphi \,//\, P_1$ can become very large
    - Simplifications must be applied after quotienting
- It requires a complex software machinery
    - Only a few implementations available

Partial model checking using CADP

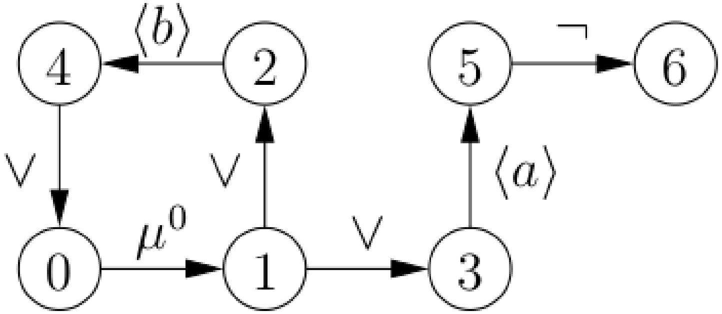
Lang-Mateescu, TACAS 2012

- Asynchronous, action-based setting
- Concurrent processes P_1 ||… || P_n :
  - Networks of LTSs (i.e., the EXP format of CADP)
  - Based on "synchronization vectors" + hiding, renaming
  - Supports the binary and n-ary parallel operators of CCS, CSP, LOTOS, LOTOS NT, etc.
- Formulas φ :
  - Alternation-free modal μ-calculus
  - + fairness operators of alternation 2

Quotienting revisited

- Formula φ is encoded as an LTS (formula graph)
  - LTSs are represented using the BCG format of CADP

$\mu X^0 . (\langle a \rangle\ \mathrm{true}) \lor (\langle b \rangle X^0)$

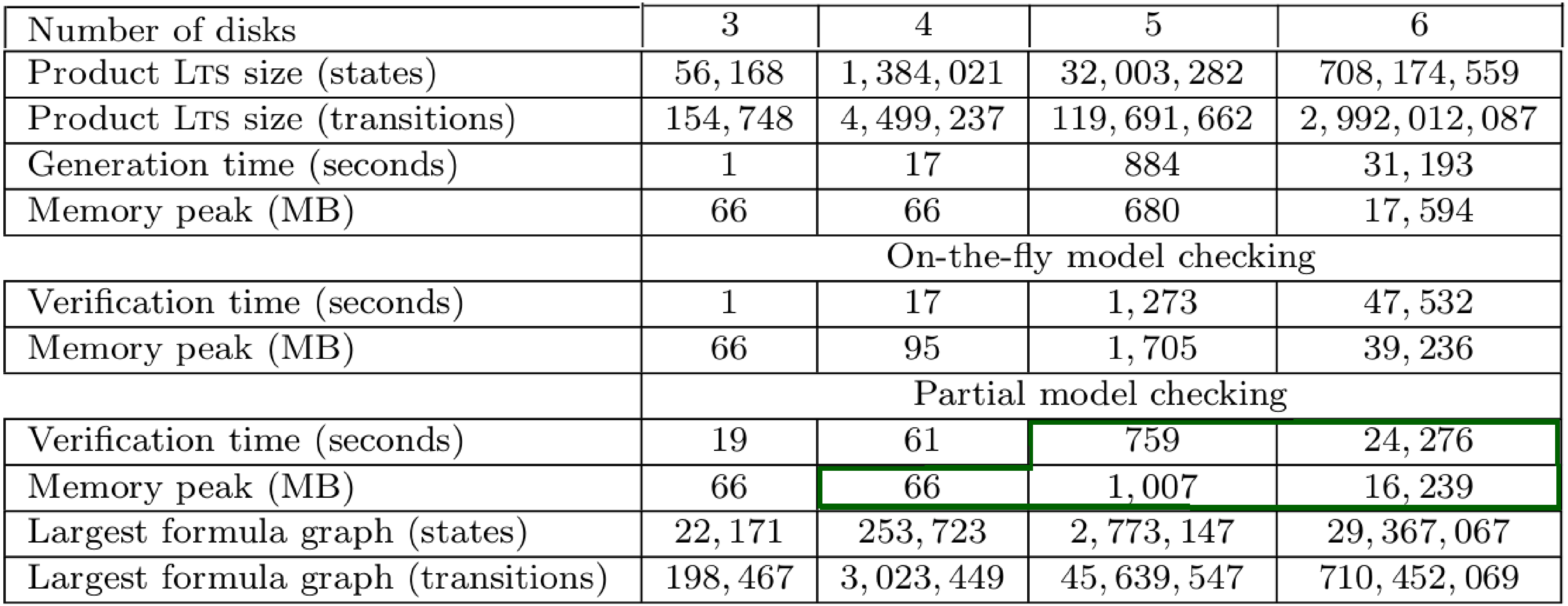


- Quotient φ // P_1 is reformulated as a synchronous product of 2 LTSs (the formula graph of φ and P_1)
  - Product can be expressed in the EXP format of CADP
  - It is computed using the EXP.OPEN tool of CADP

Post-quotienting simplifications

- Elimination of double negations
- Elimination of useless μ–transitions
  - sufficient conditions are used
- Elimination of V-transitions
  - hiding and reduction modulo τ*.a equivalence
- Sharing of identical sub-formulas
  - tagging μ-transitions → strong bisimulation reduction
- Partial evaluation of states
  - detection and propagation of constant sub-formulas
  - using the CADP solver for Boolean Equation Systems

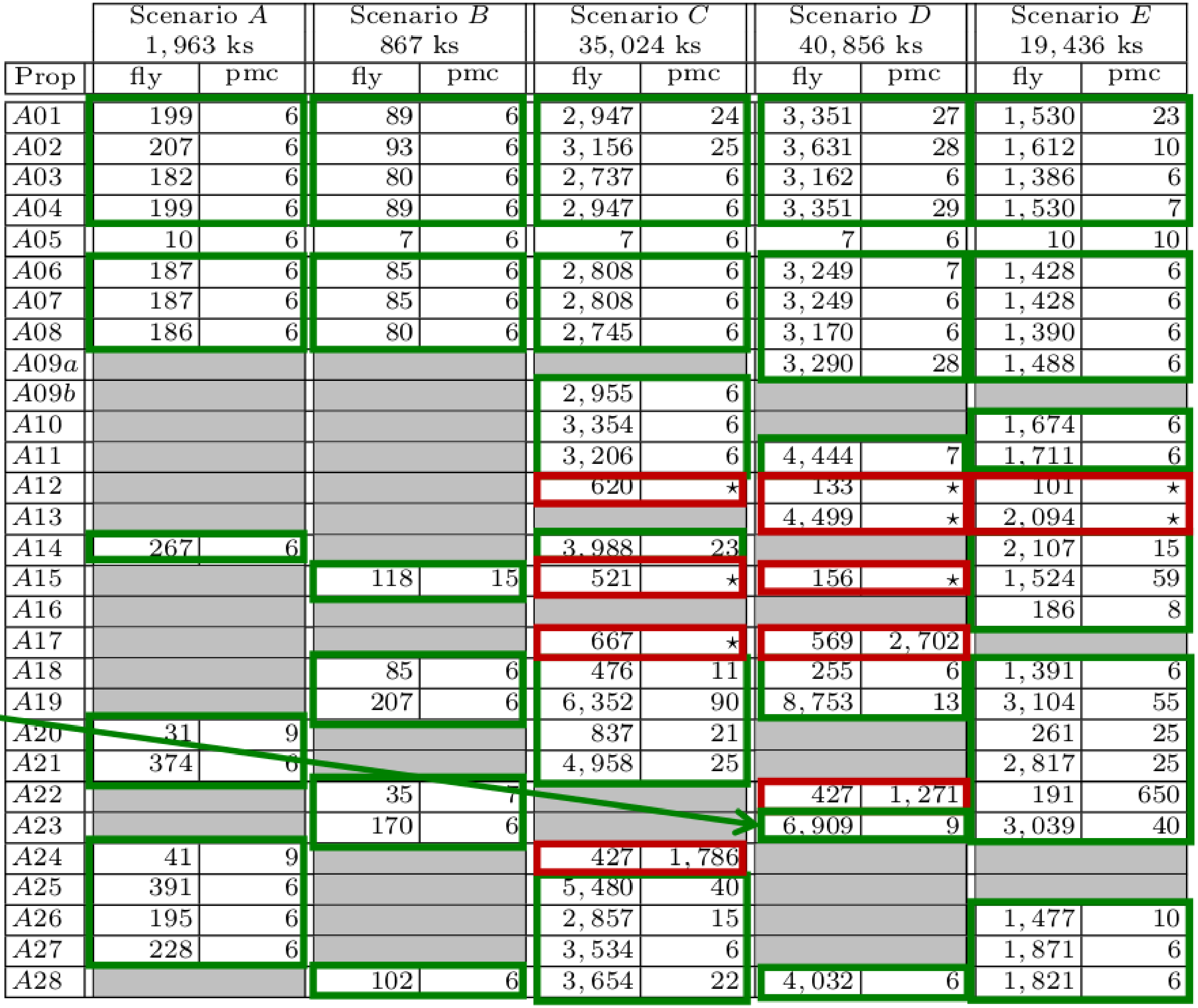
Experimental results: SCSI-2 benchmark



Number of disks	3	4	5	6
Product LTS size (states)	56, 168	1, 384, 021	32, 003, 282	708, 174, 559
Product LTS size (transitions)	154, 748	4, 499, 237	119, 691, 662	2, 992, 012, 087
Generation time (seconds)	1	17	884	31, 193
Memory peak (MB)	66	66	680	17, 594
	On-the-fly model checking			
Verification time (seconds)	1	17	1, 273	47, 532
Memory peak (MB)	66	95	1, 705	39, 236
	Partial model checking			
Verification time (seconds)	19	61	759	24, 276
Memory peak (MB)	66	66	1, 007	16, 239
Largest formula graph (states)	22, 171	253, 723	2, 773, 147	29, 367, 067
Largest formula graph (transitions)	198, 467	3, 023, 449	45, 639, 547	710, 452, 069

Experimental results: TFTP benchmark

Memory (in kbytes)



	Scenario *A* 1,963 ks		Scenario *B* 867 ks		Scenario *C* 35,024 ks		Scenario *D* 40,856 ks		Scenario *E* 19,436 ks	
Prop	fly	pmc	fly	pmc	fly	pmc	fly	pmc	fly	pmc
A01	199	6	89	6	2,947	24	3,351	27	1,530	23
A02	207	6	93	6	3,156	25	3,631	28	1,612	10
A03	182	6	80	6	2,737	6	3,162	6	1,386	6
A04	199	6	89	6	2,947	6	3,351	29	1,530	7
A05	10	6	7	6	7	6	7	6	10	10
A06	187	6	85	6	2,808	6	3,249	7	1,428	6
A07	187	6	85	6	2,808	6	3,249	6	1,428	6
A08	186	6	80	6	2,745	6	3,170	6	1,390	6
A09a							3,290	28	1,488	6
A09b					2,955	6				
A10					3,354	6			1,674	6
A11					3,206	6	4,444	7	1,711	6
A12					620	⋆	133	⋆	101	⋆
A13							4,499	⋆	2,094	⋆
A14	267	6			3,988	23			2,107	15
A15			118	15	521	⋆	156	⋆	1,524	59
A16									186	8
A17					667	⋆	569	2,702		
A18			85	6	476	11	255	6	1,391	6
A19			207	6	6,352	90	8,753	13	3,104	55
A20	31	9			837	21			261	25
A21	374	6			4,958	25			2,817	25
A22			35	7			427	1,271	191	650
A23			170	6			6,909	9	3,039	40
A24	41	9			427	1,786				
A25	391	6			5,480	40				
A26	195	6			2,857	15			1,477	10
A27	228	6			3,534	6			1,871	6
A28			102	6	3,654	22	4,032	6	1,821	6

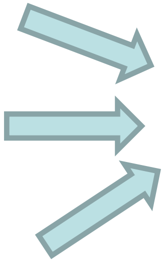
Explosion

Best ratio
= 767

3. Conclusion

Conclusion

- **Compositionality is essential**
  - modular design
  - formal verification
  - performance analysis



reusability, scalability

divide-and-conquer to fight state explosion

- **Compositionality has multiple facets**
  - data vs behaviour
  - action-based vs state-based
  - logics vs equivalences

- **Compositionality is demanding** — it requires:
  - Suitable low-level semantic **models**

    ⇒ LTSs, IMCs, etc.
  - Well-chosen behavioural **equivalences**

    ⇒ bisimulations: strong, branching, divergence-preserving, lumpability on Markov chains
  - Well-chosen **logics**

    ⇒ mu-calculus, temporal logics

    ⇒ adequation results relating logics and equivalences
  - Concurrent **languages** with a proper semantics

    ⇒ process calculi and their modern variants (such as LNT)

    ⇒ congruence results relating parallel composition and equivalences

Compositionality and CADP

- CADP:
  - A modular toolbox implementing concurrency theory
  - Used for teaching, research, and industrial problems
  - Free for academics
- Compositionality underlies CADP architecture:
  - Many compositional approaches implemented
  - Combinations of existing and new CADP components
  - Mostly in an action-based setting
- Our wish: Compositionality made easy using CADP