

# From decision procedures to full model-checking: the MCMT experience

S. Ghilardi

University of Milan, Italy

Dagstuhl Workshop, November 3, 2015

# Aim of the talk

Since about 2010, I am developing and maintaining a logic-based models checker (called **MCMT**, 'Model Checker Modulo Theories'). In this talk, I shall briefly report my experience and some case studies.

During past years, many people contributed to implementation, theoretical advances as well as experiments; among them, let me mention S. Ranise, A. Carioni, R. Bruttomesso, F. Alberti, E. Pagani, A. Orsini.

# Aim of the talk

Since about 2010, I am developing and maintaining a logic-based models checker (called **MCMT**, 'Model Checker Modulo Theories'). In this talk, I shall briefly report my experience and some case studies.

During past years, many people contributed to implementation, theoretical advances as well as experiments; among them, let me mention S. Ranise, A. Carioni, R. Bruttomesso, F. Alberti, E. Pagani, A. Orsini.

## Aim of the talk

Since about 2010, I am developing and maintaining a logic-based models checker (called **MCMT**, 'Model Checker Modulo Theories'). In this talk, I shall briefly report my experience and some case studies.

During past years, many people contributed to implementation, theoretical advances as well as experiments; among them, let me mention S. Ranise, A. Carioni, R. Bruttomesso, F. Alberti, E. Pagani, A. Orsini.

# Aim of the talk

## Main features of MCMT:

- declarative approach;
- use of decision procedures for combined theories;
- quantifier handling through instantiation;
- quantifier handling through quantifier elimination;
- large expressivity;
- flexibility and possibility of integrating old and new techniques (acceleration, abstraction, invariant synthesis,...);
- large applications spectrum (distributed, timed, fault tolerant, but also sequential systems).

# Aim of the talk

## Main features of MCMT:

- declarative approach;
- use of decision procedures for combined theories;
- quantifier handling through instantiation;
- quantifier handling through quantifier elimination;
- large expressivity;
- flexibility and possibility of integrating old and new techniques (acceleration, abstraction, invariant synthesis,...);
- large applications spectrum (distributed, timed, fault tolerant, but also sequential systems).

# Aim of the talk

## Main features of MCMT:

- declarative approach;
- use of decision procedures for combined theories;
- quantifier handling through instantiation;
- quantifier handling through quantifier elimination;
- large expressivity;
- flexibility and possibility of integrating old and new techniques (acceleration, abstraction, invariant synthesis,...);
- large applications spectrum (distributed, timed, fault tolerant, but also sequential systems).

# Aim of the talk

## Main features of MCMT:

- declarative approach;
- use of decision procedures for combined theories;
- quantifier handling through instantiation;
- quantifier handling through quantifier elimination;
- large expressivity;
- flexibility and possibility of integrating old and new techniques (acceleration, abstraction, invariant synthesis,...);
- large applications spectrum (distributed, timed, fault tolerant, but also sequential systems).



# Aim of the talk

## Main features of MCMT:

- declarative approach;
- use of decision procedures for combined theories;
- quantifier handling through instantiation;
- quantifier handling through quantifier elimination;
- large expressivity;
- flexibility and possibility of integrating old and new techniques (acceleration, abstraction, invariant synthesis,...);
- large applications spectrum (distributed, timed, fault tolerant, but also sequential systems).

# Aim of the talk

## Main features of MCMT:

- declarative approach;
- use of decision procedures for combined theories;
- quantifier handling through instantiation;
- quantifier handling through quantifier elimination;
- large expressivity;
- flexibility and possibility of integrating old and new techniques (acceleration, abstraction, invariant synthesis,...);
- large applications spectrum (distributed, timed, fault tolerant, but also sequential systems).

# Aim of the talk

## Main features of MCMT:

- declarative approach;
- use of decision procedures for combined theories;
- quantifier handling through instantiation;
- quantifier handling through quantifier elimination;
- large expressivity;
- flexibility and possibility of integrating old and new techniques (acceleration, abstraction, invariant synthesis,...);
- large applications spectrum (distributed, timed, fault tolerant, but also sequential systems).

# Aim of the talk

## Main features of MCMT:

- declarative approach;
- use of decision procedures for combined theories;
- quantifier handling through instantiation;
- quantifier handling through quantifier elimination;
- large expressivity;
- flexibility and possibility of integrating old and new techniques (acceleration, abstraction, invariant synthesis,...);
- large applications spectrum (distributed, timed, fault tolerant, but also sequential systems).

# Aim of the talk

From a logical point of view:

- MCMT handles quantified formulae to represent set of states (this is a distinguishing feature of the tool);
- prominent role is played by array fragments;
- a parallel investigation on decidability/complexity of  $\exists^*\forall^*$ -fragments of array theories have been carried out;
- for a survey on this (covering various fragments: Bradley-Manna, flat, SIL, acceleratable, etc.) see my slides at Dagstuhl seminar 15381.

# Aim of the talk

From a logical point of view:

- MCMT handles quantified formulae to represent set of states (this is a distinguishing feature of the tool);
- prominent role is played by array fragments;
- a parallel investigation on decidability/complexity of  $\exists^*\forall^*$ -fragments of array theories have been carried out;
- for a survey on this (covering various fragments: Bradley-Manna, flat, SIL, acceleratable, etc.) see my slides at Dagstuhl seminar 15381.

# Aim of the talk

From a logical point of view:

- MCMT handles quantified formulae to represent set of states (this is a distinguishing feature of the tool);
- prominent role is played by array fragments;
- a parallel investigation on decidability/complexity of  $\exists^*\forall^*$ -fragments of array theories have been carried out;
- for a survey on this (covering various fragments: Bradley-Manna, flat, SIL, acceleratable, etc.) see my slides at Dagstuhl seminar 15381.

# Aim of the talk

From a logical point of view:

- MCMT handles quantified formulae to represent set of states (this is a distinguishing feature of the tool);
- prominent role is played by array fragments;
- a parallel investigation on decidability/complexity of  $\exists^*\forall^*$ -fragments of array theories have been carried out;
- for a survey on this (covering various fragments: Bradley-Manna, flat, SIL, acceleratable, etc.) see my slides at Dagstuhl seminar 15381.



# Aim of the talk

From a logical point of view:

- MCMT handles quantified formulae to represent set of states (this is a distinguishing feature of the tool);
- prominent role is played by array fragments;
- a parallel investigation on decidability/complexity of  $\exists^*\forall^*$ -fragments of array theories have been carried out;
- for a survey on this (covering various fragments: Bradley-Manna, flat, SIL, acceleratable, etc.) see my slides at Dagstuhl seminar 15381.

# Outline

- 1 The core: a brief review on WSTS
- 2 The Declarative Perspective
- 3 The tool MCMT
- 4 Software Model Checking Applications

# Outline

- 1 The core: a brief review on WSTS
- 2 The Declarative Perspective
- 3 The tool MCMT
- 4 Software Model Checking Applications

# Outline

- 1 The core: a brief review on WSTS
- 2 The Declarative Perspective
- 3 The tool MCMT
- 4 Software Model Checking Applications

# Outline

- 1 The core: a brief review on WSTS
- 2 The Declarative Perspective
- 3 The tool MCMT
- 4 Software Model Checking Applications

- 1 The core: a brief review on WSTS
- 2 The Declarative Perspective
- 3 The tool MCMT
- 4 Software Model Checking Applications

# Verification of Parameterised Systems

- **Parameterised system** = bunch of concurrent processes (**topology** may vary, can be e.g., set-like, linear-like, tree-like, ring-like, ...)
- **Process** = instance of the same state-machine
- **Configuration** = state of a parameterised system
- **Transition** = either a process changing its locations/data or several processes simultaneously changing their respective locations/data (e.g., broadcast) [**interleaving semantics**]
- **CHALLENGE**: automatically verify a property regardless of the number of processes
- A state machine has **finitely many control locations** and can manipulate **finitely many variables over possibly unbounded domains**

# Verification of Parameterised Systems

- **Parameterised system** = bunch of concurrent processes (**topology** may vary, can be e.g., set-like, linear-like, tree-like, ring-like, ...)
- **Process** = instance of the same state-machine
- **Configuration** = state of a parameterised system
- **Transition** = either a process changing its locations/data or several processes simultaneously changing their respective locations/data (e.g., broadcast) [**interleaving semantics**]
- **CHALLENGE**: automatically verify a property regardless of the number of processes
- A state machine has **finitely many control locations** and can manipulate **finitely many variables over possibly unbounded domains**



# Verification of Parameterised Systems

- **Parameterised system** = bunch of concurrent processes (**topology** may vary, can be e.g., set-like, linear-like, tree-like, ring-like, ...)
- **Process** = instance of the same state-machine
- **Configuration** = state of a parameterised system
- **Transition** = either a process changing its locations/data or several processes simultaneously changing their respective locations/data (e.g., broadcast) [**interleaving semantics**]
- **CHALLENGE**: automatically verify a property regardless of the number of processes
- A state machine has **finitely many control locations** and can manipulate **finitely many variables over possibly unbounded domains**

# Verification of Parameterised Systems

- **Parameterised system** = bunch of concurrent processes (**topology** may vary, can be e.g., set-like, linear-like, tree-like, ring-like, ...)
- **Process** = instance of the same state-machine
- **Configuration** = state of a parameterised system
- **Transition** = either a process changing its locations/data or several processes simultaneously changing their respective locations/data (e.g., broadcast) [**interleaving semantics**]
- **CHALLENGE**: automatically verify a property regardless of the number of processes
- A state machine has **finitely many control locations** and can manipulate **finitely many variables over possibly unbounded domains**

# Verification of Parameterised Systems

- **Parameterised system** = bunch of concurrent processes (**topology** may vary, can be e.g., set-like, linear-like, tree-like, ring-like, ...)
- **Process** = instance of the same state-machine
- **Configuration** = state of a parameterised system
- **Transition** = either a process changing its locations/data or several processes simultaneously changing their respective locations/data (e.g., broadcast) [**interleaving semantics**]
- **CHALLENGE**: automatically verify a property regardless of the number of processes
- A state machine has **finitely many control locations** and can manipulate **finitely many variables over possibly unbounded domains**

# Verification of Parameterised Systems

- **Parameterised system** = bunch of concurrent processes (**topology** may vary, can be e.g., set-like, linear-like, tree-like, ring-like, ...)
- **Process** = instance of the same state-machine
- **Configuration** = state of a parameterised system
- **Transition** = either a process changing its locations/data or several processes simultaneously changing their respective locations/data (e.g., broadcast) [**interleaving semantics**]
- **CHALLENGE**: automatically verify a property regardless of the number of processes
- A state machine has **finitely many control locations** and can manipulate **finitely many variables over possibly unbounded domains**

# Well-Structured Transition Systems

Seminal paper [ACJT - LICS96]

$$(S, \tau, \preceq)$$

- $S$ : set of states;
- $\tau = \{\rightarrow_{\lambda} \subseteq S \times S\}_{\lambda}$ : labelled directed graph;
- $\preceq$ : **well quasi ordering**<sup>1</sup> (wqo) on  $S$ ;
- each  $\tau_{\lambda}$  is **monotonic**:

$$\begin{array}{ccc}
 s_1 & \preceq & s_2 \\
 \downarrow_{\lambda} & & \downarrow_{\lambda} \\
 s_3 & \preceq \exists & s_4
 \end{array}$$

<sup>1</sup>Reflexive, transitive binary relation that neither contains infinite strictly decreasing sequences nor infinite sequences of pairwise incomparable elements

# Well-Structured Transition Systems

Seminal paper [ACJT - LICS96]

$$(S, \tau, \preceq)$$

- $S$ : set of states;
- $\tau = \{\rightarrow_\lambda \subseteq S \times S\}_\lambda$ : labelled directed graph;
- $\preceq$ : **well quasi ordering**<sup>1</sup> (wqo) on  $S$ ;
- each  $\tau_\lambda$  is **monotonic**:

$$\begin{array}{ccc}
 s_1 & \preceq & s_2 \\
 \downarrow_\lambda & & \downarrow_\lambda \\
 s_3 & \preceq \exists & s_4
 \end{array}$$

<sup>1</sup>Reflexive, transitive binary relation that neither contains infinite strictly decreasing sequences nor infinite sequences of pairwise incomparable elements

# Well-Structured Transition Systems

Seminal paper [ACJT - LICS96]

$$(S, \tau, \preceq)$$

- $S$ : set of states;
- $\tau = \{\rightarrow_{\lambda} \subseteq S \times S\}_{\lambda}$ : labelled directed graph;
- $\preceq$ : **well quasi ordering**<sup>1</sup> (wqo) on  $S$ ;
- each  $\tau_{\lambda}$  is **monotonic**:

$$\begin{array}{ccc}
 s_1 & \preceq & s_2 \\
 \downarrow_{\lambda} & & \downarrow_{\lambda} \\
 s_3 & \preceq \exists & s_4
 \end{array}$$

<sup>1</sup>Reflexive, transitive binary relation that neither contains infinite strictly decreasing sequences nor infinite sequences of pairwise incomparable elements

# Well-Structured Transition Systems

Seminal paper [ACJT - LICS96]

$$(S, \tau, \preceq)$$

- $S$ : set of states;
- $\tau = \{\rightarrow_{\lambda} \subseteq S \times S\}_{\lambda}$ : labelled directed graph;
- $\preceq$ : **well quasi ordering**<sup>1</sup> (wqo) on  $S$ ;
- each  $\tau_{\lambda}$  is **monotonic**:

$$\begin{array}{ccc}
 s_1 & \preceq & s_2 \\
 \downarrow_{\lambda} & & \downarrow_{\lambda} \\
 s_3 & \preceq \exists & s_4
 \end{array}$$

<sup>1</sup>Reflexive, transitive binary relation that neither contains infinite strictly decreasing sequences nor infinite sequences of pairwise incomparable elements



# Well-Structured Transition Systems

Seminal paper [ACJT - LICS96]

$$(S, \tau, \preceq)$$

- $S$ : set of states;
- $\tau = \{\rightarrow_{\lambda} \subseteq S \times S\}_{\lambda}$ : labelled directed graph;
- $\preceq$ : **well quasi ordering**<sup>1</sup> (wqo) on  $S$ ;
- each  $\tau_{\lambda}$  is **monotonic**:

$$\begin{array}{ccc}
 s_1 & \preceq & s_2 \\
 \downarrow_{\lambda} & & \downarrow_{\lambda} \\
 s_3 & \preceq \boxplus & s_4
 \end{array}$$

<sup>1</sup>Reflexive, transitive binary relation that neither contains infinite strictly decreasing sequences nor infinite sequences of pairwise incomparable elements

# Well-Structured Transition Systems

Seminal paper [ACJT - LICS96]

$$(S, \tau, \preceq)$$

- $S$ : set of states;
- $\tau = \{\rightarrow_{\lambda} \subseteq S \times S\}_{\lambda}$ : labelled directed graph;
- $\preceq$ : **well quasi ordering**<sup>1</sup> (wqo) on  $S$ ;
- each  $\tau_{\lambda}$  is **monotonic**:

$$\begin{array}{ccc}
 s_1 & \preceq & s_2 \\
 \downarrow_{\lambda} & & \downarrow_{\lambda} \\
 s_3 & \preceq \exists & s_4
 \end{array}$$

<sup>1</sup>Reflexive, transitive binary relation that neither contains infinite strictly decreasing sequences nor infinite sequences of pairwise incomparable elements

# Well-Structured Transition Systems

- Set of **unsafe** states represented by an **upset**  $K$ :

$$s \in K \wedge s \preceq s' \rightarrow s' \in K$$

- Monotonicity implies that the **pre-image of an upset is still an upset**

$$Pre(\tau, K) := \{s \mid \exists \lambda \exists s' (s \xrightarrow{\lambda} s') \wedge s' \in K\}$$

- Since  $\preceq$  is a wqo, **upsets can be finitely represented by their *finitely many* minimal elements**
- Since  $\preceq$  is a wqo, a backward search algorithm terminates.
- Extensions to cases in which  $\preceq$  is not a wqo often terminate ‘in practice’.

# Monotonic Abstraction

But ... what to do if a transition  $\tau_\lambda$  is not monotonic?

We may have  $s \xrightarrow{\tau_\lambda} s'$  but  $\tilde{s} \xrightarrow{\tau_\lambda} s'$  for some  $\tilde{s} \preceq s$ .

In this case, *monotonic abstraction* allows  $\tau_\lambda$  to fire: roughly, the system may change its status from  $s$  to  $\tilde{s}$  to allow this.

Monotonic abstraction may introduce spurious runs (intuitively: runs in which some processes ‘crash and disappear’), but **if a safety certification is obtained for the abstract system, the certification holds for the original system too.**

Lot of success for the verification of safety properties of a variety of systems: broadcast protocols, cache coherence protocols, lossy channels systems, parameterized timed automata, etc.

# Monotonic Abstraction

But ... what to do if a transition  $\tau_\lambda$  is not monotonic?

We may have  $s \xrightarrow{\tau_\lambda} s'$  but  $\tilde{s} \xrightarrow{\tau_\lambda} s'$  for some  $\tilde{s} \preceq s$ .

In this case, *monotonic abstraction* allows  $\tau_\lambda$  to fire: roughly, the system may change its status from  $s$  to  $\tilde{s}$  to allow this.

Monotonic abstraction may introduce spurious runs (intuitively: runs in which some processes ‘crash and disappear’), but **if a safety certification is obtained for the abstract system, the certification holds for the original system too.**

Lot of success for the verification of safety properties of a variety of systems: broadcast protocols, cache coherence protocols, lossy channels systems, parameterized timed automata, etc.

# Monotonic Abstraction

But ... what to do if a transition  $\tau_\lambda$  is not monotonic?

We may have  $s \xrightarrow{\tau_\lambda} s'$  but  $\tilde{s} \xrightarrow{\tau_\lambda} s'$  for some  $\tilde{s} \preceq s$ .

In this case, *monotonic abstraction* allows  $\tau_\lambda$  to fire: roughly, the system may change its status from  $s$  to  $\tilde{s}$  to allow this.

Monotonic abstraction may introduce spurious runs (intuitively: runs in which some processes ‘crash and disappear’), but *if a safety certification is obtained for the abstract system, the certification holds for the original system too.*

Lot of success for the verification of safety properties of a variety of systems: broadcast protocols, cache coherence protocols, lossy channels systems, parameterized timed automata, etc.

# Monotonic Abstraction

But ... what to do if a transition  $\tau_\lambda$  is not monotonic?

We may have  $s \xrightarrow{\tau_\lambda} s'$  but  $\tilde{s} \xrightarrow{\tau_\lambda} s'$  for some  $\tilde{s} \preceq s$ .

In this case, *monotonic abstraction* allows  $\tau_\lambda$  to fire: roughly, the system may change its status from  $s$  to  $\tilde{s}$  to allow this.

Monotonic abstraction may introduce spurious runs (intuitively: runs in which some processes ‘crash and disappear’), but **if a safety certification is obtained for the abstract system, the certification holds for the original system too.**

Lot of success for the verification of safety properties of a variety of systems: broadcast protocols, cache coherence protocols, lossy channels systems, parameterized timed automata, etc.

# Monotonic Abstraction

But ... what to do if a transition  $\tau_\lambda$  is not monotonic?

We may have  $s \xrightarrow{\tau_\lambda} s'$  but  $\tilde{s} \xrightarrow{\tau_\lambda} s'$  for some  $\tilde{s} \preceq s$ .

In this case, *monotonic abstraction* allows  $\tau_\lambda$  to fire: roughly, the system may change its status from  $s$  to  $\tilde{s}$  to allow this.

Monotonic abstraction may introduce spurious runs (intuitively: runs in which some processes ‘crash and disappear’), but **if a safety certification is obtained for the abstract system, the certification holds for the original system too.**

Lot of success for the verification of safety properties of a variety of systems: broadcast protocols, cache coherence protocols, lossy channels systems, parameterized timed automata, etc.



- 1 The core: a brief review on WSTS
- 2 The Declarative Perspective**
- 3 The tool MCMT
- 4 Software Model Checking Applications

# Array-based Systems

**GOAL:** to get a **declarative** formulation of all this and to obtain an **efficient backward reachability analysis** by using state-of-the-art **SMT solving** for both safety and fix-point checking.

By a *theory* we mean here a pair  $T = (\Sigma, \mathcal{C})$ , where  $\Sigma$  is a first-order signature and  $\mathcal{C}$  is a class of  $\Sigma$ -structures (called the models of  $T$ ). Satisfiability of at least quantifier-free formulae in  $\mathcal{C}$  should be decidable.

We need a theory  $T_I$  for describing processes and a theory  $T_E$  for data. We combine these two theories in a 3-sorted theory  $A_I^E$ .

# Array-based Systems

**GOAL:** to get a **declarative** formulation of all this and to obtain an **efficient backward reachability analysis** by using state-of-the-art **SMT solving** for both safety and fix-point checking.

By a *theory* we mean here a pair  $T = (\Sigma, \mathcal{C})$ , where  $\Sigma$  is a first-order signature and  $\mathcal{C}$  is a class of  $\Sigma$ -structures (called the models of  $T$ ). Satisfiability of at least quantifier-free formulae in  $\mathcal{C}$  should be decidable.

We need a theory  $T_I$  for describing processes and a theory  $T_E$  for data. We combine these two theories in a 3-sorted theory  $A_I^E$ .

# Array-based Systems

**GOAL:** to get a **declarative** formulation of all this and to obtain an **efficient backward reachability analysis** by using state-of-the-art **SMT solving** for both safety and fix-point checking.

By a *theory* we mean here a pair  $T = (\Sigma, \mathcal{C})$ , where  $\Sigma$  is a first-order signature and  $\mathcal{C}$  is a class of  $\Sigma$ -structures (called the models of  $T$ ). Satisfiability of at least quantifier-free formulae in  $\mathcal{C}$  should be decidable.

We need a theory  $T_I$  for describing processes and a theory  $T_E$  for data. We combine these two theories in a 3-sorted theory  $A_I^E$ .

# Array-based Systems

**GOAL:** to get a **declarative** formulation of all this and to obtain an **efficient backward reachability analysis** by using state-of-the-art **SMT solving** for both safety and fix-point checking.

By a *theory* we mean here a pair  $T = (\Sigma, \mathcal{C})$ , where  $\Sigma$  is a first-order signature and  $\mathcal{C}$  is a class of  $\Sigma$ -structures (called the models of  $T$ ). Satisfiability of at least quantifier-free formulae in  $\mathcal{C}$  should be decidable.

We need a theory  $T_I$  for describing processes and a theory  $T_E$  for data. We combine these two theories in a 3-sorted theory  $A_I^E$ .

# Array-Based Systems

- the sort `INDEX` is constrained by  $T_I$ ;
- the sort `ELEM` is constrained by  $T_E$ ;
- the sort `ARRAY` represents arrays of `ELEM` defined on `INDEX`;
- the ‘read’ operation `_[_]` is added to  $\Sigma_I \cup \Sigma_E$ ;
- the class of models of  $A_I^E$  consists of the three-sorted structures whose reducts are models of  $T_I$ ,  $T_E$  and the sort `ARRAY` is interpreted as the set of total functions from indexes to elements and the read operation is interpreted as function application

# Array-Based Systems

- An **array-based system** on  $A_I^E$  with array state variable  $a$  is the following pair of formulae:

$$\mathcal{S} = \langle I(\mathbf{a}), \tau(\mathbf{a}, \mathbf{a}') \rangle.$$

- A **state** of an array-based system is an assignment to the variable  $a$  in a model of  $A_I^E$
- A **safety problem** for  $\mathcal{S}$  is the following: given a formula  $K(\mathbf{a})$ , is

$$I(\mathbf{a}_0) \wedge \tau(\mathbf{a}_0, \mathbf{a}_1) \wedge \cdots \wedge \tau(\mathbf{a}_{n-1}, \mathbf{a}_n) \wedge K(\mathbf{a}_n)$$

$A_I^E$ -satisfiable for some  $n$ ?

# Array-Based Systems

- An **array-based system** on  $A_I^E$  with array state variable  $a$  is the following pair of formulae:

$$\mathcal{S} = \langle I(a), \tau(a, a') \rangle.$$

- A **state** of an array-based system is an assignment to the variable  $a$  in a model of  $A_I^E$
- A **safety problem** for  $\mathcal{S}$  is the following: given a formula  $K(a)$ , is

$$I(a_0) \wedge \tau(a_0, a_1) \wedge \cdots \wedge \tau(a_{n-1}, a_n) \wedge K(a_n)$$

$A_I^E$ -satisfiable for some  $n$ ?



# Array-Based Systems

- An **array-based system** on  $A_I^E$  with array state variable  $a$  is the following pair of formulae:

$$\mathcal{S} = \langle I(a), \tau(a, a') \rangle.$$

- A **state** of an array-based system is an assignment to the variable  $a$  in a model of  $A_I^E$
- A **safety problem** for  $\mathcal{S}$  is the following: given a formula  $K(a)$ , is

$$I(a_0) \wedge \tau(a_0, a_1) \wedge \cdots \wedge \tau(a_{n-1}, a_n) \wedge K(a_n)$$

$A_I^E$ -satisfiable for some  $n$ ?

## Revisiting Backward Reachability

Idea: recast symbolically the backward reachability algorithm

```

function BReach( $K$ )
   $i \leftarrow 0$ ;  $BR^0(\tau, K) \leftarrow K$ ;  $K^0 \leftarrow K$ 
  if  $A_J^E$ -check( $BR^0(\tau, K) \wedge I$ ) = sat then return unsafe
  repeat
     $K^{i+1} \leftarrow \text{Pre}(\tau, K^i)$ 
     $BR^{i+1}(\tau, K) \leftarrow BR^i(\tau, K) \vee K^{i+1}$ 
    if  $A_J^E$ -check( $BR^{i+1}(\tau, K) \wedge I$ ) = sat then return unsafe
    else  $i \leftarrow i + 1$ 
  until  $A_J^E$ -check( $\neg(BR^{i+1}(\tau, K) \rightarrow BR^i(\tau, K))$ ) = unsat
  return safe
end

```

But this is problematic... unless right formats for  $I, \tau, K$  are found!

# Format for initialization formulae

## Format for $\forall^I$ -formulae

$$\forall \underline{i} \phi(\underline{i}, a[\underline{i}])$$

where  $\underline{i}$  is a tuple of variables of sort `INDEX` and  $\phi$  is a quantifier-free  $\Sigma_I \cup \Sigma_E$ -formula<sup>2</sup>

For instance, the formula  $\forall i. a[i] = \text{idle}$  says that all processes are in state `idle`.

$\forall^I$ -formulae can also be used to express invariants

<sup>2</sup>If  $\underline{i} = i_1, \dots, i_n$ , then  $a[\underline{i}]$  is the tuple of terms  $a[i_1], \dots, a[i_n]$  having sort `ELEM`

# Format for initialization formulae

## Format for $I$ : $\forall^I$ -formulae

$$\forall \underline{i} \phi(\underline{i}, \mathbf{a}[\underline{i}])$$

where  $\underline{i}$  is a tuple of variables of sort `INDEX` and  $\phi$  is a quantifier-free  $\Sigma_I \cup \Sigma_E$ -formula<sup>2</sup>

For instance, the formula  $\forall i. \mathbf{a}[i] = \text{idle}$  says that all processes are in state `idle`.

$\forall^I$ -formulae can also be used to express invariants

<sup>2</sup>If  $\underline{i} = i_1, \dots, i_n$ , then  $\mathbf{a}[\underline{i}]$  is the tuple of terms  $\mathbf{a}[i_1], \dots, \mathbf{a}[i_n]$  having sort `ELEM`

# Format for initialization formulae

## Format for $\forall^I$ -formulae

$$\forall \underline{i} \phi(\underline{i}, \mathbf{a}[\underline{i}])$$

where  $\underline{i}$  is a tuple of variables of sort `INDEX` and  $\phi$  is a quantifier-free  $\Sigma_I \cup \Sigma_E$ -formula<sup>2</sup>

For instance, the formula  $\forall i. \mathbf{a}[i] = \text{idle}$  says that all processes are in state `idle`.

$\forall^I$ -formulae can also be used to express invariants

<sup>2</sup>If  $\underline{i} = i_1, \dots, i_n$ , then  $\mathbf{a}[\underline{i}]$  is the tuple of terms  $\mathbf{a}[i_1], \dots, \mathbf{a}[i_n]$  having sort `ELEM`

# Format for unsafety problems formulae

## Proposed format for $K$ : $\exists^!$ -formulae

$$\exists \underline{i} \phi(\underline{i}, a[\underline{i}])$$

where  $\underline{i}$  is a tuple of variables of sort `INDEX` and  $\phi$  is a quantifier-free  $\Sigma_I \cup \Sigma_E$ -formula.

For instance, the formula

$$\exists i_1 \exists i_2. (i_1 \neq i_2 \wedge a[i_1] = \text{use} \wedge a[i_2] = \text{use})$$

expresses that mutual exclusion is violated.

# Format for unsafety problems formulae

**Proposed format for  $K$ :  $\exists^!$ -formulae**

$$\exists \underline{i} \phi(\underline{i}, \mathbf{a}[\underline{i}])$$

where  $\underline{i}$  is a tuple of variables of sort `INDEX` and  $\phi$  is a quantifier-free  $\Sigma_I \cup \Sigma_E$ -formula.

For instance, the formula

$$\exists i_1 \exists i_2. (i_1 \neq i_2 \wedge \mathbf{a}[i_1] = \text{use} \wedge \mathbf{a}[i_2] = \text{use})$$

expresses that mutual exclusion is violated.

## Format for transitions formulae

**Proposed format for  $\tau$ :** we use **disjunctions** of formulae of the kind

$$\exists \underline{i} \left( \phi_L(\underline{i}, \mathbf{a}[\underline{i}]) \wedge \mathbf{a}' = \lambda j F(\underline{i}, \mathbf{a}[\underline{i}], j, \mathbf{a}[j]) \right) \quad (1)$$

where  $F$  is a case-defined function (cases are described by quantifier-free formulae).

For instance, the formula

$$\exists i. \left( \mathbf{a}[i] = \text{use} \wedge \mathbf{a}' = \lambda j (\text{if } j = i \text{ then } \text{idle} \text{ else } \mathbf{a}[j]) \right)$$

is one of the disjunctions of the transition of the ‘bakery’ algorithm.



## Format for transitions formulae

**Proposed format for  $\tau$ :** we use **disjunctions** of formulae of the kind

$$\exists \underline{i} \left( \phi_L(\underline{i}, \mathbf{a}[\underline{i}]) \wedge \mathbf{a}' = \lambda j F(\underline{i}, \mathbf{a}[\underline{i}], j, \mathbf{a}[j]) \right) \quad (1)$$

where  $F$  is a case-defined function (cases are described by quantifier-free formulae).

For instance, the formula

$$\exists i. \left( \mathbf{a}[i] = \text{use} \wedge \mathbf{a}' = \lambda j (\text{if } j = i \text{ then } \text{idle} \text{ else } \mathbf{a}[j]) \right)$$

is one of the disjunctions of the transition of the ‘bakery’ algorithm.

## Format for transitions formulae

**Extended format for  $\tau$ :** results below apply also in case we use disjunctions of formulae in the more liberal format

$$\exists \underline{i} \exists \underline{e} \left( \phi_L(\underline{e}, \underline{i}, a[\underline{i}]) \wedge a' = \lambda j F(\underline{e}, \underline{i}, a[\underline{i}], j, a[j]) \right) \quad (2)$$

Existentially quantified data variables  $\exists \underline{e}$  are now allowed, but a *quantifier elimination* algorithm must be available for  $T_E$  - crucial for modeling **timed** systems.

An even more liberal format is obtained by replacing  $F$  with a **serial relation** - crucial for modeling nondeterminism in updates.

## Format for transitions formulae

**Extended format for  $\tau$ :** results below apply also in case we use disjunctions of formulae in the more liberal format

$$\exists \underline{i} \exists \underline{e} \left( \phi_L(\underline{e}, \underline{i}, a[\underline{i}]) \wedge a' = \lambda j F(\underline{e}, \underline{i}, a[\underline{i}], j, a[j]) \right) \quad (2)$$

Existentially quantified data variables  $\exists \underline{e}$  are now allowed, but a *quantifier elimination* algorithm must be available for  $T_E$  - crucial for modeling **timed** systems.

An even more liberal format is obtained by replacing  $F$  with a **serial relation** - crucial for modeling nondeterminism in updates.

# Format for transitions formulae

## Universal quantifiers in guards

$$\exists \underline{i} \left( \phi_L(\underline{i}, \mathbf{a}[\underline{i}]) \wedge \forall j \psi(\underline{i}, j, \mathbf{a}[\underline{i}], \mathbf{a}[j]) \wedge \mathbf{a}' = \lambda j F(\underline{i}, \mathbf{a}[\underline{i}], j, \mathbf{a}[j]) \right) \quad (3)$$

can be eliminated by recasting **monotonic abstraction**.

In this declarative context, monotonic abstraction is simulated by **syntactic transformations**.

Roughly speaking, these syntactic transformations consist in adding a Boolean flag (crashed/active) and in relativizing quantifiers to active processes. [See our [JSAT 2013](#) paper for details]

# Format for transitions formulae

Universal quantifiers in guards

$$\exists \underline{i} \left( \phi_L(\underline{i}, \mathbf{a}[\underline{i}]) \wedge \forall j \psi(\underline{i}, j, \mathbf{a}[\underline{i}], \mathbf{a}[j]) \wedge \mathbf{a}' = \lambda j F(\underline{i}, \mathbf{a}[\underline{i}], j, \mathbf{a}[j]) \right) \quad (3)$$

can be eliminated by recasting **monotonic abstraction**.

In this declarative context, monotonic abstraction is simulated by **syntactic transformations**.

Roughly speaking, these syntactic transformations consist in adding a Boolean flag (crashed/active) and in relativizing quantifiers to active processes. [See our [JSAT 2013](#) paper for details]

# Format for transitions formulae

Universal quantifiers in guards

$$\exists \underline{i} \left( \phi_L(\underline{i}, \mathbf{a}[\underline{i}]) \wedge \forall j \psi(\underline{i}, j, \mathbf{a}[\underline{i}], \mathbf{a}[j]) \wedge \mathbf{a}' = \lambda j F(\underline{i}, \mathbf{a}[\underline{i}], j, \mathbf{a}[j]) \right) \quad (3)$$

can be eliminated by recasting **monotonic abstraction**.

In this declarative context, monotonic abstraction is simulated by **syntactic transformations**.

Roughly speaking, these syntactic transformations consist in adding a Boolean flag (crashed/active) and in relativizing quantifiers to active processes. [See our [JSAT 2013](#) paper for details]

# Format for transitions formulae

Universal quantifiers in guards

$$\exists \underline{i} \left( \phi_L(\underline{i}, \mathbf{a}[\underline{i}]) \wedge \forall j \psi(\underline{i}, j, \mathbf{a}[\underline{i}], \mathbf{a}[j]) \wedge \mathbf{a}' = \lambda j F(\underline{i}, \mathbf{a}[\underline{i}], j, \mathbf{a}[j]) \right) \quad (3)$$

can be eliminated by recasting **monotonic abstraction**.

In this declarative context, monotonic abstraction is simulated by **syntactic transformations**.

Roughly speaking, these syntactic transformations consist in adding a Boolean flag (crashed/active) and in relativizing quantifiers to active processes. [See our [JSAT 2013](#) paper for details]

## Key points

- Closure: if  $H(a)$  is an  $\exists^I$ -formula, the formula  $Pre(\tau, H) := \exists a' (\tau(a, a') \wedge H(a'))$  is  $A_I^E$ -equivalent to an effectively computable  $\exists^I$ -formula: **trivial and computationally very cheap!**
- Safety tests are effective: **generally true** (e.g. under mild assumptions on the shape of the initial formula).
- Fixpoint tests are effective: **true under certain assumptions** (but good - still incomplete - algorithms available in general).
- Termination: **true under strong assumptions** (eg embeddability of finitely generated models is a wqo).

See our [\[LMCS 2010\]](#) paper.



## Key points

- Closure: if  $H(a)$  is an  $\exists^I$ -formula, the formula  $Pre(\tau, H) := \exists a' (\tau(a, a') \wedge H(a'))$  is  $A_I^E$ -equivalent to an effectively computable  $\exists^I$ -formula: **trivial and computationally very cheap!**
- Safety tests are effective: **generally true** (e.g. under mild assumptions on the shape of the initial formula).
- Fixpoint tests are effective: **true under certain assumptions** (but good - still incomplete - algorithms available in general).
- Termination: **true under strong assumptions** (eg embeddability of finitely generated models is a wqo).

See our [\[LMCS 2010\]](#) paper.

# Key points

- Closure: if  $H(a)$  is an  $\exists^I$ -formula, the formula  $Pre(\tau, H) := \exists a' (\tau(a, a') \wedge H(a'))$  is  $A_I^E$ -equivalent to an effectively computable  $\exists^I$ -formula: **trivial and computationally very cheap!**
- Safety tests are effective: **generally true** (e.g. under mild assumptions on the shape of the initial formula).
- Fixpoint tests are effective: **true under certain assumptions** (but good - still incomplete - algorithms available in general).
- Termination: **true under strong assumptions** (eg embeddability of finitely generated models is a wqo).

See our [\[LMCS 2010\]](#) paper.

## Key points

- Closure: if  $H(a)$  is an  $\exists^!$ -formula, the formula  $Pre(\tau, H) := \exists a' (\tau(a, a') \wedge H(a'))$  is  $A_I^E$ -equivalent to an effectively computable  $\exists^!$ -formula: **trivial and computationally very cheap!**
- Safety tests are effective: **generally true** (e.g. under mild assumptions on the shape of the initial formula).
- Fixpoint tests are effective: **true under certain assumptions** (but good - still incomplete - algorithms available in general).
- Termination: **true under strong assumptions** (eg embeddability of finitely generated models is a wqo).

See our [\[LMCS 2010\]](#) paper.

## Key points

- Closure: if  $H(a)$  is an  $\exists^I$ -formula, the formula  $Pre(\tau, H) := \exists a' (\tau(a, a') \wedge H(a'))$  is  $A_I^E$ -equivalent to an effectively computable  $\exists^I$ -formula: **trivial and computationally very cheap!**
- Safety tests are effective: **generally true** (e.g. under mild assumptions on the shape of the initial formula).
- Fixpoint tests are effective: **true under certain assumptions** (but good - still incomplete - algorithms available in general).
- Termination: **true under strong assumptions** (eg embeddability of finitely generated models is a wqo).

See our [\[LMCS 2010\]](#) paper.

- 1 The core: a brief review on WSTS
- 2 The Declarative Perspective
- 3 The tool MCMT**
- 4 Software Model Checking Applications

# The tool MCMT

- <http://users.mat.unimi.it/users/ghilardi/mcmt/>
- Obvious client-server architecture
- Client generates proof obligations ([satisfiability modulo theories problems](#))
- Server = state-of-the-art SMT solver (invoked via API)<sup>3</sup>
- Various heuristics implemented (including array acceleration and some interpolation-like abstraction/refinement loops).
- More than 100 problems (from various sources) included in the current distribution 2.5.2.
- Alternative recent implementation (on a parallel architecture, with additional sophisticated algorithms): CUBICLE  
<http://cubicle.lri.fr/>, by S. Conchon et al.

---

<sup>3</sup>[Yices](#) is the SMT-solver employed in MCMT.

# A case study: fault tolerant protocols

We analyzed a classical solution to the reliable broadcast problem (joint work with F. Alberti, E. Pagani, G. P. Rossi).



T. D. Chandra and S. Toueg.

Time and message efficient reliable broadcasts.

*In Proceedings of the 4th international workshop on Distributed Algorithms, 289–303, 1991.*

# A case study: fault tolerant protocols

## Paper Overview

1. First Protocol for *Stopping-failure* model.  
⇒ This model is refined to *Send-Omission* model.
2. First Protocol is unsafe for this model.
3. Second modified version: still unsafe for *Send-Omission* model.
4. Third modified version: now safe for *Send-Omission* model!



# A case study: fault tolerant protocols

**MCMT confirms all that!** In the last case, a little proof plan was needed (we asked the tool to first prove some lemmas suggested by us and then to attack the main task).

Problem	result	depth	#nodes	#deleted	#vars	#SMT calls	#inv.	time (sec)
Crash	SAFE	13	113	21	4	1731	0	0.75
Send_Omission (1)	UNSAFE	12	464	26	3	16253	0	14.16
Send_Omission (2)	UNSAFE	34	9679	770	6	1118959	0	30m 18.15s
Send_Omission (3)	SAFE	32	571	72	4	547054	94 (+7)	6m 57.19s

## Algorithm 1 Pseudo-code for Algorithms 1, 2, and 3

### Initialization:

```

if ( $p$  is the sender)
  then  $estimate_p \leftarrow m$ ;  $coord\_id_p \leftarrow 0$ ;
  else  $estimate_p \leftarrow \perp$ ;  $coord\_id_p \leftarrow -1$ ;
 $state_p \leftarrow undecided$ ;

```

### End Initialization

```

for  $c \leftarrow 1, 2, \dots$  do // Process  $c$  becomes coordinator for four rounds

```

#### Round 1:

```

All undecided processes  $p$  send request ( $estimate_p$ ,  $coord\_id_p$ ) to  $c$ ;
if ( $c$  does not receive any request) then it skips rounds 2 to 4;
  else  $estimate_c \leftarrow estimate_p$  with largest  $coord\_id_p$ ;

```

#### Round 2:

```

 $c$  multicasts  $estimate_c$ ;
All undecided processes  $p$  that receive  $estimate_c$  do
   $estimate_p \leftarrow estimate_c$  and  $coord\_id_p \leftarrow c$ ;

```

#### Round 3:

```

All undecided processes  $p$  that do not receive  $estimate_c$  send(NACK) to  $c$ ;

```

#### Round 4:

```

if ( $c$  does not receive any NACK) then  $c$  multicasts Decide; else  $c$  HALTS;
All undecided processes  $p$  that receive Decide do
   $decision_p \leftarrow estimate_p$ ;
   $state_p \leftarrow DECIDED$ ;

```

```

end for

```

## Further case studies: simulations via counter abstractions

Further fault-tolerant protocols require resilience guards. This is the case of the 'General Omission' from the above paper or of byzantine broadcast primitive from



T.K. Srikanth and S. Toueg.

Simulating authenticated broadcasts to derive simple fault-tolerant algorithms.

*Distributed Computing*, 2(2):80–94, 1987.

## Further case studies: simulations via counter abstractions

In principle, array-based formalisms support reasoning on resilience guards: one uses suitable transitions loops like

```
int I, J = 0;
for(I = 0; I ≤ N; I ++){if(received_from[I] == 1)J ++;}
```

and then uses the value of  $J$  in resilience guards.

In practice, the actual heuristics for preventing non-termination implemented in MCMT may fail to succeed when such solutions are adopted. Some success can be nevertheless obtained by specifying ad hoc abstraction parameters.

## Further case studies: simulations via counter abstractions

**In principle**, array-based formalisms support reasoning on resilience guards: one uses suitable transitions loops like

```
int I, J = 0;  
for(I = 0; I ≤ N; I ++){if(received_from[I] == 1)J ++;}
```

and then uses the value of  $J$  in resilience guards.

**In practice**, the actual heuristics for preventing non-termination implemented in MCMT may fail to succeed when such solutions are adopted. Some success can be nevertheless obtained by specifying ad hoc abstraction parameters.

## Further case studies: simulations via counter abstractions

While waiting for implementation of required additional features (some cardinality constraint reasoning, two-dimensional arrays handling, more support for decidable  $\exists^+\forall^*$ -fragments), we made the following experiment, leading to somewhat surprising results (joint work with F.Alberti, E.Pagani, A.Orsini).

We manually built counter abstraction simulations of the above verification problems and run both MCMT and some well-established IC3-based model-checkers on the resulting specifications.

Despite the fact that MCMT only have at its disposal basic acceleration techniques for handling numerical problems (i.e. problems where arrays are not involved, this is the case of counting abstractions), the tool was nevertheless able to solve them.

## Further case studies: simulations via counter abstractions

File	MCMT			Z3 ( $\mu Z$ )	nuXmv
	Depth	Nodes	Time	Time	Time
crash.mcmt	8	39	0.19"	0.69"	0.36"
sndom.mcmt	21	1772	77"	846"	22"
genom.mcmt	42	10102	6175"	t.o.	t.o.
byz_unforg.mcmt	7	51	0.42"	0.19"	0.04"
byz_corr.mcmt	7	131	6.16"	0.13"	0.24"
byz_relayA.mcmt	6	38	0.23"	0.07"	0.13"
byz_relayB.mcmt	8	185	5.21"	0.05"	0.16"

- 1 The core: a brief review on WSTS
- 2 The Declarative Perspective
- 3 The tool MCMT
- 4 Software Model Checking Applications**



# Monotonic Abstraction via Instantiation

Let us examine syntactic monotonic abstraction from another point of view. If we take an existential formula  $K$  and a transition  $\tau_h$  containing a universal guard, the preimage  $\text{Pre}(\tau_h, K)$  has the form

$$\exists \underline{j} \forall k \psi(\underline{j}, k, a[\underline{j}], a[k]), \quad (4)$$

where  $\psi$  is quantifier-free.

Instead of modifying syntactically  $\tau_h$  in order to eliminate from it the universal guard, we could over-approximate (4) via an existential formula at runtime (i.e. during backward search).

# Monotonic Abstraction via Instantiation

Let us examine syntactic monotonic abstraction from another point of view. If we take an existential formula  $K$  and a transition  $\tau_h$  containing a universal guard, the preimage  $\text{Pre}(\tau_h, K)$  has the form

$$\exists \underline{i} \forall k \psi(\underline{i}, k, a[\underline{i}], a[k]), \quad (4)$$

where  $\psi$  is quantifier-free.

Instead of modifying syntactically  $\tau_h$  in order to eliminate from it the universal guard, we could over-approximate (4) via an existential formula at runtime (i.e. during backward search).

# Monotonic Abstraction via Instantiation

The proposed overapproximation is the existential formula

$$\exists \underline{i} \bigwedge_t \psi(\underline{i}, t, a[\underline{i}], a[t]), \quad (5)$$

varying  $t$  among a set of terms  $X$ . We may call (5) a *syntactic monotonic abstraction of the formula* (4) (notice that this notion is relative to  $X$ ).

If one take the obvious choice  $X := \underline{i}$ , we do not get *in the end* anything different from syntactic monotonic abstraction applied to transitions. But the situation becomes different (we have more flexibility), when there is some arithmetics on indexes.

# Monotonic Abstraction via Instantiation

The proposed overapproximation is the existential formula

$$\exists \underline{i} \bigwedge_t \psi(\underline{i}, t, a[\underline{i}], a[t]), \quad (5)$$

varying  $t$  among a set of terms  $X$ . We may call (5) a *syntactic monotonic abstraction of the formula* (4) (notice that this notion is relative to  $X$ ).

If one take the obvious choice  $X := \underline{i}$ , we do not get *in the end* anything different from syntactic monotonic abstraction applied to transitions. But the situation becomes different (we have more flexibility), when there is some arithmetics on indexes.

# Array Acceleration

This observation can be exploited in software model checking when dealing with programs for arrays of unbounded length. We show the technique by an example.

The following ‘initialize-and-test’ simple example is considered problematic for CEGAR techniques:

```
for (I=0; I!= a_length; I++) a[I]=0;
for (J=0; J!= a_length; J++) assert (a[J]==0);
```

# Array Acceleration

This observation can be exploited in software model checking when dealing with programs for arrays of unbounded length. We show the technique by an example.

The following ‘initialize-and-test’ simple example is considered problematic for CEGAR techniques:

```
for (I=0; I != a_length; I++) a[I]=0;
for (J=0; J != a_length; J++) assert (a[J]==0);
```

# Array Acceleration

Indeed backward search trivially diverges here:

$$p = 2 \wedge J \neq a\_length \wedge a[J] \neq 0$$

$$p = 2 \wedge J + 1 \neq a\_length \wedge a[J + 1] \neq 0 \wedge a[J] = 0$$

...

$$p = 2 \wedge J + n \neq a\_length \wedge a[J + n] \neq 0 \wedge \bigwedge_{k=J}^{J+n-1} a[k] = 0$$

...

# Array Acceleration

Indeed backward search trivially diverges here:

$$p = 2 \wedge J \neq a\_length \wedge a[J] \neq 0$$

$$p = 2 \wedge J + 1 \neq a\_length \wedge a[J + 1] \neq 0 \wedge a[J] = 0$$

...

$$p = 2 \wedge J + n \neq a\_length \wedge a[J + n] \neq 0 \wedge \bigwedge_{k=J}^{J+n-1} a[k] = 0$$

...



## Array Acceleration

To stop divergence, we need to re-introduce quantifiers. One possible solution is to summarize the effect of  $n$  executions of a loop into a single transition, representing transitive closure. This technique is known as *acceleration* in model-checking and has been extensively investigated for fragments of Presburger arithmetic.

In the example above, we can accelerate the two loops, resulting in

$$\begin{aligned} & \exists n > 0 \left( p = 1 \wedge \forall k (I \leq k < I + n \rightarrow k \neq a\_length) \wedge p' = 1 \wedge \right. \\ & \quad \left. I' = I + n \wedge J' = J \wedge a' = wr(a, [I, I + n - 1], 0) \right); \\ & \exists n > 0 \left( p = 2 \wedge \forall k (J \leq k < J + n \rightarrow k \neq a\_length \wedge a[k] = 0) \right. \\ & \quad \left. \wedge p' = 2 \wedge I' = I \wedge J' = J + n \wedge a' = a \right). \end{aligned}$$

## Array Acceleration

To stop divergence, we need to re-introduce quantifiers. One possible solution is to summarize the effect of  $n$  executions of a loop into a single transition, representing transitive closure. This technique is known as *acceleration* in model-checking and has been extensively investigated for fragments of Presburger arithmetic.

In the example above, we can accelerate the two loops, resulting in

$$\begin{aligned} & \exists n > 0 \left( p = 1 \wedge \forall k (I \leq k < I + n \rightarrow k \neq a\_length) \wedge p' = 1 \wedge \right. \\ & \quad \left. I' = I + n \wedge J' = J \wedge a' = wr(a, [I, I + n - 1], 0) \right); \\ & \exists n > 0 \left( p = 2 \wedge \forall k (J \leq k < J + n \rightarrow k \neq a\_length \wedge a[k] = 0) \right. \\ & \quad \left. \wedge p' = 2 \wedge I' = I \wedge J' = J + n \wedge a' = a \right). \end{aligned}$$

# Array Acceleration

The plan is now clear: we got existential transitions with universal guards, **so let us apply monotonic abstraction to them!**

The idea is quite successful indeed in the applications: a lot of benchmarks are easily solved.

# Array Acceleration

The plan is now clear: we got existential transitions with universal guards, **so let us apply monotonic abstraction to them!**

The idea is quite successful indeed in the applications: a lot of benchmarks are easily solved.

# Monotonic Abstraction for Arrays

There are however remarkable differences in the use of abstraction here wrt the distributed case.

- Monotonic abstraction here is just an abstraction technique among many others (we loose intuitive justifications in terms of crash failures).
- Monotonic abstraction can produce spurious traces, but here we can *ignore* such spurious traces: no refinement is needed, one simply drops unsafe traces containing accelerations (if the system is unsafe, unsafety should be discovered without acceleration!)
- Our monotonic abstraction is purely syntactic, hence it can be used *in combination with other abstraction techniques* (in MCMT it is combined with predicate abstraction via interpolants).

# Monotonic Abstraction for Arrays

There are however remarkable differences in the use of abstraction here wrt the distributed case.

- Monotonic abstraction here is just an abstraction technique among many others (we loose intuitive justifications in terms of crash failures).
- Monotonic abstraction can produce spurious traces, but here we can *ignore* such spurious traces: no refinement is needed, one simply drops unsafe traces containing accelerations (if the system is unsafe, unsafety should be discovered without acceleration!)
- Our monotonic abstraction is purely syntactic, hence it can be used *in combination with other abstraction techniques* (in MCMT it is combined with predicate abstraction via interpolants).

# Monotonic Abstraction for Arrays

There are however remarkable differences in the use of abstraction here wrt the distributed case.

- Monotonic abstraction here is just an abstraction technique among many others (we loose intuitive justifications in terms of crash failures).
- Monotonic abstraction can produce spurious traces, but here we can *ignore* such spurious traces: no refinement is needed, one simply drops unsafe traces containing accelerations (if the system is unsafe, unsafety should be discovered without acceleration!)
- Our monotonic abstraction is purely syntactic, hence it can be used *in combination with other abstraction techniques* (in MCMT it is combined with predicate abstraction via interpolants).

# Monotonic Abstraction for Arrays

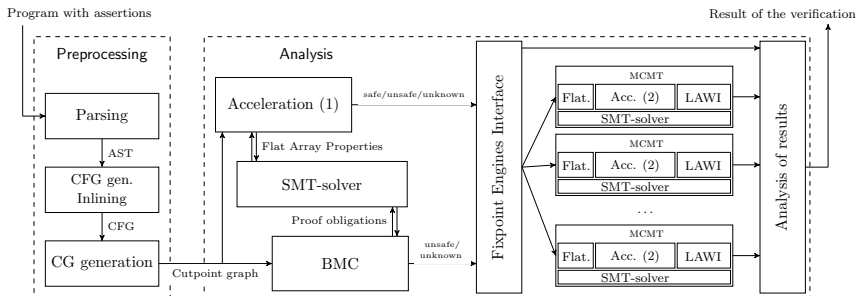
There are however remarkable differences in the use of abstraction here wrt the distributed case.

- Monotonic abstraction here is just an abstraction technique among many others (we loose intuitive justifications in terms of crash failures).
- Monotonic abstraction can produce spurious traces, but here we can *ignore* such spurious traces: no refinement is needed, one simply drops unsafe traces containing accelerations (if the system is unsafe, unsafety should be discovered without acceleration!)
- Our monotonic abstraction is purely syntactic, hence it can be used *in combination with other abstraction techniques* (in MCMT it is combined with predicate abstraction via interpolants).



# The BOOSTER Tool

An acceleration-based software model-checker



F. Alberti, S. Ghilardi, and N. Sharygina.

Booster: an acceleration-based verification framework for array programs  
In *ATVA*, Springer, 2014. To appear.

# BOOSTER: Experiments

FILENAME	STATUS	ACC+ABS	ABS	ACC
data_structures/set_multi_proc.c	SAFE	1.600	TO	TO
data_structures/set_multi_proc_trivial.c	SAFE	0.208	0.208	0.314
data_structures/set_multi_proc_unsafe.c	UNSAFE	1.946	1.257	2.102
sanfoundry/06.c	SAFE	0.016	TO	0.016
sanfoundry/07.c	SAFE	4.623	TO	TO
sanfoundry/08.c	SAFE	2.926	TO	TO
sanfoundry/09.c	SAFE	8.447	TO	TO
sanfoundry/10.c	SAFE	0.157	TO	TO
sanfoundry/24.c	SAFE	0.101	0.071	0.085
sanfoundry/27.c	SAFE	0.066	0.076	108.724
sanfoundry/28.c	SAFE	0.676	0.151	63.932
sanfoundry/39.c	SAFE	1.832	TO	TO
sorting/bubblesort.c	SAFE	0.233	0.107	0.407
sorting/bubblesort_unsafe.c	UNSAFE	0.090	0.090	0.135
sorting/selectionsort.c	SAFE	85.326	TO	TO
sorting/selectionsort_unsafe.c	UNSAFE	1.500	1.658	1.629
standard/allDiff_safe.c	SAFE	0.010	0.044	0.010
standard/allDiff_unsafe.c	UNSAFE	0.007	0.036	0.006
svcomp/loops/array_false-unreach-label.c	UNSAFE	0.135	0.039	0.094
svcomp/loops/array_true-unreach-label.c	SAFE	0.169	0.057	TO
svcomp/loops/compact_false-unreach-label.c	UNSAFE	0.010	0.051	0.010
svcomp/loops/heavy_false-unreach-label.c	SAFE	0.363	0.277	TO
svcomp/loops/heavy_true-unreach-label.c	UNSAFE	0.296	0.217	0.393
svcomp/loops/linear_search_false-unreach-label.c	UNSAFE	0.154	0.053	0.062
svcomp/loops/linear_search_true-unreach-label.c	SAFE	0.016	0.101	TO
svcomp/loops/nec11_false-unreach-label.c	UNSAFE	0.053	0.040	0.75
svcomp/loops/nec40_true-unreach-label.c	SAFE	0.010	0.607	0.16
svcomp/loops/string_true-unreach-label.c	SAFE	0.860	0.781	1.04
svcomp/loops/sum_array_false-unreach-label.c	UNSAFE	0.068	0.059	0.104
svcomp/loops/sum_array_true-unreach-label.c	SAFE	0.070	0.080	TO

# BOOSTER: Comparisons (?)

BENCHMARK	COMPASS	Z3 HORN	ARMC	DUALITY	BOOSTER
init	0.01	0.06	0.15	0.72	0.01
init_non_constant	0.02	0.08	0.48	6.60	0.01
init_partial	0.01	0.03	0.14	2.60	0.01
init_partial_buggy	0.02	0.01	0.07	0.03	0.01
init_even	0.04	TO	?	TO	0.02
init_even_buggy	0.04	NA	NA	NA	0.01
copy	0.01	0.04	0.20	1.40	0.01
copy_partial	0.01	0.04	0.21	1.80	0.01
copy_odd	0.04	TO	?	4.50	TO
copy_odd_buggy	0.05	NA	NA	NA	0.07
reverse	0.03	0.12	2.28	8.50	0.02
reverse_buggy	0.04	0.01	0.08	0.03	0.01
swap	0.12	0.41	3.0	40.60	0.12
swap_buggy	0.11	NA	NA	NA	0.03
double_swap	0.16	1.37	4.4	TO	0.34
check_strcpy	0.07	0.05	0.15	0.62	0.02
check_memcpy	0.04	0.04	0.20	16.30	0.02
find	0.02	0.01	0.08	0.38	0.26
find_first_nonnull	0.02	0.01	0.08	0.39	0.09
array_append	0.02	0.04	1.76	1.50	0.02
merge_interleave	0.09	0.04	?	1.50	0.15
merge_interleave_buggy	0.11	NA	NA	NA	0.01

# Conclusions

- Monotonic abstraction is a technique originated in model checking parameterized distributed systems.
- In a declarative context, monotonic abstraction can be turned to a **syntactic** operation.
- This syntactic reformulation can be **combined with acceleration** in other applications domains (eg model checking sequential array programs).
- The resulting technique turns out to be **simple, easily implementable and quite effective**.
- It can also be **integrated** in a natural way with other abstraction methodologies.

# Conclusions

- Monotonic abstraction is a technique originated in model checking parameterized distributed systems.
- In a declarative context, monotonic abstraction can be turned to a **syntactic** operation.
- This syntactic reformulation can be **combined with acceleration** in other applications domains (eg model checking sequential array programs).
- The resulting technique turns out to be **simple, easily implementable and quite effective**.
- It can also be **integrated** in a natural way with other abstraction methodologies.

# Conclusions

- Monotonic abstraction is a technique originated in model checking parameterized distributed systems.
- In a declarative context, monotonic abstraction can be turned to a **syntactic** operation.
- This syntactic reformulation can be **combined with acceleration** in other applications domains (eg model checking sequential array programs).
- The resulting technique turns out to be **simple, easily implementable and quite effective**.
- It can also be **integrated** in a natural way with other abstraction methodologies.

# Conclusions

- Monotonic abstraction is a technique originated in model checking parameterized distributed systems.
- In a declarative context, monotonic abstraction can be turned to a **syntactic** operation.
- This syntactic reformulation can be **combined with acceleration** in other applications domains (eg model checking sequential array programs).
- The resulting technique turns out to be **simple, easily implementable and quite effective**.
- It can also be **integrated** in a natural way with other abstraction methodologies.

# Conclusions

- Monotonic abstraction is a technique originated in model checking parameterized distributed systems.
- In a declarative context, monotonic abstraction can be turned to a **syntactic** operation.
- This syntactic reformulation can be **combined with acceleration** in other applications domains (eg model checking sequential array programs).
- The resulting technique turns out to be **simple, easily implementable and quite effective**.
- It can also be **integrated** in a natural way with other abstraction methodologies.