Monotonic Abstraction Techniques: from Parametric to Software Model Checking

F. Alberti$^1$, S. Ghilardi$^2$, N. Sharygina$^1$

$^1$University of Lugano, Switzerland
$^2$University of Milan, Italy

Milano, September 25, 2014
Aim of the talk

My talk will report recent experience concerning the development of an SMT-based model checker. Main features:

- declarative approach;
- use of decision procedures for combined theories;
- prominent role played by array fragments;
- 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

My talk will report recent experience concerning the development of an SMT-based model checker. Main features:

- declarative approach;
- use of decision procedures for combined theories;
- prominent role played by array fragments;
- 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

My talk will report recent experience concerning the development of an SMT-based model checker. Main features:

- declarative approach;
- use of decision procedures for combined theories;
- prominent role played by array fragments;
- 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

My talk will report recent experience concerning the development of an SMT-based model checker. Main features:

- declarative approach;
- use of decision procedures for combined theories;
- prominent role played by array fragments;
- 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

My talk will report recent experience concerning the development of an SMT-based model checker. Main features:

- declarative approach;
- use of decision procedures for combined theories;
- prominent role played by array fragments;
- 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

My talk will report recent experience concerning the development of an SMT-based model checker. Main features:

- declarative approach;
- use of decision procedures for combined theories;
- prominent role played by array fragments;
- 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

My talk will report recent experience concerning the development of an SMT-based model checker. Main features:

- declarative approach;
- use of decision procedures for combined theories;
- prominent role played by array fragments;
- 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

My talk will report recent experience concerning the development of an SMT-based model checker. Main features:

- declarative approach;
- use of decision procedures for combined theories;
- prominent role played by array fragments;
- 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

My talk will report recent experience concerning the development of an SMT-based model checker. Main features:

- declarative approach;
- use of decision procedures for combined theories;
- prominent role played by array fragments;
- 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).
Outline

1. Infinite state model-checking
2. Our Declarative Proposal
3. The tool MCMT
4. Software Model Checking Applications
Outline

1. Infinite state model-checking
2. Our Declarative Proposal
3. The tool MCMT
4. Software Model Checking Applications
Outline

1. Infinite state model-checking
2. Our Declarative Proposal
3. The tool MCMT
4. Software Model Checking Applications
Outline

1. Infinite state model-checking
2. Our Declarative Proposal
3. The tool MCMT
4. Software Model Checking Applications
1. Infinite state model-checking

2. Our Declarative Proposal

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\(^1\) (wqo) on \(S\);
- each \(\tau_{\lambda}\) is monotonic:

\[\begin{align*}
S_1 &\preceq S_2 \\
\downarrow_{\lambda} &\preceq \exists \\
S_3 &\preceq S_4
\end{align*}\]

\(^1\)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\(^1\) (wqo) on \(S\);
- each \(\tau_\lambda\) is monotonic:

\[
\begin{array}{c}
S_1 \preceq S_2 \\
\downarrow_\lambda \\
S_3 \preceq \exists \ S_4
\end{array}
\]

\(^1\)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\(^1\) (wqo) on \(S\);
- each \(\tau_{\lambda}\) is monotonic:

\[
\begin{array}{c}
S_1 \preceq S_2 \\
\downarrow_{\lambda} \\
S_3 \preceq \exists S_4
\end{array}
\]

\(^1\)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\(^1\) (wqo) on \(S\);
- each \(\tau_\lambda\) is monotonic:

\[
\begin{align*}
S_1 & \preceq S_2 \\
\downarrow_\lambda & \downarrow_\lambda \\
S_3 & \preceq \exists S_4
\end{align*}
\]

\(^1\)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\(^1\) (wqo) on \(S\);
- each \(\tau_\lambda\) is monotonic:

\[
\begin{array}{c}
S_1 \preceq S_2 \\
\downarrow_\lambda \quad \downarrow_\lambda \\
S_3 \preceq \exists S_4
\end{array}
\]

---

\(^1\)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\(^1\) (wqo) on $S$;
- **each $\tau_{\lambda}$ is monotonic:**

\[
\begin{align*}
S_1 & \preceq S_2 \\
\downarrow^{\lambda} & \downarrow^{\lambda} \\
S_3 & \preceq S_4
\end{align*}
\]

\(^1\)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 $K$:

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

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

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

- Since $\preceq$ is a wqo, upsets can be finitely represented by their finitely many minimal elements
Backward Reachability

Checking that a set $K$ of unsafe states is (un-)reachable from a set $I$ of initial states

\[
\text{function } \text{BReach}(K) \\
\quad i \leftarrow 0; \ BR^0(\tau, K) \leftarrow K; \ K^0 \leftarrow K \\
\quad \text{if } BR^0(\tau, K) \cap I \neq \emptyset \text{ then return unsafe} \\
\quad \text{repeat} \\
\quad \quad K^{i+1} \leftarrow \text{Pre}(\tau, K^i) \\
\quad \quad BR^{i+1}(\tau, K) \leftarrow BR^i(\tau, K) \cup K^{i+1} \\
\quad \quad \text{if } BR^{i+1}(\tau, K) \cap I \neq \emptyset \text{ then return unsafe} \\
\quad \quad \text{else } i \leftarrow i + 1 \\
\quad \text{until } BR^{i+1}(\tau, K) \subseteq BR^i(\tau, K) \\
\quad \text{return safe} \\
\text{end}
\]
Backward Reachability

Checking that a set $K$ of unsafe states is (un-)reachable from a set $I$ of initial states

function $\text{BReach}(K)$

\[
i \leftarrow 0; \quad BR^0(\tau, K) \leftarrow K; \quad K^0 \leftarrow K
\]

if $BR^0(\tau, K) \cap I \neq \emptyset$ then return unsafe

repeat

\[
K^{i+1} \leftarrow \text{Pre}(\tau, K^i)
\]

$BR^{i+1}(\tau, K) \leftarrow BR^i(\tau, K) \cup K^{i+1}$

if $BR^{i+1}(\tau, K) \cap I \neq \emptyset$ then return unsafe

else $i \leftarrow i + 1$

until $BR^{i+1}(\tau, K) \subseteq BR^i(\tau, K)$

return safe

dec

S. Ghilardi (UniMi)
Termination

- Since $\preceq$ is a wqo, the algorithm terminates.
- Extensions to cases in which $\preceq$ is not a wqo often terminate ‘in practice’.
Termination

1. Since $\preceq$ is a wqo, the algorithm terminates.
2. Extensions to cases in which $\preceq$ is not a wqo often terminate ‘in practice’.
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: 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.
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: 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.
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: 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.
Infinite state model-checking

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: 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: 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. Infinite state model-checking

2. Our Declarative Proposal

3. The tool MCMT

4. Software Model Checking Applications
Array-based Systems

**OUR 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 $T = (\Sigma, C)$, where $\Sigma$ is a first-order signature and $C$ is a class of $\Sigma$-structures (called the models of $T$). Satisfiability of at least quantifier-free formulae in $C$ 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 $A_E^I$. 
Array-based Systems

**OUR 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 $T = (\Sigma, C)$, where $\Sigma$ is a first-order signature and $C$ is a class of $\Sigma$-structures (called the models of $T$). Satisfiability of at least quantifier-free formulae in $C$ 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 $A^E_I$. 
Array-based Systems

**OUR 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 \( T = (\Sigma, C) \), where \( \Sigma \) is a first-order signature and \( C \) is a class of \( \Sigma \)-structures (called the models of \( T \)). Satisfiability of at least quantifier-free formulae in \( C \) 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 \( A^E_I \).
Array-based Systems

**OUR 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 $A^E_I$. 
Array-Based Systems

- the sort \texttt{INDEX} is constrained by $T_I$;
- the sort \texttt{ELEM} is constrained by $T_E$;
- the sort \texttt{ARRAY} represents arrays of \texttt{ELEM} defined on \texttt{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 \texttt{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:

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

- 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 $S$ is the following: given a formula $K(a)$, is

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

  $A_I^E$-satisfiable for some $n$?
Array-Based Systems

- An array-based system on $A^E_I$ with array state variable $a$ is the following pair of formulae:
  \[ S = \langle I(a), \tau(a, a') \rangle. \]

- A state of an array-based system is an assignment to the variable $a$ in a model of $A^E_I$.

- A safety problem for $S$ is the following: given a formula $K(a)$, is $A^E_I$-satisfiable for some $n$?

  \[ I(a_0) \land \tau(a_0, a_1) \land \cdots \land \tau(a_{n-1}, a_n) \land K(a_n) \]
Array-Based Systems

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

  \[ S = \langle I(a), \tau(a, a') \rangle. \]

- A **state** of an array-based system is an assignment to the variable $a$ in a model of $A^E_i$.

- A **safety problem** for $S$ is the following: given a formula $K(a)$, is

  \[ I(a_0) \land \tau(a_0, a_1) \land \cdots \land \tau(a_{n-1}, a_n) \land K(a_n) \]

  $A^E_i$-satisfiable for some $n$?
Revisiting Backward Reachability

Idea: recast symbolically the backward reachability algorithm

```
function BReach(K)
    i ← 0; BR^0(τ, K) ← K; K^0 ← K
    if BR^0(τ, K) ∩ l ≠ ∅ then return unsafe
    repeat
      K^{i+1} ← Pre(τ, K^i)
      BR^{i+1}(τ, K) ← BR^i(τ, K) ∪ K^{i+1}
      if BR^{i+1}(τ, K) ∩ l ≠ ∅ then return unsafe
      else i ← i + 1
    until BR^{i+1}(τ, K) ⊆ BR^i(τ, K)
    return safe
end
```
Revisiting Backward Reachability

Idea: recast symbolically the backward reachability algorithm

```
function BReach(K)
    i ← 0; BR^0(τ, K) ← K; K^0 ← K
    if A^E_check(BR^0(τ, K) ∧ l) = sat then return unsafe
    repeat
        K^{i+1} ← Pre(τ, K^i)
        BR^{i+1}(τ, K) ← BR^{i}(τ, K) ∨ K^{i+1}
        if A^E_check(BR^{i+1}(τ, K) ∧ l) = sat then return unsafe
        else i ← i + 1
    until A^E_check(¬(BR^{i+1}(τ, K) → BR^{i}(τ, K))) = unsat
    return safe
end
```

But this is problematic... unless right formats for l, τ, K are found!
Format for initialization formulae

Proposed format for \( \forall I \)-formulae

\[
\forall i \phi(i, a[i])
\]

where \( i \) is a tuple of variables of sort \text{INDEX} and \( \phi \) is a quantifier-free \( \Sigma_I \cup \Sigma_E \)-formula\(^2\)

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

\( \forall I \)-formulae can also be used to express invariants

\(^2\)If \( i = i_1, \ldots, i_n \), then \( a[i] \) is the tuple of terms \( a[i_1], \ldots, a[i_n] \) having sort \text{ELEM}.\)
Proposed format for $I$: $\forall^I$-formulae

$$\forall i \, \phi(i, a[i])$$

where $i$ is a tuple of variables of sort $\text{INDEX}$ and $\phi$ is a quantifier-free $\Sigma_I \cup \Sigma_E$-formula$^2$

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

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

---

$^2$If $i = i_1, \ldots, i_n$, then $a[i]$ is the tuple of terms $a[i_1], \ldots, a[i_n]$ having sort $\text{ELEM}$. 
Format for initialization formulae

Proposed format for $\forall$-formulae

$$\forall i \phi(i, a[i])$$

where $i$ is a tuple of variables of sort $\text{INDEX}$ and $\phi$ is a quantifier-free $\Sigma_I \cup \Sigma_E$-formula\(^2\).

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

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

\(^2\)If $i = i_1, \ldots, i_n$, then $a[i]$ is the tuple of terms $a[i_1], \ldots, a[i_n]$ having sort $\text{ELEM}$.
Format for unsafety problems formulae

Proposed format for $K$: $\exists I$-formulae

$$\exists i \phi(i, a[i])$$

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

For instance, the formula

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

expresses that mutual exclusion is violated.
Format for unsafety problems formulae

 Proposed format for $K$: $\exists^I$-formulae

$$\exists i \phi(i, a[i])$$

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

For instance, the formula

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

expresses that mutual exclusion is violated.
Format for transitions formulae

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

$$\exists i \left( \phi_L(i, a[i]) \land a' = \lambda j F(i, a[i], j, a[j]) \right)$$

(1)

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

For instance, the formula

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

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

$$\exists i \left( \phi_L(i, a[i]) \land a' = \lambda j F(i, a[i], j, a[j]) \right)$$

(1)

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

For instance, the formula

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

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

\[
\exists i \exists e \left( \phi_L(e, i, a[i]) \land a' = \lambda j F(e, i, a[i], j, a[j]) \right) \tag{2}
\]

Existentially quantified data variables \( \exists e \) 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

$$\exists i \exists e \left( \phi_L(e, i, a[i]) \land a' = \lambda j F(e, i, a[i], j, a[j]) \right)$$

(2)

Existentially quantified data variables $\exists e$ 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

\[ \exists i \left( \phi_L(i, a[i]) \land \forall j \psi(i, j, a[i], a[j]) \land a' = \lambda j F(i, a[i], j, a[j]) \right) \]  

(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]
Our Declarative Proposal

Format for transitions formulae

Universal quantifiers in guards

\[ \exists i \left( \phi_L(i, a[i]) \land \forall j \psi(i, j, a[i], a[j]) \land a' = \lambda j F(i, a[i], j, a[j]) \right) \] (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

\[ \exists i \left( \phi_L(i, a[i]) \land \forall j \psi(i, j, a[i], a[j]) \land a' = \lambda j F(i, a[i], j, a[j]) \right) \]  \hspace{1cm} (3)

can be eliminated by recasting \textbf{monotonic abstraction}.

In this declarative context, monotonic abstraction is simulated by \textbf{syntactic trasformations}.

Roughly speaking, these syntactic trasformations 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

\[
\exists i \left( \phi_L(i, a[i]) \land \forall j \psi(i, j, a[i], a[j]) \land a' = \lambda j F(i, a[i], j, a[j]) \right) \tag{3}
\]

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

- Clusure: if $H(a)$ is an $\exists^I$-formula, the formula
  
  $Pre(\tau, H) := \exists a' (\tau(a, a') \land H(a'))$
  
  is $A_E^I$-equivalent to an effectively computable $\exists^I$-formula: true and easy!

- 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

- Clusure: if $H(a)$ is an $\exists^I$-formula, the formula
  $Pre(\tau, H) := \exists a' (\tau(a, a') \land H(a'))$ is $A^E_I$-equivalent to an effectively computable $\exists^I$-formula: true and easy!

- 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 (e.g. embeddability of finitely generated models is a wqo).

See our [LMCS 2010] paper.
Key points

- Clusure: if $H(a)$ is an $\exists^I$-formula, the formula
  
  $\text{Pre}(\tau, H) := \exists a' (\tau(a, a') \land H(a'))$
  
  is $A^E_I$-equivalent to an effectively computable $\exists^I$-formula: true and easy!

- 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 (e.g. embeddability of finitely generated models is a wqo).

See our [LMCS 2010] paper.
Key points

- **Clusure**: if $H(a)$ is an $\exists^I$-formula, the formula
  
  $$Pre(\tau, H) := \exists a' \ (\tau(a, a') \land H(a'))$$

  is $A^E_I$-equivalent to an effectively computable $\exists^I$-formula: **true and easy**!

- **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** (e.g. embeddability of finitely generated models is a wqo).

See our [LMCS 2010] paper.
Key points

- Clusure: if $H(a)$ is an $\exists^I$-formula, the formula $Pre(\tau, H) := \exists a' (\tau(a, a') \land H(a'))$ is $A^E_I$-equivalent to an effectively computable $\exists^I$-formula: true and easy!

- 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 (e.g. embeddability of finitely generated models is a wqo).

See our [LMCS 2010] paper.
1. Infinite state model-checking

2. Our Declarative Proposal

3. The tool MCMT

4. Software Model Checking Applications
The tool MCMT

- [http://users.mat.unimi.it/users/ghilardi/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)
- Various heuristics implemented.

---

\(^3\)Yices is the SMT-solver employed in MCMT.
MCMT: mutual exclusion and cache coherence protocols

We first report benchmarks included in the distribution (in the best settings for the tool).4

Mutual Exclusion

<table>
<thead>
<tr>
<th>Problem</th>
<th>depth</th>
<th>#nodes</th>
<th>#deleted</th>
<th>#SMT calls</th>
<th>#inv.</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bakery_Lamport</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>222</td>
<td>7</td>
<td>0.03</td>
</tr>
<tr>
<td>Bakery_Bogus</td>
<td>8</td>
<td>90</td>
<td>14</td>
<td>1440</td>
<td>7</td>
<td>0.44</td>
</tr>
<tr>
<td>Distrib_Lamport</td>
<td>23</td>
<td>248</td>
<td>42</td>
<td>19622</td>
<td>7</td>
<td>27.87</td>
</tr>
<tr>
<td>Rickart_Agrawala</td>
<td>13</td>
<td>458</td>
<td>119</td>
<td>35241</td>
<td>0</td>
<td>148.24</td>
</tr>
<tr>
<td>Szymanski_atomic</td>
<td>9</td>
<td>21</td>
<td>9</td>
<td>3102</td>
<td>39</td>
<td>0.82</td>
</tr>
</tbody>
</table>

Cache Coherence

<table>
<thead>
<tr>
<th>Problem</th>
<th>depth</th>
<th>#nodes</th>
<th>#deleted</th>
<th>#SMT calls</th>
<th>#inv.</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>German</td>
<td>26</td>
<td>2121</td>
<td>255</td>
<td>117121</td>
<td>0</td>
<td>60.00</td>
</tr>
<tr>
<td>German_buggy</td>
<td>16</td>
<td>1631</td>
<td>203</td>
<td>40884</td>
<td>0</td>
<td>26.01</td>
</tr>
<tr>
<td>German_ca</td>
<td>9</td>
<td>13</td>
<td>0</td>
<td>216</td>
<td>0</td>
<td>0.02</td>
</tr>
<tr>
<td>Illinois</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>212</td>
<td>0</td>
<td>0.06</td>
</tr>
</tbody>
</table>

---

4 The experiments were run on a laptop Intel(R) Core(TM) i3 CPU 2.27GHz with 4GB RAM running Linux Ubuntu 12.04.
MCMT: parametrized timed systems

We analyzed parametrised systems where single processes are endowed with clocks. Fourier-Motzkin QE is applied when computing preimages.

Figure: Fischer’s algorithm
MCMT: parametrized timed systems

Figure: CSMA, client and bus automata
MCMT: parametrized timed systems

**MCMT statistics**

<table>
<thead>
<tr>
<th>Problem</th>
<th>depth</th>
<th>#nodes</th>
<th>#SMT calls</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fischer_abd</td>
<td>14</td>
<td>111</td>
<td>5181</td>
<td>3.39</td>
</tr>
<tr>
<td>Fischer_sal</td>
<td>15</td>
<td>56</td>
<td>1186</td>
<td>0.34</td>
</tr>
<tr>
<td>Fischer_sal_buggy</td>
<td>6</td>
<td>16</td>
<td>307</td>
<td>0.08</td>
</tr>
<tr>
<td>Fischer_std</td>
<td>10</td>
<td>16</td>
<td>363</td>
<td>0.08</td>
</tr>
<tr>
<td>Fischer_upp</td>
<td>8</td>
<td>15</td>
<td>327</td>
<td>0.07</td>
</tr>
<tr>
<td>Lynch_mah</td>
<td>17</td>
<td>35</td>
<td>493</td>
<td>0.08</td>
</tr>
<tr>
<td>Lynch_full</td>
<td>25</td>
<td>1103</td>
<td>45554</td>
<td>37.56</td>
</tr>
<tr>
<td>CSMA</td>
<td>4</td>
<td>23</td>
<td>1363</td>
<td>0.61</td>
</tr>
<tr>
<td>CSMA_buggy</td>
<td>7</td>
<td>39</td>
<td>1778</td>
<td>0.90</td>
</tr>
<tr>
<td>tta</td>
<td>7</td>
<td>36</td>
<td>916</td>
<td>1.98</td>
</tr>
<tr>
<td>tta2</td>
<td>8</td>
<td>70</td>
<td>2017</td>
<td>14.00</td>
</tr>
</tbody>
</table>

**Uppaal timings (for increasing N)**

<table>
<thead>
<tr>
<th>Problem</th>
<th>N = 2</th>
<th>N = 5</th>
<th>N = 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fischer_sal</td>
<td>0.01</td>
<td>0.08</td>
<td>37392</td>
</tr>
<tr>
<td>Lynch_mah</td>
<td>0.00</td>
<td>0.05</td>
<td>44.34</td>
</tr>
<tr>
<td>CSMA</td>
<td>0.00</td>
<td>0.18</td>
<td>&gt;10min</td>
</tr>
<tr>
<td>tta2</td>
<td>0.01</td>
<td>0.06</td>
<td>&gt;10min</td>
</tr>
</tbody>
</table>
MCMT: 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.
MCMT: 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.


4. Third modified version: now safe for *Send-Omission* model!
MCMT: 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).

<table>
<thead>
<tr>
<th>Problem</th>
<th>result</th>
<th>depth</th>
<th>#nodes</th>
<th>#deleted</th>
<th>#vars</th>
<th>#SMT calls</th>
<th>#inv.</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash</td>
<td>SAFE</td>
<td>13</td>
<td>113</td>
<td>21</td>
<td>4</td>
<td>1731</td>
<td>0</td>
<td>0.75</td>
</tr>
<tr>
<td>Send_Omission (1)</td>
<td>UNSAFE</td>
<td>12</td>
<td>464</td>
<td>26</td>
<td>3</td>
<td>16253</td>
<td>0</td>
<td>14.16</td>
</tr>
<tr>
<td>Send_Omission (2)</td>
<td>UNSAFE</td>
<td>34</td>
<td>9679</td>
<td>770</td>
<td>6</td>
<td>1118959</td>
<td>0</td>
<td>30m 18.15s</td>
</tr>
<tr>
<td>Send_Omission (3)</td>
<td>SAFE</td>
<td>32</td>
<td>571</td>
<td>72</td>
<td>4</td>
<td>547054</td>
<td>94 (+7)</td>
<td>6m 57.19s</td>
</tr>
</tbody>
</table>
Algorithm 1 Pseudo-code for Algorithms 1, 2, and 3

Initialization:

\[
\begin{align*}
\text{if } (p\text{ is the sender}) & \quad \text{then } estimate_p \leftarrow m; \ coord\_id_p \leftarrow 0; \\
\text{else } estimate_p \leftarrow \bot; \ coord\_id_p \leftarrow -1; \\
state_p & \leftarrow \text{undecided};
\end{align*}
\]

End Initialization

for \( c \leftarrow 1, 2, \ldots \) do \quad // 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
1. Infinite state model-checking

2. Our Declarative Proposal

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

$$\exists i \forall k \psi(i, k, a[i], a[k]),$$

(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

$$\exists i \forall k \psi(i, k, a[i], a[k]),$$

(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

\[ \exists i \bigwedge_{t} \psi(i, t, a[i], a[t]), \]  

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 \( X := i \), 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

$$\exists i \bigwedge_t \psi(i, t, a[i], a[t]),$$

(5)

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

If one take the obvious choice $X := i$, we do not get \textit{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:

```c
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:

```c
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:

\[ p = 2 \land J \neq a_{\text{length}} \land a[J] \neq 0 \]
\[ p = 2 \land J + 1 \neq a_{\text{length}} \land a[J + 1] \neq 0 \land a[J] = 0 \]
\[ \ldots \]
\[ p = 2 \land J + n \neq a_{\text{length}} \land a[J + n] \neq 0 \land \bigwedge_{k=J}^{J+n-1} a[k] = 0 \]
\[ \ldots \]
Indeed backward search trivially diverges here:

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

$$
\exists n > 0 \left( p = 1 \land \forall k \ (l \leq k < l + n \rightarrow k \neq a\_length) \land p' = 1 \land \\
l' = l + n \land J' = J \land a' = \text{wr}(a, [l, l + n - 1], 0) \right);
$$

$$
\exists n > 0 \left( p = 2 \land \forall k \ (J \leq k < J + n \rightarrow k \neq a\_length \land a[k] = 0) \land p' = 2 \land l' = l \land J' = J + n \land a' = a \right).
$$
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 \textit{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

\[
\exists n > 0 \left( p = 1 \land \forall k (l \leq k < l + n \rightarrow k \neq a\_length) \land p' = 1 \land \\
I' = l + n \land J' = J \land a' = \text{wr}(a, [l, l + n - 1], 0) \right); \\
\exists n > 0 \left( p = 2 \land \forall k (J \leq k < J + n \rightarrow k \neq a\_length \land a[k] = 0) \land \\
p' = 2 \land I' = l \land J' = J + n \land a' = a \right).
\]
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 gets 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 gets 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 lose 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 lose 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

The combination with monotonic abstraction with other abstraction is quite powerful: typically, when there are nested loops, monotonic abstraction takes care of inner loops, thus leaving predicate abstraction to care about outer loops only.

Very often, array accelerated transitions gives formulae in an $\exists^* \forall^*$-fragment which is decidable modulo array axioms (Bradley fragment, our flat fragment [TACAS 14], ...). In these cases, when the control flow graph is flat, safety is decidable and it is convenient not to use any abstraction at all.
Monotonic Abstraction for Arrays

The combination with monotonic abstraction with other abstraction is quite powerful: typically, when there are nested loops, monotonic abstraction takes care of inner loops, thus leaving predicate abstraction to care about outer loops only.

Very often, array accelerated transitions gives formulae in an $\exists^*\forall^*$-fragment which is decidable modulo array axioms (Bradley fragment, our flat fragment [TACAS 14], ...). In these cases, when the control flow graph is flat, safety is decidable and it is convenient not to use any abstraction at all.
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**

## BOOSTER: Experiments

<table>
<thead>
<tr>
<th>Filename</th>
<th>Status</th>
<th>ACC+Abs</th>
<th>ABS</th>
<th>ACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_structures/set_multi_proc.c</td>
<td>SAFE</td>
<td>1.600</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>data_structures/set_multi_proc_trivial.c</td>
<td>SAFE</td>
<td>0.208</td>
<td>0.208</td>
<td>0.314</td>
</tr>
<tr>
<td>data_structures/set_multi_proc_unsafe.c</td>
<td>UNSAFE</td>
<td>1.946</td>
<td>1.257</td>
<td>2.102</td>
</tr>
<tr>
<td>sanfoundry/06.c</td>
<td>SAFE</td>
<td>0.016</td>
<td>TO</td>
<td>0.016</td>
</tr>
<tr>
<td>sanfoundry/07.c</td>
<td>SAFE</td>
<td>4.623</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>sanfoundry/08.c</td>
<td>SAFE</td>
<td>2.926</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>sanfoundry/09.c</td>
<td>SAFE</td>
<td>8.447</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>sanfoundry/10.c</td>
<td>SAFE</td>
<td>0.157</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>sanfoundry/24.c</td>
<td>SAFE</td>
<td>0.101</td>
<td>0.071</td>
<td>0.085</td>
</tr>
<tr>
<td>sanfoundry/27.c</td>
<td>SAFE</td>
<td>0.066</td>
<td>0.076</td>
<td>108.724</td>
</tr>
<tr>
<td>sanfoundry/28.c</td>
<td>SAFE</td>
<td>0.676</td>
<td>0.151</td>
<td>63.932</td>
</tr>
<tr>
<td>sanfoundry/39.c</td>
<td>SAFE</td>
<td>1.832</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>sorting/bubblesort.c</td>
<td>SAFE</td>
<td>0.233</td>
<td>0.107</td>
<td>0.407</td>
</tr>
<tr>
<td>sorting/bubblesort_unsafe.c</td>
<td>UNSAFE</td>
<td>0.090</td>
<td>0.090</td>
<td>0.135</td>
</tr>
<tr>
<td>sorting/selecionsort.c</td>
<td>SAFE</td>
<td>85.326</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>standard/allDiff_safe.c</td>
<td>SAFE</td>
<td>0.010</td>
<td>0.044</td>
<td>0.010</td>
</tr>
<tr>
<td>standard/allDiff_unsafe.c</td>
<td>UNSAFE</td>
<td>0.007</td>
<td>0.036</td>
<td>0.006</td>
</tr>
<tr>
<td>svcomp/loops/array_false-unreach-label.c</td>
<td>UNSAFE</td>
<td>0.135</td>
<td>0.039</td>
<td>0.094</td>
</tr>
<tr>
<td>svcomp/loops/array_true-unreach-label.c</td>
<td>SAFE</td>
<td>0.169</td>
<td>0.057</td>
<td>TO</td>
</tr>
<tr>
<td>svcomp/loops/compact_false-unreach-label.c</td>
<td>UNSAFE</td>
<td>0.010</td>
<td>0.051</td>
<td>0.010</td>
</tr>
<tr>
<td>svcomp/loops/heavy_false-unreach-label.c</td>
<td>SAFE</td>
<td>0.363</td>
<td>0.277</td>
<td>TO</td>
</tr>
<tr>
<td>svcomp/loops/heavy_true-unreach-label.c</td>
<td>UNSAFE</td>
<td>0.296</td>
<td>0.217</td>
<td>0.393</td>
</tr>
<tr>
<td>svcomp/loops/linear_search_false-unreach-label.c</td>
<td>UNSAFE</td>
<td>0.154</td>
<td>0.053</td>
<td>0.062</td>
</tr>
<tr>
<td>svcomp/loops/linear_search_true-unreach-label.c</td>
<td>SAFE</td>
<td>0.016</td>
<td>0.101</td>
<td>TO</td>
</tr>
<tr>
<td>svcomp/loops/nec11_false-unreach-label.c</td>
<td>UNSAFE</td>
<td>0.053</td>
<td>0.040</td>
<td>0.75</td>
</tr>
<tr>
<td>svcomp/loops/nec40_true-unreach-label.c</td>
<td>SAFE</td>
<td>0.010</td>
<td>0.607</td>
<td>0.16</td>
</tr>
<tr>
<td>svcomp/loops/string_true-unreach-label.c</td>
<td>SAFE</td>
<td>0.860</td>
<td>0.781</td>
<td>1.04</td>
</tr>
<tr>
<td>svcomp/loops/sum_array_false-unreach-label.c</td>
<td>UNSAFE</td>
<td>0.068</td>
<td>0.059</td>
<td>0.104</td>
</tr>
<tr>
<td>svcomp/loops/sum_array_true-unreach-label.c</td>
<td>SAFE</td>
<td>0.070</td>
<td>0.080</td>
<td>TO</td>
</tr>
</tbody>
</table>
## BOOSTER: Comparisons (?)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>COMPASS</th>
<th>Z3 HORN</th>
<th>ARMC</th>
<th>DUALITY</th>
<th>BOOSTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>0.01</td>
<td>0.06</td>
<td>0.15</td>
<td>0.72</td>
<td>0.01</td>
</tr>
<tr>
<td>init_non_constant</td>
<td>0.02</td>
<td>0.08</td>
<td>0.48</td>
<td>6.60</td>
<td>0.01</td>
</tr>
<tr>
<td>init_partial</td>
<td>0.01</td>
<td>0.03</td>
<td>0.14</td>
<td>2.60</td>
<td>0.01</td>
</tr>
<tr>
<td>init_partial_buggy</td>
<td>0.02</td>
<td>0.01</td>
<td>0.07</td>
<td>0.03</td>
<td>0.01</td>
</tr>
<tr>
<td>init_even</td>
<td>0.04</td>
<td>TO</td>
<td>-</td>
<td>TO</td>
<td>0.02</td>
</tr>
<tr>
<td>init_even_buggy</td>
<td>0.04</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>0.01</td>
</tr>
<tr>
<td>copy</td>
<td>0.01</td>
<td>0.04</td>
<td>0.20</td>
<td>1.40</td>
<td>0.01</td>
</tr>
<tr>
<td>copy_partial</td>
<td>0.01</td>
<td>0.04</td>
<td>0.21</td>
<td>1.80</td>
<td>0.01</td>
</tr>
<tr>
<td>copy_odd</td>
<td>0.04</td>
<td>TO</td>
<td>-</td>
<td>4.50</td>
<td>TO</td>
</tr>
<tr>
<td>copy_odd_buggy</td>
<td>0.05</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>0.07</td>
</tr>
<tr>
<td>reverse</td>
<td>0.03</td>
<td>0.12</td>
<td>2.28</td>
<td>8.50</td>
<td>0.02</td>
</tr>
<tr>
<td>reverse_buggy</td>
<td>0.04</td>
<td>0.01</td>
<td>0.08</td>
<td>0.03</td>
<td>0.01</td>
</tr>
<tr>
<td>swap</td>
<td>0.12</td>
<td>0.41</td>
<td>3.0</td>
<td>40.60</td>
<td>0.12</td>
</tr>
<tr>
<td>swap_buggy</td>
<td>0.11</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>0.03</td>
</tr>
<tr>
<td>double_swap</td>
<td>0.16</td>
<td>1.37</td>
<td>4.4</td>
<td>TO</td>
<td>0.34</td>
</tr>
<tr>
<td>check_swap</td>
<td>0.07</td>
<td>0.05</td>
<td>0.15</td>
<td>0.62</td>
<td>0.02</td>
</tr>
<tr>
<td>check_memcpy</td>
<td>0.04</td>
<td>0.04</td>
<td>0.20</td>
<td>16.30</td>
<td>0.02</td>
</tr>
<tr>
<td>find</td>
<td>0.02</td>
<td>0.01</td>
<td>0.08</td>
<td>0.38</td>
<td>0.26</td>
</tr>
<tr>
<td>find_first_nonnull</td>
<td>0.02</td>
<td>0.01</td>
<td>0.08</td>
<td>0.39</td>
<td>0.09</td>
</tr>
<tr>
<td>array_append</td>
<td>0.02</td>
<td>0.04</td>
<td>1.76</td>
<td>1.50</td>
<td>0.02</td>
</tr>
<tr>
<td>merge_interleave</td>
<td>0.09</td>
<td>0.04</td>
<td>-</td>
<td>1.50</td>
<td>0.15</td>
</tr>
<tr>
<td>merge_interleave_buggy</td>
<td>0.11</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>0.01</td>
</tr>
</tbody>
</table>
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 (e.g., 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 (e.g., 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 (e.g., 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 (e.g., 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 (e.g., 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.