SMT Model Checking of Array-Based Systems

S. Ghilardi\textsuperscript{1} and S. Ranise\textsuperscript{2}

\textsuperscript{1}Università degli Studi di Milano

\textsuperscript{2}FBK -Trento

SMT Workshop, Edinburgh, July 15, 2010
Aim of the talk

My talk will report recent experience (ranging from theoretical foundations to implementation) 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 (ranging from theoretical foundations to implementation) 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 (ranging from theoretical foundations to implementation) 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 (ranging from theoretical foundations to implementation) 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 (ranging from theoretical foundations to implementation) 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 (ranging from theoretical foundations to implementation) 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 (ranging from theoretical foundations to implementation) 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 (ranging from theoretical foundations to implementation) 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 (ranging from theoretical foundations to implementation) 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. Experiments and case studies
Outline

1. Infinite state model-checking
2. Our Declarative Proposal
3. The tool MCMT
4. Experiments and case studies
Outline

1. Infinite state model-checking
2. Our Declarative Proposal
3. The tool MCMT
4. Experiments and case studies
Outline

1. Infinite state model-checking
2. Our Declarative Proposal
3. The tool MCMT
4. Experiments and case studies
1. Infinite state model-checking

2. Our Declarative Proposal

3. The tool MCMT

4. Experiments and case studies
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
Infinite state model-checking

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:

\[ s_1 \preceq s_2 \]
\[ \downarrow_\lambda \]
\[ s_3 \preceq \exists \]
\[ \downarrow_\lambda \]
\[ s_4 \]

---

\(^1\)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:

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

\(^1\) 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:

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

\(^1\)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;
- **\(\tau = \{\rightarrow_\lambda \subseteq S \times S\}\)**: labelled directed graph;
- **\(\preceq\)**: well quasi ordering\(^1\) (wqo) on \(S\);
- each \(\tau_\lambda\) is monotonic:

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

\(^1\)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:

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

---

\(^1\)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;
- \(\tau = \{\rightarrow_\lambda \subset S \times S\}_\lambda\): labelled directed graph;
- \(\preceq\): well quasi ordering\(^1\) (wqo) on \(S\);
- each \(\tau_\lambda\) is monotonic:

\[
\begin{array}{ccc}
S_1 & \preceq & S_2 \\
\downarrow_\lambda & & \downarrow_\lambda \\
S_3 & \preceq & S_4
\end{array}
\]

\(^1\)Reflexive, transitive binary relation that neither contains infinite strictly decreasing sequences nor infinite sequences of pairwise incomparable elements.
Well-Structured Transition Systems

- 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

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

- 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

```
function BReach(K)
    $i \leftarrow 0; BR^0(\tau, K) \leftarrow K; K^0 \leftarrow K$
    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
end
```
Backward Reachability

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

```
function BReach(K)
    i ← 0; BR^0(τ, K) ← K; K^0 ← K
    if BR^0(τ, K) ∩ I ≠ ∅ 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) ∩ I ≠ ∅ then return unsafe
        else i ← i + 1
    until BR^{i+1}(τ, K) ⊆ BR^i(τ, K)
    return safe
end
```
Termination and Empirical Success

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

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.
Termination and Empirical Success

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

**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.
Termination and Empirical Success

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

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.
Termination and Empirical Success

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

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.
1. Infinite state model-checking

2. Our Declarative Proposal

3. The tool MCMT

4. Experiments and case studies
Array-based Systems

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 $T$).

- **Topology** of the parameterised system: theory $T_I = (\Sigma_I, \mathcal{C}_I)$
  - E.g.: $\mathcal{C}_I$ consists of all (finite) sets, linear orders, forests/trees, graphs, ...

- **Data** manipulated by the parameterised system: theories $T_E = (\Sigma_E, \mathcal{C}_E)$
  - Usually $\mathcal{C}_E$ contains just one structure: integers, reals, Booleans, control locations, ...

- We assume the availability of SMT solvers deciding the satisfiability of quantifier-free formulae modulo $T_I$ and $T_E$. We limit ourselves to a single data theory $T_E$ and to systems with only one array variable (this limitation is for simplifying notation only).
Array-based Systems

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 $T$).

- **Topology** of the parameterised system: theory $T_I = (\Sigma_I, C_I)$
  E.g.: $C_I$ consists of all (finite) sets, linear orders, forests/trees, graphs, ...

- **Data** manipulated by the parameterised system: theories $T_E = (\Sigma_{E_i}, C_{E_i})$
  Usually $C_{E_i}$ contains just one structure: integers, reals, Booleans, control locations, ...

- We assume the availability of SMT solvers deciding the satisfiability of quantifier-free formulae modulo $T_I$ and $T_{E_i}$.
  We limit ourselves to a single data theory $T_E$ and to systems with only one array variable (this limitation is for simplifying notation only).
Array-based Systems

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 $T$).

- **Topology** of the parameterised system: theory $T_I = (\Sigma_I, C_I)$
  E.g.: $C_I$ consists of all (finite) sets, linear orders, forests/trees, graphs, ...

- **Data** manipulated by the parameterised system: theories $T_{E_i} = (\Sigma_{E_i}, C_{E_i})$
  Usually $C_{E_i}$ contains just one structure: integers, reals, Booleans, control locations, ...

- We assume the availability of SMT solvers deciding the satisfiability of quantifier-free formulae modulo $T_I$ and $T_{E_i}$.
  We limit ourselves to a single data theory $T_{E_i}$ and to systems with only one array variable (this limitation is for simplifying notation only).
Array-based Systems

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 \( T \)).

- **Topology** of the parameterised system: theory \( T_I = (\Sigma_I, C_I) \)
  - E.g.: \( C_I \) consists of all (finite) sets, linear orders, forests/trees, graphs, ...

- **Data** manipulated by the parameterised system: theories \( T_E = (\Sigma_E, C_E) \)
  - Usually \( C_E \) contains just one structure: integers, reals, Booleans, control locations, ...

- We assume the availability of SMT solvers deciding the satisfiability of quantifier-free formulae modulo \( T_I \) and \( T_{E_I} \).
  - We limit ourselves to a single data theory \( T_E \) and to systems with only one array variable (this limitation is for simplifying notation only).
Array-Based Systems

- the sort \( \text{INDEX} \) is constrained by \( T_I \);
- the sort \( \text{ELEM} \) is constrained by \( T_E \);
- the sort \( \text{ARRAY} \) represents arrays of \( \text{ELEM} \) defined on \( \text{INDEX} \);
- the ‘read’ operation \( _{-}[_] \) is added to \( \Sigma_I \cup \Sigma_E \);
- the class of models of \( A^F_I \) consists of the three-sorted structures whose reducts are models of \( T_I, T_E \) and the sort \( \text{ARRAY} \) 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 $A^E_i$ 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 $A^E_j$ 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 $A^E_j$.

- A safety problem for $S$ is the following: given a formula $K(a)$, is $A^E_j$-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 $A_I^E$ with array state variable $a$ is the following pair of formulae:
  \[ S = \langle l(a), \tau(a, a') \rangle. \]

- 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 $A_I^E$-satisfiable for some $n$?
  \[ l(a_0) \land \tau(a_0, a_1) \land \cdots \land \tau(a_{n-1}, a_n) \land K(a_n) \]
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) ∩ I ≠ ∅ 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) ∩ I ≠ ∅ 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 \( \text{ELEM} \).
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 \( \text{ELEM} \).
Format for initialization formulae

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 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

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

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

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

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

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

can be eliminated by syntactic transformations by adopting the stopping failures model (also known as the ‘approximate model’ in the model checking literature). Safety certifications for the stopping failures model are stronger than for the original model.

The stopping failure model is the highest failure model used to formalize fault tolerant systems.

When universal guards are met, the stopping failure model is automatically adopted by our tool and a warning in this sense is displayed to inform the user.
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 syntactic transformations by adopting the stopping failures model (also known as the ‘approximate model’ in the model checking literature). Safety certifications for the stopping failures model are stronger than for the original model.

The stopping failure model is the lighest failure model used to formalize fault tolerant systems.

When universal guards are met, the stopping failure model is automatically adopted by our tool and a warning in this sense is displayed to inform the user.
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 syntactic transformations by adopting the stopping failures model (also known as the ‘approximate model’ in the model checking literature). Safety certifications for the stopping failures model are stronger than for the original model.

The stopping failure model is the highest failure model used to formalize fault tolerant systems.

When universal guards are met, the stopping failure model is automatically adopted by our tool and a warning in this sense is displayed to inform the user.
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 syntactic transformations by adopting the stopping failures model (also known as the ‘approximate model’ in the model checking literature). Safety certifications for the stopping failures model are stronger than for the original model.

The stopping failure model is the highest failure model used to formalize fault tolerant systems.

When universal guards are met, the stopping failure model is automatically adopted by our tool and a warning in this sense is displayed to inform the user.
Closure under pre-image

The following result guarantees that backward reachability can only produce $\exists^I$-formulae:

**Theorem**

If $H(a)$ is an $\exists^I$-formula, the formula

$$\text{Pre}(\tau, H) := \exists a' (\tau(a, a') \land H(a'))$$

is $A^E_T$-equivalent to an effectively computable $\exists^I$-formula.

Proof is easy (except for the extension with serial updates).
Closure under pre-image

The following result guarantees that backward reachability can only produce $\exists^l$-formulae:

**Theorem**

*If $H(a)$ is an $\exists^l$-formula, the formula*

$$Pre(\tau, H) := \exists a' (\tau(a, a') \land H(a'))$$

*is $A^E_I$-equivalent to an effectively computable $\exists^l$-formula.*

Proof is easy (except for the extension with serial updates).
Closure under pre-image

The following result guarantees that backward reachability can only produce $\exists^I$-formulae:

**Theorem**

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.

Proof is easy (except for the extension with serial updates).
SMT problems for safety and fix-point checking

The following result guarantees effectiveness of satisfiability for safety and fix-point checking during backward reachability

**Theorem**

Suppose that $\Sigma_I$ does not contain function symbols\(^a\) and that $\mathcal{C}_I$ is closed under substructures. Then the formulae of the kind

$\exists i \forall j \psi(i, j, a[i], a[j])$ \hspace{1cm} (4)

(where $\psi$ is a quantifier-free $\Sigma_I \cup \Sigma_E$-formula) are decidable for $A^E_I$-satisfiability.

\(^a\)More generally, that $T_I$ is locally finite.
The following result guarantees effectiveness of satisfiability for safety and fix-point checking during backward reachability.

**Theorem**

Suppose that $\Sigma_I$ does not contain function symbols\(^a\) and that $C_I$ is closed under substructures. Then the formulae of the kind

\[ \exists i \forall j \psi(i, j, a[i], a[j]) \]  \hspace{1cm} (4)

(assuming $\psi$ is a quantifier-free $\Sigma_I \cup \Sigma_E$-formula) are decidable for $A^E_I$-satisfiability.

\(^a\)More generally, that $T_I$ is locally finite.
The two assumptions on $T_I$ are both indispensable (undecidability arises otherwise); they concern the ‘topology’ of the parameterised system.

Closure under substructures means that you can ‘delete’ processes while maintaining the topology; e.g., deleting processes from a finite set, linear order, graph, forest, one still gets a finite set, linear order, graph, forest (but this is not true for rings).

The proof of the above theorem is by quantifier instantiation, followed by purification and standard combination techniques.
Our Declarative Proposal

SMT problems for safety and fix-point checking

- The two assumptions on $T_I$ are both indispensable (undecidability arises otherwise); they concern the ‘topology’ of the parameterised system.
- **Closure under substructures** means that you can ‘delete’ processes while maintaining the topology; e.g., deleting processes from a finite set, linear order, graph, forest, one still gets a finite set, linear order, graph, forest (but this is not true for rings).
- The proof of the above theorem is by quantifier instantiation, followed by purification and standard combination techniques.
Our Declarative Proposal

SMT problems for safety and fix-point checking

- The two assumptions on $T$ are both indispensable (undecidability arises otherwise); they concern the ‘topology’ of the parameterised system.

- **Closure under substructures** means that you can ‘delete’ processes while maintaining the topology; e.g., deleting processes from a finite set, linear order, graph, forest, one still gets a finite set, linear order, graph, forest (but this is not true for rings).

- The proof of the above theorem is by **quantifier instantiation**, followed by purification and standard combination techniques.
Termination issues

- A configuration is a pair \((s, M)\) such that \(s\) is an array of a finite index model \(M\) of \(A^E_I\).

- Given \(s\), the \(\Sigma_I\)-structure \(s_I\) is the \(\Sigma_I\)-reduct of \(M\) and the \(\Sigma_E\)-structure \(s_E\) is the smallest \(\Sigma_E\)-substructure of the \(\Sigma_E\)-reduct of \(M\) containing the image of \(s\).

- Let \(s, s'\) be configurations, we define a preorder as follows: \(s' \preceq s\) holds iff there exists
  - a \(\Sigma_I\)-embedding \(\mu : s'_I \rightarrow s_I\) and
  - a \(\Sigma_E\)-embedding \(\nu : s'_E \rightarrow s_E\) such that
  - the set-theoretical compositions of \(\mu\) with \(s\) and of \(s'\) with \(\nu\) are equal.

Theorem

If \(T_E\) is locally finite and \(\preceq\) is a wqo, backward search terminates.
Termination issues

- A configuration is a pair \((s, \mathcal{M})\) such that \(s\) is an array of a finite index model \(\mathcal{M}\) of \(A_I^E\).
- Given \(s\), the \(\Sigma_I\)-structure \(s_I\) is the \(\Sigma_I\)-reduct of \(\mathcal{M}\) and the \(\Sigma_E\)-structure \(s_E\) is the smallest \(\Sigma_E\)-substructure of the \(\Sigma_E\)-reduct of \(\mathcal{M}\) containing the image of \(s\).
- Let \(s, s'\) be configurations, we define a preorder as follows: \(s' \preceq s\) holds iff there exists
  - a \(\Sigma_I\)-embedding \(\mu : s'_I \rightarrow s_I\) and
  - a \(\Sigma_E\)-embedding \(\nu : s'_E \rightarrow s_E\) such that
  - the set-theoretical compositions of \(\mu\) with \(s\) and of \(s'\) with \(\nu\) are equal.

Theorem

If \(T_E\) is locally finite and \(\preceq\) is a wqo, backward search terminates.
Termination issues

- A configuration is a pair \((s, \mathcal{M})\) such that \(s\) is an array of a finite index model \(\mathcal{M}\) of \(A_I^E\).
- Given \(s\), the \(\Sigma_I\)-structure \(s_I\) is the \(\Sigma_I\)-reduct of \(\mathcal{M}\) and the \(\Sigma_E\)-structure \(s_E\) is the smallest \(\Sigma_E\)-substructure of the \(\Sigma_E\)-reduct of \(\mathcal{M}\) containing the image of \(s\).
- Let \(s, s'\) be configurations, we define a preorder as follows: \(s' \preceq s\) holds iff there exists
  - a \(\Sigma_I\)-embedding \(\mu : s'_I \rightarrow s_I\) and
  - a \(\Sigma_E\)-embedding \(\nu : s'_E \rightarrow s_E\) such that
  - the set-theoretical compositions of \(\mu\) with \(s\) and of \(s'\) with \(\nu\) are equal.

Theorem

If \(T_E\) is locally finite and \(\preceq\) is a wqo, backward search terminates.
Termination issues

- A configuration is a pair \((s, \mathcal{M})\) such that \(s\) is an array of a finite index model \(\mathcal{M}\) of \(A^E_I\).
- Given \(s\), the \(\Sigma_I\)-structure \(s_I\) is the \(\Sigma_I\)-reduct of \(\mathcal{M}\) and the \(\Sigma_E\)-structure \(s_E\) is the smallest \(\Sigma_E\)-substructure of the \(\Sigma_E\)-reduct of \(\mathcal{M}\) containing the image of \(s\).
- Let \(s, s'\) be configurations, we define a preorder as follows: \(s' \preceq s\) holds iff there exists
  - a \(\Sigma_I\)-embedding \(\mu : s'_I \rightarrow s_I\) and
  - a \(\Sigma_E\)-embedding \(\nu : s'_E \rightarrow s_E\) such that
  - the set-theoretical compositions of \(\mu\) with \(s\) and of \(s'\) with \(\nu\) are equal.

**Theorem**

If \(T_E\) is locally finite and \(\preceq\) is a wqo, backward search terminates.
Termination issues

- A configuration is a pair \((s, M)\) such that \(s\) is an array of a finite index model \(M\) of \(A^E_I\).
- Given \(s\), the \(\Sigma_I\)-structure \(s_I\) is the \(\Sigma_I\)-reduct of \(M\) and the \(\Sigma_E\)-structure \(s_E\) is the smallest \(\Sigma_E\)-substructure of the \(\Sigma_E\)-reduct of \(M\) containing the image of \(s\).
- Let \(s, s'\) be configurations, we define a preorder as follows: \(s' \preceq s\) holds iff there exists \(s' \preceq s\) such that
  - a \(\Sigma_I\)-embedding \(\mu : s'_I \rightarrow s_I\) and
  - a \(\Sigma_E\)-embedding \(\nu : s'_E \rightarrow s_E\) such that
  - the set-theoretical compositions of \(\mu\) with \(s\) and of \(s'\) with \(\nu\) are equal.

Theorem

If \(T_E\) is locally finite and \(\preceq\) is a wqo, backward search terminates.
Termination issues

- A configuration is a pair \((s, \mathcal{M})\) such that \(s\) is an array of a finite index model \(\mathcal{M}\) of \(A^E_I\).

- Given \(s\), the \(\Sigma_I\)-structure \(s_I\) is the \(\Sigma_I\)-reduct of \(\mathcal{M}\) and the \(\Sigma_E\)-structure \(s_E\) is the smallest \(\Sigma_E\)-substructure of the \(\Sigma_E\)-reduct of \(\mathcal{M}\) containing the image of \(s\).

- Let \(s, s'\) be configurations, we define a preorder as follows: \(s' \preceq s\) holds iff there exists \(s' \preceq s\) holds iff there exists
  - a \(\Sigma_I\)-embedding \(\mu : s'_I \rightarrow s_I\) and
  - a \(\Sigma_E\)-embedding \(\nu : s'_E \rightarrow s_E\) such that
  - the set-theoretical compositions of \(\mu\) with \(s\) and of \(s'\) with \(\nu\) are equal.

**Theorem**

If \(T_E\) is locally finite and \(\preceq\) is a wqo, backward search terminates.
Termination issues

- A configuration is a pair \((s, \mathcal{M})\) such that \(s\) is an array of a finite index model \(\mathcal{M}\) of \(A^E_I\).
- Given \(s\), the \(\Sigma_I\)-structure \(s_I\) is the \(\Sigma_I\)-reduct of \(\mathcal{M}\) and the \(\Sigma_E\)-structure \(s_E\) is the smallest \(\Sigma_E\)-substructure of the \(\Sigma_E\)-reduct of \(\mathcal{M}\) containing the image of \(s\).
- Let \(s, s'\) be configurations, we define a preorder as follows:
  \(s' \preceq s\) holds iff there exists
  - a \(\Sigma_I\)-embedding \(\mu : s'_I \rightarrow s_I\) and
  - a \(\Sigma_E\)-embedding \(\nu : s'_E \rightarrow s_E\) such that
  - the set-theoretical compositions of \(\mu\) with \(s\) and of \(s'\) with \(\nu\) are equal.

**Theorem**

*If \(T_E\) is locally finite and \(\preceq\) is a wqo, backward search terminates.*
Invariant Synthesis

An inductive invariant is a $\forall^I$-formula $H(a)$ s.t.

(i) $A^E_I \models I \rightarrow H$;

(ii) $A^E_I \models H(a) \land \tau(a, a') \rightarrow H(a')$;

(iii) $H(a)$ is $A^E_I$-inconsistent with the formula $K(a)$ describing unsafe states.

Thanks to Theorem 2, conditions (i)-(ii)-(iii) are decidable (even for transitions with universal guards, without using the stopping failures model).

Invariant checking and invariant synthesis are common deductive techniques in formal methods. We give here a method for invariant synthesis specifically tailored to our array-based systems. It exploits the configuration ordering.
Invariant Synthesis

An inductive invariant is a $\forall^I$-formula $H(a)$ s.t.

(i) $\models A^E_i \models I \rightarrow H$;

(ii) $\models A^E_i \models H(a) \land \tau(a, a') \rightarrow H(a')$;

(iii) $H(a)$ is $A^E_i$-inconsistent with the formula $K(a)$ describing unsafe states.

Thanks to Theorem 2, conditions (i)-(ii)-(iii) are decidable (even for transitions with universal guards, without using the stopping failures model).

Invariant checking and invariant synthesis are common deductive techniques in formal methods. We give here a method for invariant synthesis specifically tailored to our array-based systems. It exploits the configuration ordering.
Invariant Synthesis

An **inductive invariant** is a \( \forall^I \)-formula \( H(a) \) s.t.

(i) \( A^E \models I \rightarrow H \);

(ii) \( A^E \models H(a) \land \tau(a, a') \rightarrow H(a') \);

(iii) \( H(a) \) is \( A^E \)-inconsistent with the formula \( K(a) \) describing unsafe states.

Thanks to Theorem 2, conditions (i)-(ii)-(iii) are decidable (even for transitions with universal guards, without using the stopping failures model).

Invariant checking and invariant synthesis are common **deductive** techniques in formal methods. We give here a method for invariant synthesis specifically tailored to our array-based systems. It exploits the configuration ordering.
Invariant Synthesis

An inductive invariant is a \( \forall^I \)-formula \( H(a) \) s.t.

(i) \( A^E_{\bar{a}} \models I \rightarrow H; \)

(ii) \( A^E_{\bar{a}} \models H(a) \land \tau(a, a') \rightarrow H(a'); \)

(iii) \( H(a) \) is \( A^E_{\bar{a}} \)-inconsistent with the formula \( K(a) \) describing unsafe states.

Thanks to Theorem 2, conditions (i)-(ii)-(iii) are decidable (even for transitions with universal guards, without using the stopping failures model).

Invariant checking and invariant synthesis are common deductive techniques in formal methods. We give here a method for invariant synthesis specifically tailored to our array-based systems. It exploits the configuration ordering.
Invariant Synthesis

An **inductive invariant** is a $\forall^I$-formula $H(a)$ s.t.

(i) $A^E_I \models I \rightarrow H$;

(ii) $A^E_I \models H(a) \land \tau(a, a') \rightarrow H(a')$;

(iii) $H(a)$ is $A^E_I$-inconsistent with the formula $K(a)$ describing unsafe states.

Thanks to Theorem 2, conditions (i)-(ii)-(iii) are decidable (even for transitions with universal guards, without using the stopping failures model).

Invariant checking and invariant synthesis are common *deductive* techniques in formal methods. We give here a method for invariant synthesis specifically tailored to our array-based systems. It exploits the configuration ordering.
Invariant Synthesis

An inductive invariant is a $\forall^I$-formula $H(a)$ s.t.

(i) $A^E_I \models I \rightarrow H$;

(ii) $A^E_I \models H(a) \land \tau(a, a') \rightarrow H(a')$;

(iii) $H(a)$ is $A^E_I$-inconsistent with the formula $K(a)$ describing unsafe states.

Thanks to Theorem 2, conditions (i)-(ii)-(iii) are decidable (even for transitions with universal guards, without using the stopping failures model).

Invariant checking and invariant synthesis are common deductive techniques in formal methods. We give here a method for invariant synthesis specifically tailored to our array-based systems. It exploits the configuration ordering.
Invariant Synthesis

Suppose that $T_E$ is locally finite. Then it is possible to associate to each configuration $s$ an $\exists I'$-formula $K_s(a)$ (the Robinson diagram of $s$) such that $s' \models K_s$ iff $s \leq s'$.

A predecessor of $s$ is any $s'$ that is $\leq$-minimal in the set of configurations satisfying $Pre(\tau, K_s)$.

Let $s, s'$ be configurations: $s$ is sub-reachable from $s'$ iff there exist configurations $s_0, \ldots, s_n$ such that (i) $s_0 = s$, (ii) $s_n = s'$, and (iii) either $s_{i-1} \leq s_i$ or $s_{i-1}$ is a predecessor of $s_i$, for each $i = 1, \ldots, n$.

If $K$ is an $\exists I'$-formula, $s$ is sub-reachable from $K$ iff $s$ is sub-reachable from some minimal $s'$ satisfying $K$. 
Invariant Synthesis

Suppose that $T_E$ is locally finite. Then it is possible to associate to each configuration $s$ an $\exists I$-formula $K_s(a)$ (the Robinson diagram of $s$) such that $s' \models K_s$ iff $s \leq s'$.

A predecessor of $s$ is any $s'$ that is $\leq$-minimal in the set of configurations satisfying $\text{Pre}(\tau, K_s)$.

Let $s, s'$ be configurations: $s$ is sub-reachable from $s'$ iff there exist configurations $s_0, \ldots, s_n$ such that (i) $s_0 = s$, (ii) $s_n = s'$, and (iii) either $s_{i-1} \leq s_i$ or $s_{i-1}$ is a predecessor of $s_i$, for each $i = 1, \ldots, n$.

If $K$ is an $\exists I$-formula, $s$ is sub-reachable from $K$ iff $s$ is sub-reachable from some minimal $s'$ satisfying $K$. 
Invariant Synthesis

Suppose that $T_E$ is locally finite. Then it is possible to associate to each configuration $s$ an $\exists^I$-formula $K_s(a)$ (the Robinson diagram of $s$) such that $s' \models K_s$ iff $s \leq s'$.

A predecessor of $s$ is any $s'$ that is $\leq$-minimal in the set of configurations satisfying $Pre(\tau, K_s)$.

Let $s, s'$ be configurations: $s$ is sub-reachable from $s'$ iff there exist configurations $s_0, \ldots, s_n$ such that (i) $s_0 = s$, (ii) $s_n = s'$, and (iii) either $s_{i-1} \leq s_i$ or $s_{i-1}$ is a predecessor of $s_i$, for each $i = 1, \ldots, n$.

If $K$ is an $\exists^I$-formula, $s$ is sub-reachable from $K$ iff $s$ is sub-reachable from some minimal $s'$ satisfying $K$. 
Invariant Synthesis

Suppose that $T_E$ is locally finite. Then it is possible to associate to each configuration $s$ an $\exists I$-formula $K_s(a)$ (the Robinson diagram of $s$) such that $s' \models K_s$ iff $s \leq s'$.

A predecessor of $s$ is any $s'$ that is $\leq$-minimal in the set of configurations satisfying $Pre(\tau, K_s)$.

Let $s, s'$ be configurations: $s$ is sub-reachable from $s'$ iff there exist configurations $s_0, \ldots, s_n$ such that (i) $s_0 = s$, (ii) $s_n = s'$, and (iii) either $s_{i-1} \leq s_i$ or $s_{i-1}$ is a predecessor of $s_i$, for each $i = 1, \ldots, n$.

If $K$ is an $\exists I$-formula, $s$ is sub-reachable from $K$ iff $s$ is sub-reachable from some minimal $s'$ satisfying $K$. 
Invariant Synthesis

**Theorem**

*Suppose that* $T_E$ *is locally finite. If there exists an inductive invariant, then there are finitely many configurations* $s_1, \ldots, s_k$ *which are sub-reachable from the unsafety formula* $K$ *such that* $\neg (K_{s_1} \lor \cdots \lor K_{s_k})$ *is also an inductive invariant.*

The meaning of the above theorem is that an inductive invariant can always be found - provided it exists - by an algorithm alternating backward search steps and nondeterministic guessings of subconfigurations. The Theorem has an equivalent declarative formulation (not mentioning configurations at all).
Invariant Synthesis

Theorem

Suppose that $T_E$ is locally finite. If there exists an inductive invariant, then there are finitely many configurations $s_1, \ldots, s_k$ which are sub-reachable from the unsafety formula $K$ such that $\neg(K_{s_1} \lor \cdots \lor K_{s_k})$ is also an inductive invariant.

The meaning of the above theorem is that an inductive invariant can always be found - provided it exists - by an algorithm alternating backward search steps and nondeterministic guessings of subconfigurations. The Theorem has an equivalent declarative formulation (not mentioning configurations at all).
Invariant Synthesis

Due to large use of non-determinism, a full implementation of the above results does not look to be promising.

The idea implemented in our tool is to use abstraction heuristics (index abstraction, signature abstraction, etc.) to guess trace invariants (i.e. properties of reachable states) during backward search.

The tool then makes a copy of itself, checks such trace invariants, and in case it validates them, it uses them during the main parent verification task.

This strategy is sometimes quite powerful (can prune search considerably and even avoid divergence). In a sense, it is emblematic about the flexibility of the deductive SMT-approach to model-checking and of its potentialities to incorporate disparate technologies (not only invariant synthesis, but - we believe - also predicate abstraction, acceleration, widening, etc.).

\(^3\)From now on, ‘invariant’ means ‘trace invariant’.
Invariant Synthesis

Due to large use of non-determinism, a full implementation of the above results does not look to be promising.

The idea implemented in our tool is to use abstraction heuristics (index abstraction, signature abstraction, etc.) to guess trace invariants (i.e. properties of reachable states)\(^3\) during backward search.

The tool then makes a copy of itself, checks such trace invariants, and in case it validates them, it uses then during the main parent verification task.

This strategy is sometimes quite powerful (can prune search considerably and even avoid divergence). In a sense, it is emblematic about the flexibility of the deductive SMT-approach to model-checking and of its potentialities to incorporate disparate technologies (not only invariant synthesis, but - we believe - also predicate abstraction, acceleration, widening, etc.).

\(^3\)From now on, ‘invariant’ means ‘trace invariant’.
Invariant Synthesis

Due to large use of non-determinism, a full implementation of the above results does not look to be promising.

The idea implemented in our tool is to use abstraction heuristics (index abstraction, signature abstraction, etc.) to guess trace invariants (i.e. properties of reachable states)\(^3\) during backward search.

The tool then makes a copy of itself, checks such trace invariants, and in case it validates them, it uses then during the main parent verification task.

This strategy is sometimes quite powerful (can prune search considerably and even avoid divergence). In a sense, it is emblematic about the flexibility of the deductive SMT-approach to model-checking and of its potentialities to incorporate disparate technologies (not only invariant synthesis, but - we believe - also predicate abstraction, acceleration, widening, etc.).

\(^3\)From now on, ‘invariant’ means ‘trace invariant’.
Invariant Synthesis

Due to large use of non-determinism, a full implementation of the above results does not look to be promising.

The idea implemented in our tool is to use abstraction heuristics (index abstraction, signature abstraction, etc.) to guess trace invariants (i.e. properties of reachable states)\(^3\) during backward search.

The tool then makes a copy of itself, checks such trace invariants, and in case it validates them, it uses them during the main parent verification task.

This strategy is sometimes quite powerful (can prune search considerably and even avoid divergence). In a sense, it is emblematic about the flexibility of the deductive SMT-approach to model-checking and of its potentialities to incorporate disparate technologies (not only invariant synthesis, but - we believe - also predicate abstraction, acceleration, widening, etc.).

\(^3\)From now on, ‘invariant’ means ‘trace invariant’.
1. Infinite state model-checking

2. Our Declarative Proposal

3. The tool MCMT

4. Experiments and case studies
Initial experience

- Obvious client-server architecture
- Client generates proof obligations (satisfiability modulo theories problems)
- Server = state-of-the-art SMT solver (invoked via API)
- Bottle-neck: huge number of calls to SMT solver had dramatic slow-down of system performances
- Need for techniques to significantly lower the number of invocations to SMT solver

---

Yices is the SMT-solver employed in MCMT.
MCMT: first key problem

- To prevent implicit redundancies, backward reachable states are represented by disjunctions of primitive differentiated formulae

\[ \exists i_1 \cdots \exists i_n \ (L_1(i) \land \cdots \land L_m(i) \land \bigwedge_{k<l} i_k \neq i_l) \]

where \( L_1, \ldots, L_m \) are literals and \( i = i_1 \cdots i_n \).

- Fixpoint and safety tests amount to check consistency of sets like

\[ \exists i P, \neg \exists j_1 Q_1, \ldots, \neg \exists j_n Q_n, \]

where the existentially quantified formulae are all primitive differentiated.
MCMT: first key problem

- To prevent implicit redundancies, backward reachable states are represented by disjunctions of primitive differentiated formulae

\[ \exists i_1 \cdots \exists i_n \left( L_1(i) \land \cdots \land L_m(i) \land \bigwedge_{k<l} i_k \neq i_l \right) \]

where \( L_1, \ldots, L_m \) are literals and \( i = i_1 \cdots i_n \).

- Fixpoint and safety tests amount to check consistency of sets like

\[ \exists i \, P, \neg \exists j_1 \, Q_1, \ldots, \neg \exists j_n \, Q_n, \]

where the existentially quantified formulae are all primitive differentiated.
MCMT: first key problem

- According to our decision procedure, we should **skolemize** the $i$ and **instantiate** the $j_1, \ldots, j_n$ to the $i$ in all possible ways.
- Most such instances however do not contribute to inconsistency.
- To illustrate, suppose that

\[
P(i) := a[i_1] = c \land P'(i)
\]
\[
Q_1(j_1) := a[j_1] = d \land Q'
\]

Then, any instance containing $(j_1 \mapsto i_1)$ is useless, because

\[
a[i_1] = c \land P'(i) \land \neg (a[i_1] = d \land Q')
\]

is equivalent to $a[i_1] = c \land P'(i)$, in case $T_E |\models c \neq d$ (this is the case e.g. when $c, d$ are distinct program locations).
MCMT: first key problem

- According to our decision procedure, we should skolemize the $i$ and instantiate the $j_1, \ldots, j_n$ to the $i$ in all possible ways.
- Most such instances however do not contribute to inconsistency.

To illustrate, suppose that

$$P(i) := a[i_1] = c \land P'(i)$$
$$Q_1(j_1) := a[j_1] = d \land Q'$$

Then, any instance containing ($j_1 \mapsto i_1$) is useless, because

$$a[i_1] = c \land P'(i) \land \neg(a[i_1] = d \land Q')$$

is equivalent to $a[i_1] = c \land P'(i)$, in case $T_E \models c \neq d$ (this is the case e.g. when $c, d$ are distinct program locations).
MCMT: first key problem

- According to our decision procedure, we should **skolemize** the $i$ and **instantiate** the $j_1, \ldots, j_n$ to the $i$ in all possible ways.
- Most such instances however do not contribute to inconsistency.
- To illustrate, suppose that

\[
P(i) := a[i_1] = c \land P'(i)
\]
\[
Q_1(j_1) := a[j_1] = d \land Q'
\]

Then, any instance containing $(j_1 \mapsto i_1)$ is useless, because

\[
a[i_1] = c \land P'(i) \land \neg(a[i_1] = d \land Q')
\]

is equivalent to $a[i_1] = c \land P'(i)$, in case $T_E \models c \neq d$ (this is the case e.g. when $c, d$ are distinct program locations).
MCMT: first key problem

- According to our decision procedure, we should skolemize the $i$ and instantiate the $j_1, \ldots, j_n$ to the $i$ in all possible ways.
- Most such instances however do not contribute to inconsistency.
- To illustrate, suppose that

$$P(i) := a[i_1] = c \land P'(i)$$
$$Q_1(j_1) := a[j_1] = d \land Q'$$

Then, any instance containing $(j_1 \mapsto i_1)$ is useless, because

$$a[i_1] = c \land P'(i) \land \neg(a[i_1] = d \land Q')$$

is equivalent to $a[i_1] = c \land P'(i)$, in case $T_E \models c \neq d$ (this is the case e.g. when $c, d$ are distinct program locations).
MCMT: solution to the first key problem

- Filtering modulo enumerated data-type theory
- Each primitive differentiated formula is decorated with the values of each array variables whose co-domain is an enumerated data-type (in case these values are cheaply available from formula reading)
- Such decorations are used to dynamically filter out useless instances in the incremental safety and the fix-point checks
- Also used to establish statically whether a transition can fire or not.
MCMT: solution to the first key problem

- Filtering modulo enumerated data-type theory
- Each primitive differentiated formula is decorated with the values of each array variables whose co-domain is an enumerated data-type (in case these values are cheaply available from formula reading)
- Such decorations are used to dynamically filter out useless instances in the incremental safety and the fix-point checks
- Also used to establish statically whether a transition can fire or not.
MCMT: solution to the first key problem

- Filtering modulo enumerated data-type theory
- Each primitive differentiated formula is decorated with the values of each array variables whose co-domain is an enumerated data-type (in case these values are cheaply available from formula reading)
- Such decorations are used to **dynamically** filter out useless instances in the *incremental* safety and the fix-point checks
- Also used to establish **statically** whether a transition can fire or not.
MCMT: solution to the first key problem

- Filtering modulo enumerated data-type theory
- Each primitive differentiated formula is decorated with the values of each array variables whose co-domain is an enumerated data-type (in case these values are cheaply available from formula reading)
- Such decorations are used to dynamically filter out useless instances in the incremental safety and the fix-point checks
- Also used to establish statically whether a transition can fire or not.
MCMT: additional features

- When computing the pre-image, the **existential prefix grows**: under suitable hypotheses, it is possible to **statically** pre-process the transition $\tau$ so as to lower the number of (or even to avoid adding) new existentially quantified variables.

- Simple forms of **acceleration techniques** are also used to preprocess single transitions.

- **Backward and forward redundancy elimination** are applied before/after generating new preimages; we start from instances of more recent formulae (chronological redundancy checking).
MCMT: additional features

- When computing the pre-image, the **existential prefix grows**: under suitable hypotheses, it is possible to **statically** pre-process the transition $\tau$ so as to lower the number of (or even to avoid adding) new existentially quantified variables.

- Simple forms of **acceleration techniques** are also used to preprocess single transitions.

- **Backward and forward redundancy elimination** are applied before/after generating new preimages; we start from instances of more recent formulae (chronological redundancy checking).
MCMT: additional features

- When computing the pre-image, the existential prefix grows: under suitable hypotheses, it is possible to statically pre-process the transition \( \tau \) so as to lower the number of (or even to avoid adding) new existentially quantified variables.

- Simple forms of acceleration techniques are also used to preprocess single transitions.

- Backward and forward redundancy elimination are applied before/after generating new preimages; we start from instances of more recent formulae (chronological redundancy checking)
MCMT: invariant synthesis

- **Invariant synthesis**: candidate invariants are extracted from the set of reachable states and **limited resource backward reachability attempts** to show they are indeed invariants.

- Techniques for candidate invariants extraction can be either syntactic or based on **index** and/or **signature abstraction**.

- Invariants can also be **user-suggested** via a kind of ‘proof plan’.

- Effective impact of various invariant search settings strongly depends on benchmarks.
Invariant synthesis: candidate invariants are extracted from the set of reachable states and limited resource backward reachability attempts to show they are indeed invariants.

Techniques for candidate invariants extraction can be either syntactic or based on index and/or signature abstraction.

Invariants can also be user-suggested via a kind of ‘proof plan’.

Effective impact of various invariant search settings strongly depends on benchmarks.
MCMT: invariant synthesis

- **Invariant synthesis**: candidate invariants are extracted from the set of reachable states and limited resource backward reachability attempts to show they are indeed invariants.
- Techniques for candidate invariants extraction can be either syntactic or based on index and/or signature abstraction.
- Invariants can also be user-suggested via a kind of ‘proof plan’.
- Effective impact of various invariant search settings strongly depends on benchmarks.
MCMT: invariant synthesis

- **Invariant synthesis**: candidate invariants are extracted from the set of reachable states and limited resource backward reachability attempts to show they are indeed invariants.
- Techniques for candidate invariants extraction can be either syntactic or based on index and/or signature abstraction.
- Invariants can also be user-suggested via a kind of ‘proof plan’.
- Effective impact of various invariant search settings strongly depends on benchmarks.
Experiments and case studies

1. Infinite state model-checking

2. Our Declarative Proposal

3. The tool MCMT

4. Experiments and case studies
Experiments and case studies

MCMT: mutual exclusion protocol

We first report benchmarks included in the distribution.\(^5\)

<table>
<thead>
<tr>
<th>Problem</th>
<th>depth</th>
<th>#nodes</th>
<th>#deleted</th>
<th>#SMT calls</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bakery_Lamport</td>
<td>12</td>
<td>56</td>
<td>15</td>
<td>595</td>
<td>0.27</td>
</tr>
<tr>
<td>Bakery_Bogus</td>
<td>8</td>
<td>90</td>
<td>14</td>
<td>1413</td>
<td>0.81</td>
</tr>
<tr>
<td>Distrib_Lamport</td>
<td>23</td>
<td>913</td>
<td>242</td>
<td>47574</td>
<td>120.62</td>
</tr>
<tr>
<td>Rickart_Agrawala</td>
<td>13</td>
<td>458</td>
<td>119</td>
<td>35355</td>
<td>187.04</td>
</tr>
<tr>
<td>Szymanski_atomic</td>
<td>23</td>
<td>1745</td>
<td>311</td>
<td>424630</td>
<td>540.19</td>
</tr>
</tbody>
</table>

<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>209</td>
<td>7</td>
<td>0.08</td>
</tr>
<tr>
<td>Bakery_Bogus</td>
<td>8</td>
<td>53</td>
<td>4</td>
<td>1400</td>
<td>7</td>
<td>0.68</td>
</tr>
<tr>
<td>Distrib_Lamport</td>
<td>23</td>
<td>248</td>
<td>42</td>
<td>19254</td>
<td>7</td>
<td>32.84</td>
</tr>
<tr>
<td>Rickart_Agrawala</td>
<td>13</td>
<td>458</td>
<td>119</td>
<td>35355</td>
<td>0</td>
<td>187.04</td>
</tr>
<tr>
<td>Szymanski_atomic</td>
<td>9</td>
<td>22</td>
<td>10</td>
<td>2987</td>
<td>42</td>
<td>1.25</td>
</tr>
</tbody>
</table>

---

\(^5\) Experiments performed on an Intel Centrino 1.729 Ghz with 1 Gbytes of RAM, running Ubuntu v9.0 (Linux kernel v.2.6).
### Default setting

<table>
<thead>
<tr>
<th>Problem</th>
<th>depth</th>
<th>#nodes</th>
<th>#deleted</th>
<th>#SMT calls</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>German07</td>
<td>26</td>
<td>2442</td>
<td>576</td>
<td>121388</td>
<td>145.68</td>
</tr>
<tr>
<td>German_buggy</td>
<td>16</td>
<td>1631</td>
<td>203</td>
<td>41497</td>
<td>49.70</td>
</tr>
<tr>
<td>German_ca</td>
<td>9</td>
<td>13</td>
<td>0</td>
<td>62</td>
<td>0.03</td>
</tr>
<tr>
<td>German_pfs</td>
<td>33</td>
<td>11605</td>
<td>2755</td>
<td>858184</td>
<td>31m 01s</td>
</tr>
</tbody>
</table>

### Best setting

<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>German07</td>
<td>26</td>
<td>2442</td>
<td>576</td>
<td>121388</td>
<td>0</td>
<td>145.68</td>
</tr>
<tr>
<td>German_buggy</td>
<td>16</td>
<td>1631</td>
<td>203</td>
<td>41497</td>
<td>0</td>
<td>49.70</td>
</tr>
<tr>
<td>German_ca</td>
<td>9</td>
<td>13</td>
<td>0</td>
<td>62</td>
<td>0</td>
<td>0.03</td>
</tr>
<tr>
<td>German_pfs</td>
<td>33</td>
<td>11141</td>
<td>2673</td>
<td>784168</td>
<td>149</td>
<td>30m 27s</td>
</tr>
</tbody>
</table>
MCMT: array manipulating programs

**Default setting**

<table>
<thead>
<tr>
<th>Problem</th>
<th>depth</th>
<th>#nodes</th>
<th>#deleted</th>
<th>#SMT calls</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find</td>
<td>4</td>
<td>27</td>
<td>7</td>
<td>691</td>
<td>0.90</td>
</tr>
<tr>
<td>Max_in_Array</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>timeout</td>
</tr>
<tr>
<td>Selection_Sort</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>timeout</td>
</tr>
<tr>
<td>Strcat</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>timeout</td>
</tr>
<tr>
<td>Strcmp</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>timeout</td>
</tr>
<tr>
<td>Strcopy</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>694</td>
<td>1.22</td>
</tr>
</tbody>
</table>

**Best setting**

<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>Find</td>
<td>4</td>
<td>27</td>
<td>7</td>
<td>691</td>
<td>0</td>
<td>0.90</td>
</tr>
<tr>
<td>Max_in_Array</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>46</td>
<td>5</td>
<td>0.03</td>
</tr>
<tr>
<td>Selection_Sort</td>
<td>5</td>
<td>13</td>
<td>2</td>
<td>1141</td>
<td>11</td>
<td>0.62</td>
</tr>
<tr>
<td>Strcat</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>80</td>
<td>2</td>
<td>0.07</td>
</tr>
<tr>
<td>Strcmp</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>21</td>
<td>3</td>
<td>0.01</td>
</tr>
<tr>
<td>Strcopy</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>564</td>
<td>4</td>
<td>0.38</td>
</tr>
</tbody>
</table>
MCMT: parametrized timed systems

We analyzed parametrised systems where single processes are endowed with clocks (joint work with A. Carioni). 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>17</td>
<td>105</td>
<td>5922</td>
<td>8.58</td>
</tr>
<tr>
<td>Fischer_sal</td>
<td>15</td>
<td>56</td>
<td>1107</td>
<td>0.62</td>
</tr>
<tr>
<td>Fischer_sal_buggy</td>
<td>6</td>
<td>16</td>
<td>225</td>
<td>0.13</td>
</tr>
<tr>
<td>Fischer_std</td>
<td>10</td>
<td>16</td>
<td>336</td>
<td>0.18</td>
</tr>
<tr>
<td>Fischer_upp</td>
<td>8</td>
<td>13</td>
<td>198</td>
<td>0.11</td>
</tr>
<tr>
<td>Lynch_mah</td>
<td>17</td>
<td>35</td>
<td>420</td>
<td>0.19</td>
</tr>
<tr>
<td>Lynch_full</td>
<td>25</td>
<td>1128</td>
<td>64147</td>
<td>95.83</td>
</tr>
<tr>
<td>CSMA</td>
<td>4</td>
<td>23</td>
<td>1023</td>
<td>0.90</td>
</tr>
<tr>
<td>CSMA_buggy</td>
<td>7</td>
<td>39</td>
<td>1471</td>
<td>1.35</td>
</tr>
<tr>
<td>tta</td>
<td>7</td>
<td>36</td>
<td>735</td>
<td>2.09</td>
</tr>
<tr>
<td>tta2</td>
<td>8</td>
<td>70</td>
<td>1836</td>
<td>14.07</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).\(^6\)

<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>2792</td>
<td>0</td>
<td>1.18</td>
</tr>
<tr>
<td>Send_Omission (1)</td>
<td>UNSAFE</td>
<td>12</td>
<td>464</td>
<td>26</td>
<td>5</td>
<td>20009</td>
<td>0</td>
<td>17.66</td>
</tr>
<tr>
<td>Send_Omission (2)</td>
<td>UNSAFE</td>
<td>34</td>
<td>9679</td>
<td>770</td>
<td>6</td>
<td>1338058</td>
<td>0</td>
<td>1709.93</td>
</tr>
<tr>
<td>Send_Omission (3)</td>
<td>SAFE</td>
<td>39</td>
<td>11158</td>
<td>1290</td>
<td>6</td>
<td>2558096</td>
<td>19 (+7)</td>
<td>4719.51</td>
</tr>
</tbody>
</table>

\(^6\)This last group of experiments was performed on an Intel Core2 Duo @ 2.66 GHz, 2 GB RAM, running Linux Debian.
Algorithm 1 Pseudo-code for Algorithms 1, 2, and 3

Initialization:
   if (p is the sender)
      then estimate\textsubscript{p} ← m; coord\textsubscript{id}\textsubscript{p} ← 0;
      else estimate\textsubscript{p} ← ⊥; coord\textsubscript{id}\textsubscript{p} ← −1;
   state\textsubscript{p} ← undecided;
End Initialization

for c ← 1, 2, . . . do // Process c becomes coordinator for four rounds
   Round 1:
      All undecided processes p send request (estimate\textsubscript{p}, coord\textsubscript{id}\textsubscript{p}) to c;
      if (c does not receive any request) then it skips rounds 2 to 4;
      else estimate\textsubscript{c} ← estimate\textsubscript{p} with largest coord\textsubscript{id}\textsubscript{p};
   Round 2:
      c multicasts estimate\textsubscript{c};
      All undecided processes p that receive estimate\textsubscript{c} do
         estimate\textsubscript{p} ← estimate\textsubscript{c} and coord\textsubscript{id}\textsubscript{p} ← c;
   Round 3:
      All undecided processes p that do not receive estimate\textsubscript{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\textsubscript{p} ← estimate\textsubscript{p};
         state\textsubscript{p} ← DECIDED;
end for
Experiments and case studies

MCMT: availability

- Free download at
  http://homes.dsi.unimi.it/~ghilardi/mcmt
- Executables for Linux (Ubuntu 8.04) and Mac OSX
- User Manual and Tutorial
- Files for above described benchmarks
- Papers about theoretical framework and implementation
References (general infinite state model checking)

P. A. Abdulla, K. Cerans, B. Jonsson, Y. K. Tsay

General Decidability Theorems for Infinite State Systems,

P. A. Abdulla, N. Ben Henda, G. Delzanno, A. Rezine.

Regular model checking without transducers (on efficient verification of parameterized systems).

P. A. Abdulla, G. Delzanno, A. Rezine.

Parameterized verification of infinite-state processes with global conditions.

P. A. Abdulla, N. Ben Henda, G. Delzanno, A. Rezine.

Handling parameterized systems with non-atomic global conditions.

A. Bouajjani, P. Habermehl, Y. Yurski, and M. Sighireanu.

Rewriting systems with data.

T. Rybina, A. Voronkov.

A logical reconstruction of reachability.
Parosh Aziz Abdulla, Johann Deneux, and Pritha Mahata.
Multi-clock timed networks.

Parosh Aziz Abdulla and Bengt Jonsson.
Model checking of systems with many identical timed processes.

P. A. Abdulla, B. Jonsson.
Verifying programs with unreliable channels.

G. Delzanno, J. Esparza, A. Podelski.
Constraint-based analysis of broadcast protocols.

J. Esparza, A. Finkel, R. Mayr.
On the verification of broadcast protocols.
References (array-based systems and MCMT)

Ghilardi, S., Nicolini, E., Ranise, S. and Zucchelli, D.
Towards SMT Model-Checking of Array-based Systems.
In *Proc. of IJCAR 08*, LNCS 5195, 2008.

S. Ghilardi, S. Ranise, and T. Valsecchi.
Light-Weight SMT-based Model-Checking.

S. Ghilardi and S. Ranise.
Goal-Directed Invariant Synthesis in Model Checking Modulo Thoeries.

S. Ghilardi and S. Ranise.
Model Checking Modulo Theory at work: the intergration of Yices in MCMT.

S. Ghilardi and S. Ranise.
MCMT: A Model Checker Modulo Theories.

A. Carioni, S. Ghilardi and S. Ranise
MCMT in the Land of Parameterized Timed Automata.
In *Proc. of VERIFY 2010*.

F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G.P. Rossi
(Brief Announcement) Automated Support for the Design and Validation of Fault Tolerant Parameterized Systems - a case study.