Automated Verification of Multithreaded and Mobile Programs
via
Infinite-state Symbolic Model Checking

Giorgio Delzanno

D.I.S.I. - Dipartimento di Informatica e Scienze dell’Informazione
University of Genova

Grenoble, March 2003
Background

- Practical examples of multithreaded programs and protocols for distributed systems often have
  - unbounded data: generation of fresh names, …
  - unbounded control: spawning of new processes, …
  - unbounded data and control: multithreaded software
  - process mobility: dynamic reconfiguration of the network programs,…

- Can we still apply automated verification techniques when their state-space becomes infinite in one or more dimensions?
Bounded control, unbounded data
Constraints to symbolically represent data

- Henzinger-Ho-Wong-Toi. *HyTech: a Model Checker for Hybrid Systems*, CAV’97 Based on the Polyhedra library

- Bultan-Gerber-Pugh. *Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetics*, CAV’97 Based on the Omega Library

- ...
Unbounded control, bounded data
Constraints to symbolically represent sets of processes

- Bouajjani-Jonsson-Nilsson-Touili. *Regular Model Checking*, CAV 00
  Based on regular languages

- Delzanno-Raskin-Van Begin. *Towards Verif. of Multithreaded Programs*, TACAS 02

Symbolic analysis for Petri Nets
Unbounded data and parameterized control


Based on symbolic model checking


Based on abstractions+deductive verification
Current Research Line

Overall goal
To develop *sound* and *fully-automatic* methods based on *constraint* programming technology for the verification of concurrent systems with

- *unbounded control*
- *unbounded data*
- *process mobility*

Practical applications
Consistency protocols for distributed systems with shared memory
Cache coherence protocols for multi-processors and multi-line caches
Security protocols
...
Abstractions of *multithreaded* programs
TDL: Thread Definition Language

A language for concurrent systems based on CFSMs enriched with

- local variables over an infinite name domain
- transitions of the form $s \xrightarrow{\alpha} s'[\varphi]$ where
  - $s$ and $s'$ are control locations,
  - $\varphi$ contains guards ($x = y$, $x \neq y$) and assignments over local variables and message templates,
  - $\alpha$ is a channel expression
TDL: Thread Definition Language

- A primitive for generations of new names \( x := \text{new} \) where \( x \) is a local variable
- A primitive \( \text{run } T \text{ with } \alpha \) for spawning a new thread \( T \) with initialization of local variables \( \alpha \)
- Rendez-vous communication: \( e!m, e?m \) where
  - \( e \) (channel) is either a constant \( c \) or a local variable \( x \)
  - \( m \) (message) is a tuple \( \langle x_1, \ldots, x_n \rangle \) of (local) variables
- Variables (ranging over an infinite domain of names) are used as ports to achieve \textit{process mobility}
A Challenge-Response Protocol

Thread Alice(local id_A, n_A, m_A);

\[ init_A \xrightarrow{\text{fresh}} \text{gen}_A[n_A := \text{new}] \]

\[ \text{gen}_A \xrightarrow{c!\langle n_A \rangle} \text{wait}_A[\text{true}] \]

\[ \text{wait}_A \xrightarrow{n_A?\langle y \rangle} \text{stop}_A[m_A := y] \]

Thread Bob(local id_B, n_B, m_B);

\[ init_B \xrightarrow{c?\langle x \rangle} \text{gen}_B[n_B := x] \]

\[ \text{gen}_B \xrightarrow{\text{fresh}} \text{ready}_B[m_B := \text{new}] \]

\[ \text{ready}_B \xrightarrow{n_B!\langle m_B \rangle} \text{stop}_B[\text{true}] \]
Initiator

Thread Main;
Local x;

\[ \text{init}_M \xrightarrow{id} \text{create}[x := \text{new}] \]

\[ \text{create} \xrightarrow{\text{new}_A} \text{init}_M[\text{run Alice with } id_A := x, n_A := \bot, m_A := \bot, x := \bot] \]

\[ \text{create} \xrightarrow{\text{new}_B} \text{init}_M[\text{run Bob with } id_B := x, n_B := \bot, m_B := \bot, x := \bot] \]
Sample Run

Global configuration

\[
\langle \langle i_1, \ldots, i_n \rangle, \ell_1, \ldots, \ell_k \rangle
\]

used names  local states

Run

\[
\langle \{ \perp, i_1, i_2 \}, \langle \text{init}_M, \perp \rangle, \langle \text{init}_A, i_1, \perp, \perp \rangle, \langle \text{init}_B, i_2, \perp, \perp \rangle \rangle
\]

\[\Rightarrow \langle \{ \perp, i_1, i_2 \}, \langle \text{init}_M, \perp \rangle, \langle \text{gen}_A, i_1, a^1, \perp \rangle, \langle \text{init}_B, i_2, \perp, \perp \rangle \rangle\]

\[\Rightarrow \langle \{ \perp, i_1, i_2, a^1 \}, \langle \text{init}_M, \perp \rangle, \langle \text{wait}_A, i_1, a^1, \perp \rangle, \langle \text{gen}_B, i_2, a^1, \perp \rangle \rangle\]

\[\Rightarrow \langle \{ \perp, i_1, i_2, a^1, a^2 \}, \langle \text{init}_M, \perp \rangle, \langle \text{wait}_A, i_1, a^1, \perp \rangle, \langle \text{ready}_B, i_2, a^1, a^2 \rangle \rangle\]

\[\Rightarrow \langle \{ \perp, i_1, i_2, a^1, a^2 \}, \langle \text{init}_M, \perp \rangle, \langle \text{stop}_A, i_1, a^1, a^2 \rangle, \langle \text{stop}_B, i_2, a^1, a^2 \rangle \rangle\]
The Verification of our Example is Challenging

- Suppose we want to prove that at the end of every session any two agents who started the protocol eventually get to know both nonces they exchanged

- This is a verification problem for a parameterized system in which individual components have infinitely many possible states (we generate fresh names and new threads)
Several Problems to Solve

- We need a specification language for *parameterized systems with unbounded local data*
- We need an *assertional language* to specify safety properties
- We need *sound and fully automatic* procedures to validate the specification against the desired property
Low Level Specification Language
Multiset Rewriting + Constraints

• Multiset rewriting over first order atomic formulas (MSR) can be used as a flexible specification language for concurrent systems

• MSR has been introduced to specify security protocols
  – Locality of process definitions and communication via rendez-vous
  – First order terms as color for processes

• The combination of MSR with a constraint system $C$ can be used to symbolically represent systems with heterogeneous data structures

• The resulting specification language is called MSR($C$)
**MSR(>,=) specification of the sample protocol**

We use a global counter to manage fresh name generation

\[
\begin{align*}
\text{init} & \rightarrow \text{fresh}(x) \mid \text{init}_M(y) : x > 0, y = 0. \\
\text{fresh}(x) \mid \text{init}_M(y) & \rightarrow \text{fresh}(x') \mid \text{create}(y') : x' > y', y' > x. \\
\text{create}(x) & \rightarrow \text{init}_M(x') \mid \text{init}_A(id', n', m') : x' = x, id' = x, n' = 0, m' = 0. \\
\text{create}(x) & \rightarrow \text{init}_M(x') \mid \text{init}_B(id', n', m') : x' = x, id' = x, n' = 0, m' = 0.
\end{align*}
\]
Core Protocol

\[\text{init}_A(id, n, m) | \text{fresh}(u) \rightarrow \text{gen}_A(id', n', m') | \text{fresh}(u') : \]
\[u' > n', n' > u, m' = m, id' = id.\]

\[\text{gen}_A(id_1, n, m) | \text{init}_B(id_2, u, v) \rightarrow \text{wait}_A(id'_1, n', m') | \text{gen}_B(id'_2, u', v') : \]
\[n' = n, m' = m, u' = n, v' = v, id'_1 = id_1, id'_2 = id_2.\]

\[\text{gen}_B(id, n, m) | \text{fresh}(u) \rightarrow \text{ready}_B(id', n', m') | \text{fresh}(u') : \]
\[u' > m', m' > u, n' = n, id' = id.\]

\[\text{wait}_A(id_1, n, m) | \text{ready}_B(id_2, u, v) \rightarrow \text{stop}_A(id'_1, n', m') | \text{stop}_B(id'_2, u', v') : \]
\[n = u, n' = n, m' = v, u' = u, v' = v, id'_1 = id_1, id'_2 = id_2.\]

\[\text{stop}_A(id, n, m) \rightarrow \text{init}_A(id', n', m') : n' = 0, m' = 0, id' = id.\]

\[\text{stop}_B(id, n, m) \rightarrow \text{init}_B(id', n', m') : n' = 0, m' = 0, id' = id.\]
Configuration and Run

A Configuration is a multiset $\mathcal{M}$ of ground atomic formulas

One Step Rewriting

$$\underline{\text{init}_A(i, j, l) | \text{init}_B(a, b, c) | \text{init}_A(r, s, t) | \text{fresh}(k)} \quad \Rightarrow \quad \underline{\text{gen}_A(i, j', l) \mid \text{init}_B(a, b, c) | \text{init}_A(r, s, t) | \text{fresh}(k')}$$

using the instance rule

$$\text{init}_A(i, j, l) | \text{fresh}(k) \longrightarrow \text{gen}_A(i, j', l) \mid \text{fresh}(k') \text{ with } k' > j' > k$$

Reachability $\mathcal{M}$ is reachable if $\text{init} \Rightarrow^* \mathcal{M}$
Properties and Assertional Language
Parameterized Verification

• Let $S$ be the set of *good configurations*. The corresponding *safety property* holds if for any $\mathcal{M}$

\[ \text{if } \text{init} \Rightarrow^* \mathcal{M} \text{ then } \mathcal{M} \in S \]

• Dually, let $U$ be the set of *bad configurations*, then the property holds if

\[ \text{init} \notin \text{Pre}^*(U) \]

where $\text{Pre}^*(U) = \{ \mathcal{M} \mid \mathcal{M} \Rightarrow^* \mathcal{M}', \mathcal{M}' \in U \}$

• We have to explore a potentially infinite number of configurations
Symbolic Representation of Configurations

- Unsafe States can be represented as the constrained configuration:

\[ \text{stop}_A(i_1, n_1, m_1) \mid \text{stop}_B(i_2, n_2, m_2) : n_1 = n_2, m_1 > m_2. \]
\[ \text{stop}_A(i_3, n_3, m_3) \mid \text{stop}_B(i_4, n_4, m_4) : n_3 = n_4, m_4 > m_3. \]

- if we consider its upward-closed denotations

\[
\begin{align*}
\llbracket U \rrbracket &= \{ \text{stop}_A(i, n, m) \mid \text{stop}_B(j, n, m') \oplus M \\
& \quad \forall i, j, n, m \neq m', \\
& \quad \forall \text{ configuration } M \} \nonumber
\end{align*}
\]

- defined in general as follows

\[
\begin{align*}
\llbracket M : \varphi \rrbracket &= \{ N \mid \sigma(M) \preceq N, \sigma \text{ solution of } \varphi \}
\end{align*}
\]
Verification Procedures
Backward Reachability

$I = \text{Initial States}$

$Pre^*(U)$

$U = \text{Unsafe States}$

$Pre(U)$

minimal violations
Pre-image Computation

From

\[ p(u) \mid p(v) : true \]

using the rule

\[ w(x) \mid t(y) \rightarrow p(x') \mid t(y') : x = y, x' = x, y' = y \]

we get

\[ p(u) \mid w(x) \mid t(y) : x = y \]

but also

\[ p(u) \mid p(v) \mid w(x) \mid t(y) : x = y \]
Entailment

- We define an ordering based on AC unification and on the entailment relation of the underlying constraints:

- For instance

\[ p(x, y) \mid q(z) \mid r(u) : x > y, y = z \]

entails

\[ q(z') \mid p(x', y') : x' > y' \]

- In fact,

\[ p(x, y) \mid q(z) \text{ and } q(z') \mid p(x', y') \text{ unify via } x = x', y = y', z = z'\]

\[ x' > y', x' = z' \text{ entails } x' > y'. \]
Enforcing Termination
Invariant Strengthening

• We observe that

\[ [U] \subseteq [U'] \]
\[ init \notin \text{Pre}^*(U') \]

implies
\[ init \notin \text{Pre}^*(U) \]

• This idea can be viewed as a static widening
Widening

- We can apply abstractions working on the components of the symbolic representation

\[
\alpha(M : \varphi) = \alpha_f(\alpha_m(M) : \alpha_c(\varphi))
\]

and extend it to \textbf{Pre} as follows \[\textbf{Pre}_\alpha(S) = \alpha(\text{Pre}(S))\]
Sufficient Conditions for Guarantee of Termination

- Monadic constrained configurations with constraints like $x = y$ and $x > y$
  
  $$p(x) \mid q(y) : x > y$$

- Constrained configurations whose constraints are *separable* with respect to positions in atomic formulas
  
  $$p(x, y) \mid q(u, z) : x = u, y > z$$
Towards Security Protocols

- The combination of constraints and uninterpreted function symbols can be used to naturally encode several protocols used for security

\[
fresh(nonce(x)) \rightarrow \\
fresh(nonce(y)) \mid step1(pk(a), \langle n, pk(b) \rangle) \mid net(enc(pk(b), \langle nc(n), pk(a) \rangle)): \\
a > 0, b > 0, y > n, n > x
\]

- We can extend the symbolic verification procedure to the new class of specifications
  - Contrary to forward exploration, in the symbolic backward approach it is not necessary to generate new constants
Conclusions

- *Push-button* verification method for infinite-state concurrent systems based on the paradigm of symbolic model checking and constraints

- Potential application to *nominal process calculi* with *unbounded control, fresh name generation*, and *name mobility*

- Potential application to *verification of security protocols*

- Specialized data structures are needed to scale up

- Abstractions/accelerations are needed for terminations (class of widening operators for security protocols?)