Automated Deductive Verification of Reactive Systems

Hassen SAÏDI

Computer Science Laboratory
SRI International
Menlo Park

saidi@csl.sri.com

www.csl.sri.com/~saidi

Formal verification of systems

$\boxed{\text{Goal}}$

Combination of tools and algorithms for the formal verification of systems.

$$Prog \models \varphi$$

- Deductive approach.
  proof systems : axioms + rules
- Algorithmic approach.
  algorithms : decision procedures

Deductive methods

Approach

- Deductive system.
- Hoare, Dijkstra and Floyd.
- Predicate transformers : pre- et post-conditions.

Tools

- Proof assistants : LCF [Milner]
  PVS, Coq, HOL, Isabell, Nqthm ...

Abstract Interpretation

Approach

- Theory of abstract interpretation P.Cousot & R.Cousot.
- A theory of semantics approximation used for semantics-based program analysis.

Tools

- Abstraction function α.
- Abstract compiler: $[\,] : \mathcal{C} \mapsto \mathcal{A}$.
- Abstract interpreter: Computation of properties at the abstract level.

Model-based methods

Approach

- Decision procedure

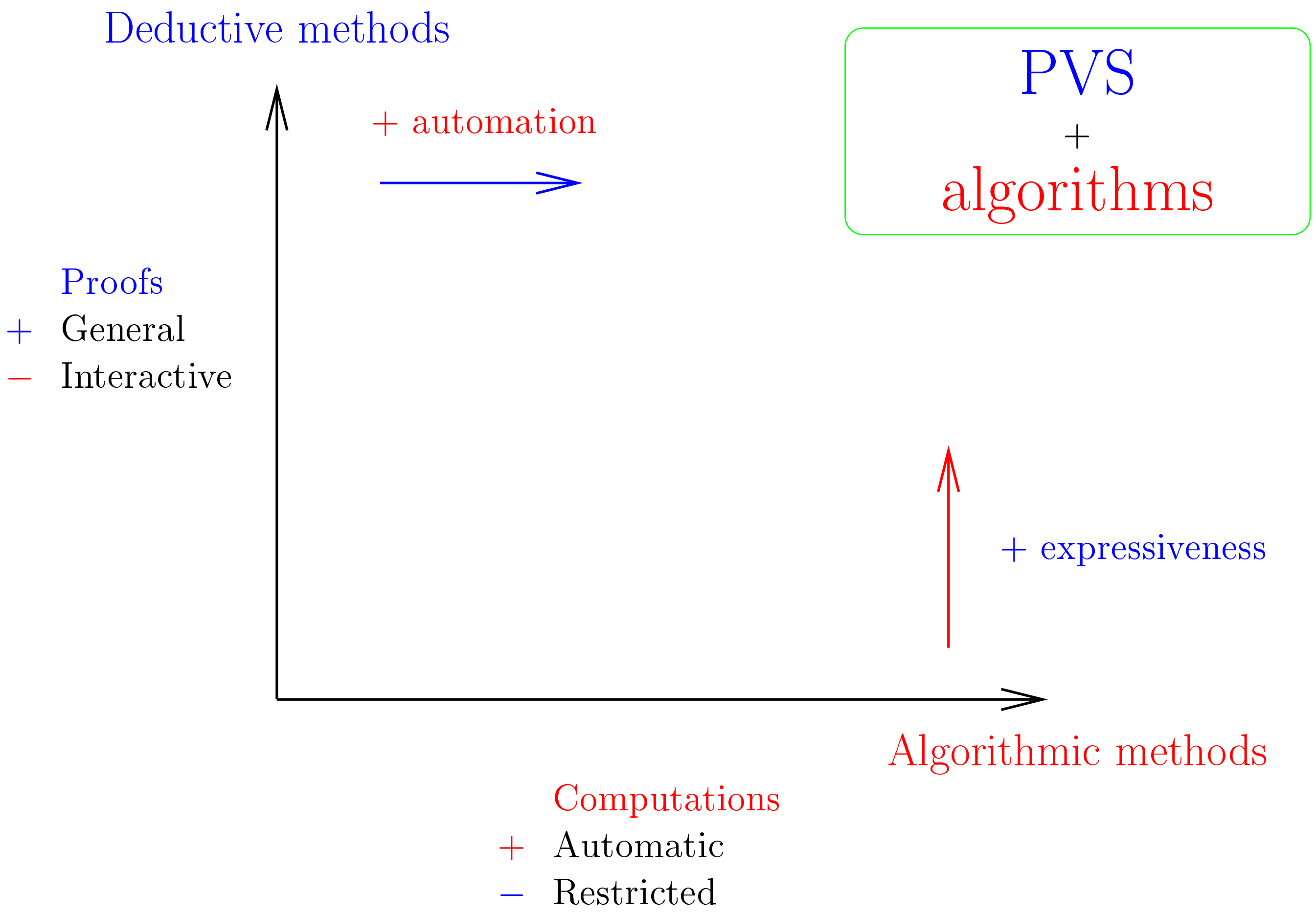
$$\mathcal{M}_{Prog} \models \varphi$$

- Model $=$ transition system.
- Temporal logic [Pnueli77].
- Model-checking algorithms [Queille & Sifakis] and [Clarke & Emerson].

Tools

- SMV, Spin, Kronos, HyTech...
  - Finite state systems.
  - Particular classes of infinite state systems.

Approach : Combination of algorithmic and deductive approaches



Infinite state systems

Mutual exclusion : Bakery.

$$Var \;\; y1, y2 : nat$$

$$\begin{array}{llll}
1: & true & \xrightarrow{Try_1} & y1 := y2 + 1 \quad goto\ 2 \\
2: & y2 = 0 \ \vee \ y1 \leq y2 & \xrightarrow{In_1} & \quad goto\ 3 \\
3: & true & \xrightarrow{Out_1} & y1 := 0 \quad goto\ 1
\end{array}$$

$$\|$$

$$\begin{array}{llll}
1: & true & \xrightarrow{Try_2} & y2 := y1 + 1 \quad goto\ 2 \\
2: & y1 = 0 \ \vee \ y2 < y1 & \xrightarrow{In_2} & \quad goto\ 3 \\
3: & true & \xrightarrow{Out_2} & y2 := 0 \quad goto\ 1
\end{array}$$

- Parallel composition:
  - Communication using shared variables.
  - Interleaving semantics.

inductive invariants

Proof rule

$$\begin{array}{ll} \vdash \; Init \Rightarrow \varphi & (1) \\ \vdash \; \forall \tau_i.\ \{\varphi\}\tau_i\{\varphi\} & (2) \end{array}$$

$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$

$$Prog \models \Box\varphi$$

$$(2) \quad \Leftrightarrow \quad \varphi \Rightarrow \widetilde{pre}[\tau_i](\varphi) \quad \Leftrightarrow \quad post[\tau_i](\varphi) \Rightarrow \varphi$$

Problems

- Discharging the proof obligations.
- Generally, φ is not inductive.

Invariants proof

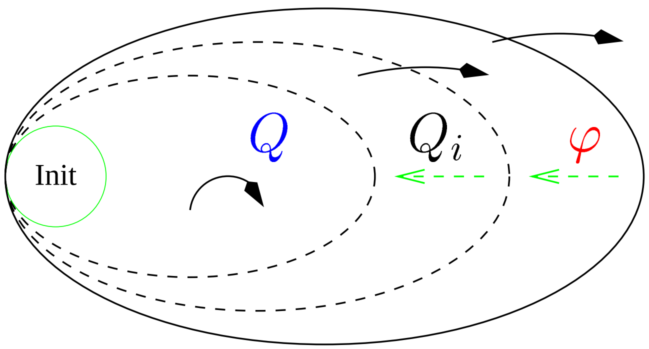
general proof rule

$$\begin{array}{rll} Prog & \models \Box I & (1) \\ & \vdash \; Init \Rightarrow Q & (2) \\ & \vdash \; \forall \tau_i.\ I \wedge \{Q\}\tau_i\{Q\} & (3) \\ & \vdash \; I \wedge Q \Rightarrow \varphi & (4) \\ \hline Prog & \models \Box \varphi & \end{array}$$

Problems

1 Find suitable inductif Q.
2 Find usefull I.
3 Dischare the proof obligations.

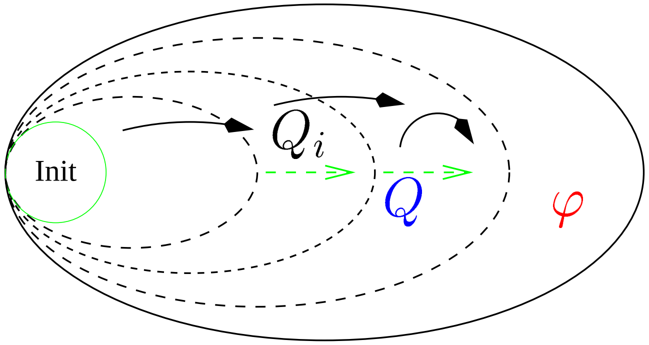
Symbolic backward analysis



$$
\begin{aligned}
Q_0 &= \varphi \\
Q_{i+1} &= Q_i \wedge \widetilde{pre}(Q_i)
\end{aligned}
$$

$$\boxed{Q \;=\; \nu X.\,(X \Rightarrow \widetilde{pre}(X) \wedge \varphi)}$$

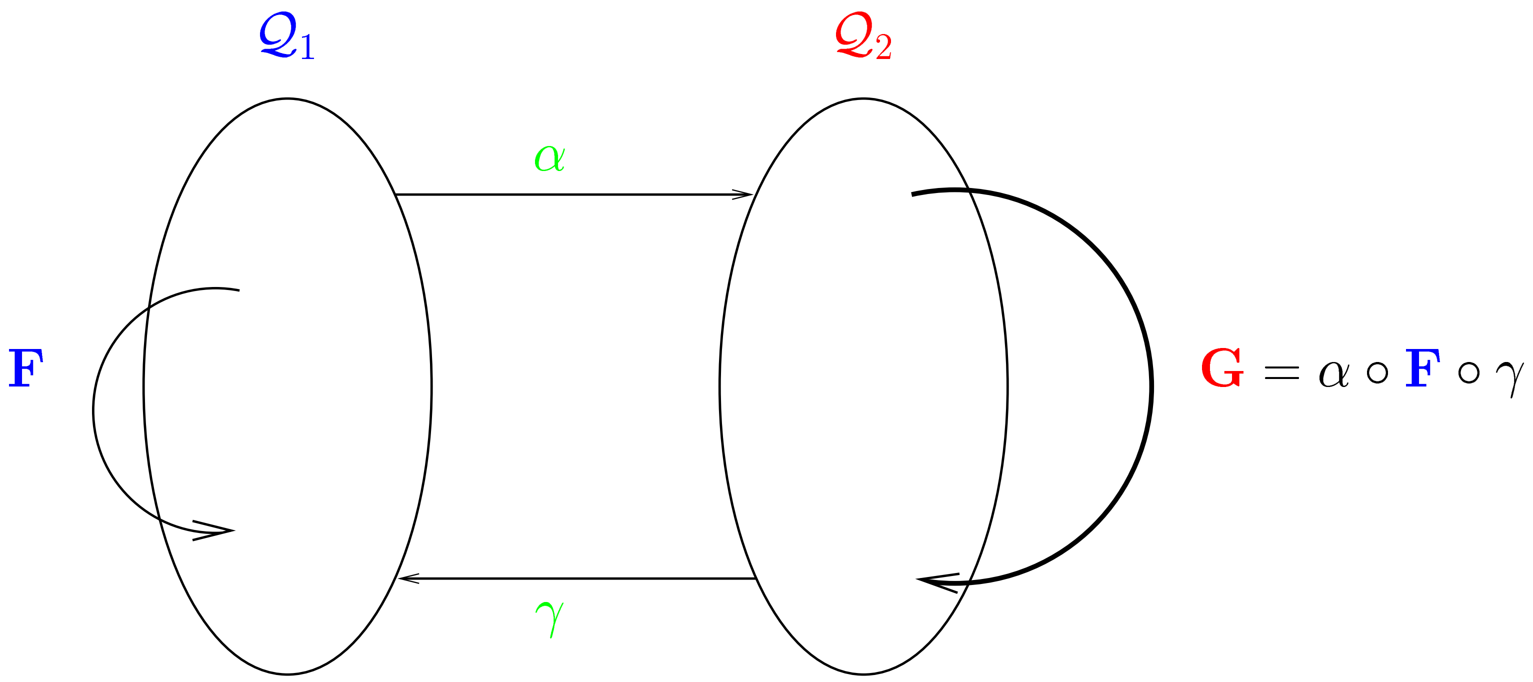
Symbolic forward analysis



$$\begin{aligned} Q_0 &= Init \\ Q_{i+1} &= Q_i \vee post(Q_i) \end{aligned}$$

$$\boxed{Q \;=\; \mu X.\, (Init \vee post(X) \Rightarrow X)}$$

Verification using abstract interpretation



Exact verification:

$$X = \mathbf{F}(X)$$

Approximated verification:

$$Y = \mathbf{G}(Y)$$

Use of symbolic model-checking techniques

- Find Q using the backward analysis.

  At each step, prove:

$$Init \Rightarrow Q_i$$
$$\forall \tau_i.\ Q_i\{\tau_i\}Q_i$$

  Problem : termination

  $Bakery \models \Box\neg(Pc1 = 3 \wedge Pc2 = 3)$: 4 iterations.

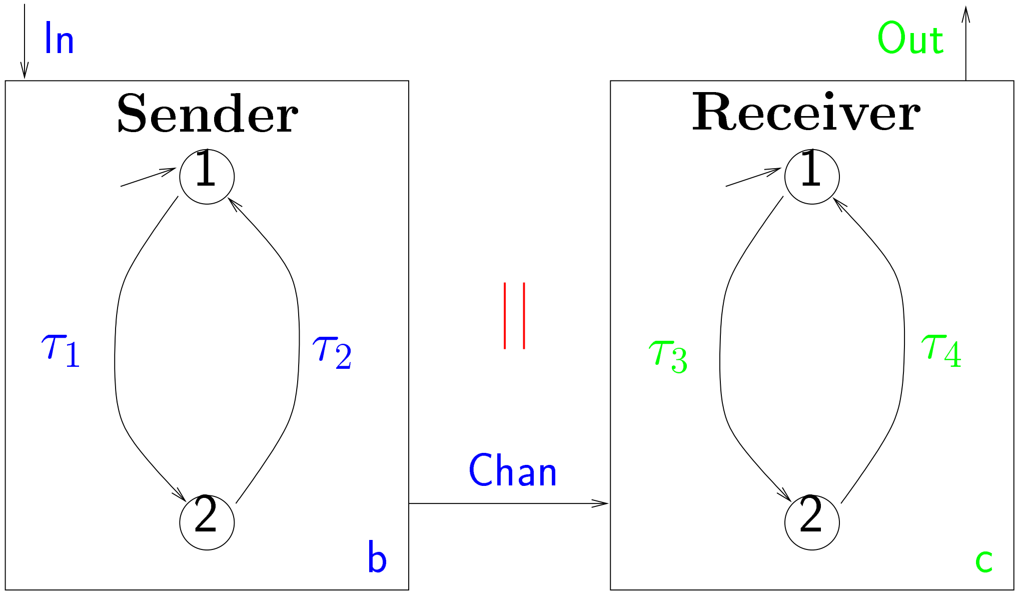
- Find I using the forward: Approximation of the set of reachable states.
  - Generation of invariants by static analysis.

  $Bakery \models \Box\neg(Pc1 = 3 \wedge Pc2 = 3)$: 0 iterations.

Inductive invariant of the Bakery system

$$
\begin{aligned}
&(((\neg(\mathrm{pc1} = 3 \wedge \mathrm{pc2} = 3))\\
&\wedge(\mathrm{pc2} = 2 \wedge (y_1 = 0 \vee y_2 \ < \ y_1) \ \Rightarrow \ \neg\mathrm{pc1} = 3)\\
&\wedge(\mathrm{pc1} = 2 \wedge (y_2 = 0 \vee y_1 \leq y_2) \ \Rightarrow \ \neg\mathrm{pc2} = 3))\\
&\wedge(\mathrm{pc2} = 2 \wedge (y_1 = 0 \vee y_2 \ < \ y_1) \ \Rightarrow \ \neg\mathrm{pc1} = 3 \wedge \neg(\mathrm{pc1} = 2 \wedge (y_2 = 0 \vee y_1 \leq y_2)))\\
&\wedge(\mathrm{pc2} = 1 \ \Rightarrow \ y_1 = 0 \ \Rightarrow \ \neg\mathrm{pc1} = 3)\\
&\wedge(\mathrm{pc1} = 2 \wedge (y_2 = 0 \vee y_1 \leq y_2) \ \Rightarrow \ \neg\mathrm{pc2} = 3 \wedge \neg(\mathrm{pc2} = 2 \wedge (y_1 = 0 \vee y_2 \ < \ y_1)))\\
&\wedge(\mathrm{pc1} = 1 \ \Rightarrow \ y_2 = 0 \ \Rightarrow \ \neg\mathrm{pc2} = 3))\\
&\wedge(\mathrm{pc2} = 2 \wedge (y_1 = 0 \vee y_2 \ < \ y_1) \ \Rightarrow \ \neg\mathrm{pc1} = 3\\
&\wedge\neg(\mathrm{pc1} = 2 \wedge (y_2 = 0 \vee y_1 \leq y_2))\\
&\wedge(\mathrm{pc1} = 1 \ \Rightarrow \ \neg y_2 = 0)) \wedge (\mathrm{pc2} = 1 \ \Rightarrow \ (y_1 = 0 \ \Rightarrow \ \neg\mathrm{pc1} = 3)\\
&\wedge(y_1 = 0 \ \Rightarrow \ \neg\mathrm{pc1} = 3 \wedge \neg\mathrm{pc1} = 2) \wedge (\mathrm{pc1} = 2 \ \Rightarrow \ \neg y_1 = 0))\\
&\wedge(\mathrm{pc1} = 2 \wedge (y_2 = 0 \vee y_1 \leq y_2) \ \Rightarrow \ \neg\mathrm{pc2} = 3\\
&\wedge\neg(\mathrm{pc2} = 2 \wedge (y_1 = 0 \vee y_2 \ < \ y_1))\\
&\wedge(\mathrm{pc2} = 1 \ \Rightarrow \ \neg y_1 = 0)) \wedge (\mathrm{pc1} = 1 \ \Rightarrow \ (y_2 = 0 \ \Rightarrow \ \neg\mathrm{pc2} = 3)\\
&\wedge(\mathrm{pc2} = 2 \ \Rightarrow \ \neg y_2 = 0) \wedge (y_2 = 0 \ \Rightarrow \ \neg\mathrm{pc2} = 3 \wedge \neg\mathrm{pc2} = 2))
\end{aligned}
$$

Example: Alternating bit protocol



$$\begin{aligned}
\tau_1 &: b = c \longrightarrow In := cons(input, In)\\
\tau_2 &: true \longrightarrow Chan := head(In),\ b := \overline{b}\\
\tau_3 &: b \neq c \longrightarrow Out := cons(Chan, Out)\\
\tau_4 &: true \longrightarrow c := \overline{c}
\end{aligned}$$

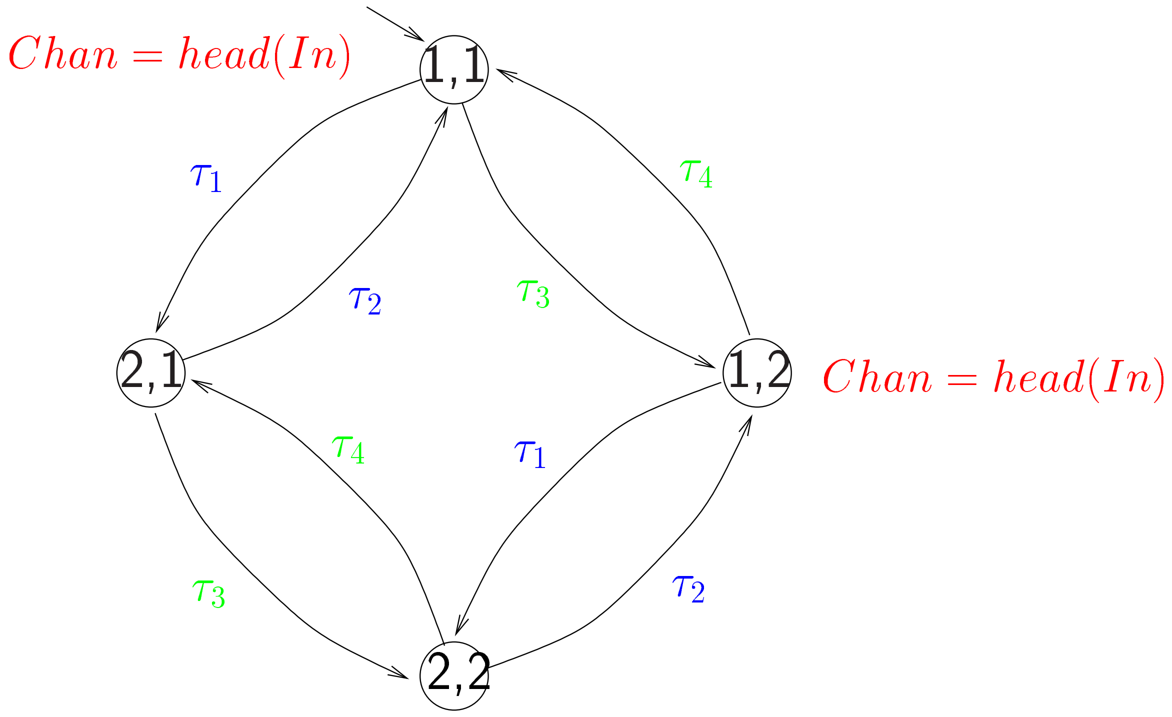
$$Init \quad = \quad b=0 \ \wedge\ c=0 \ \wedge\ In=Out$$

$$\text{Verify} : \quad \text{Altbit} \ \models\ \Box((Out = In) \vee (Out = tail(In)))$$

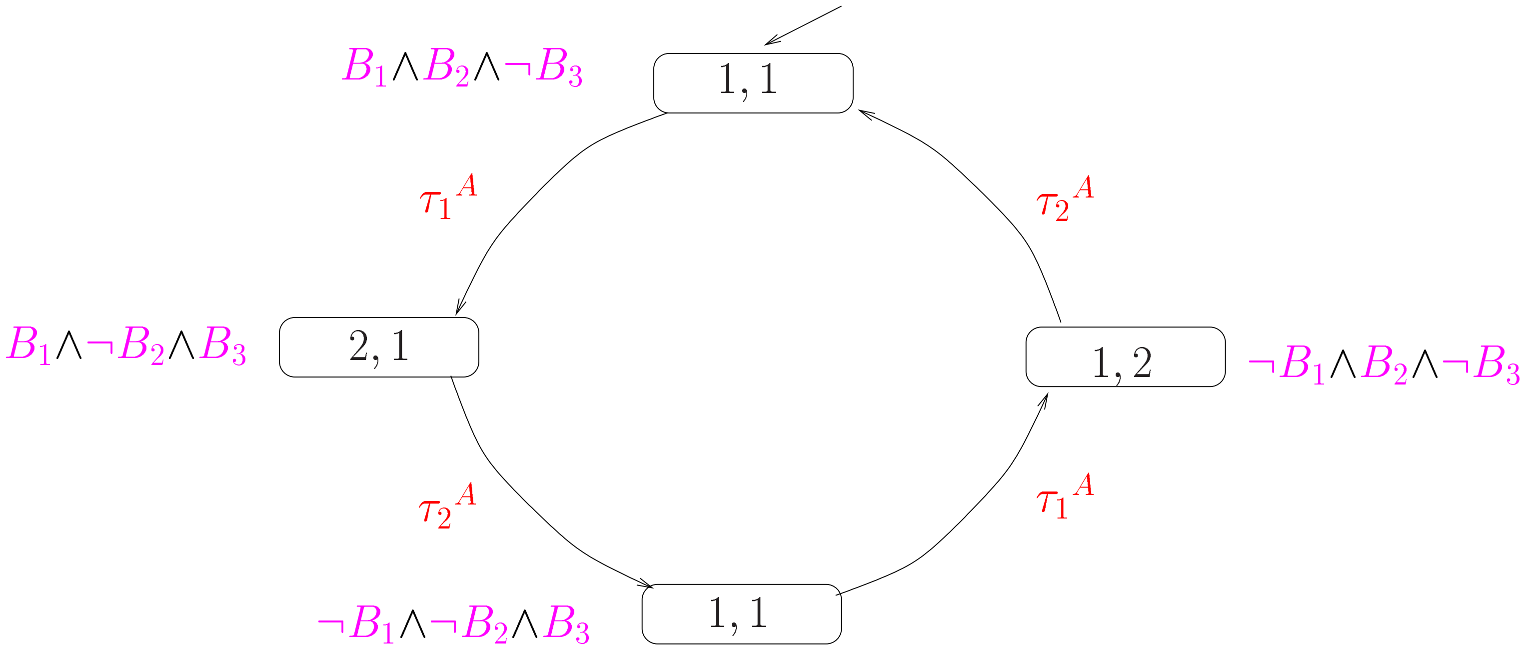
Verification of the protocol

proof fails using:

generation of structural invariants.
Backward computation of an inductive invariant.



Defining a finite state abstraction



$B_1 \equiv b = c$
$B_2 \equiv Out = In$
$B_3 \equiv Out = tail(In)$

$\tau_1{}^A : \; true \longrightarrow B_2 := \neg B_2 \; ; \; B_3 := \neg B_3$
$\tau_2{}^A : \; true \longrightarrow B_1 := \neg B_1$

$$Init \quad = \quad B_1 \wedge B_2 \wedge \neg B_3$$

Abstraction and verification

- Comparison between an implementation (program) and a specification (abstract program) using a simulation relation.
- Simplification of models.
  - What are the preserved properties .
  - What is the abstraction criterion.

Our abstraction

- Construction of an abstract program which
  - simulates the original system.
  - represents a finite state abstraction.

Abstraction and simulation

$S = (Q,\ R,\ q_0)$ a transition system
Q^A un set of abstract states (finite set)
$\alpha \subseteq Q \times Q^A$ an abstraction relation

Then $S^A = (Q^A,\ R^A,\ q_0^A)$ is an abstraction of S iff

$$\begin{tikzcd}
q_1 \arrow[r, "\alpha"] \arrow[d, "R"'] & q_1^A \arrow[d, dashed, "R^A"] \\
q_2 \arrow[r, dashed, "\alpha"'] & q_2^A
\end{tikzcd}$$

Solution

$$\boxed{R^A(q_1^A, q_2^A) = \exists q_1, q_2.\ \ \alpha(q_1, q_1^A) \quad \wedge \quad \alpha(q_2, q_2^A) \quad \wedge \quad R(q_1, q_2)}$$

Construction of abstractions

$$
\begin{tikzcd}
 & Q_1 \arrow[dl, dashed] \arrow[d, dashed, "\times" marking] & q_1^A \arrow[l, "\alpha^{-1}"'] \arrow[d, "\times" marking, "R^A"] \\
\overline{Q_2} & Q_2 & q_2^A \arrow[l, "\alpha^{-1}"]
\end{tikzcd}
$$

$$q_1^A \overset{R^A}{\not\longrightarrow} q_2^A \text{ iff } post[R](\alpha^{-1}(q_1^A)) \Rightarrow \overline{\alpha^{-1}(q_2^A)} \qquad (*)$$

Our abstraction scheme

- Q^A = Valuations $(B_1, \cdots, B_k)$ $\qquad B_i : Bool.$
- $\alpha = \bigwedge_{i=1}^{k} (B_i \equiv \varphi_i) \quad \longrightarrow \quad \alpha^{-1}(Bexp(B_1, \cdots, B_k)) = Bexp[\varphi_i / B_i]$
- $\tau : \quad guard \longrightarrow V_1 := Exp_1, \cdots, V_k := Exp_k$

  $(*)$ is of the form:

$$guard \wedge q_1^A[\varphi_i / B_i] \Rightarrow q_2^A[\varphi_i / B_i][Exp_j / V_j]$$

Abstract Interpretation

Let $\mathcal{Q}_1(\subseteq)$ and $\mathcal{Q}_2(\sqsubseteq)$ two sets. A Galois connexions is a pair (α, γ) of monotonic functions $\alpha : 2^{\mathcal{Q}_1} \mapsto 2^{\mathcal{Q}_2}$ and $\gamma : 2^{\mathcal{Q}_2} \mapsto 2^{\mathcal{Q}_1}$ such that:

$$Id_{2^{\mathcal{Q}_1}} \subseteq \gamma \circ \alpha \quad \text{and} \quad \alpha \circ \gamma \sqsubseteq Id_{2^{\mathcal{Q}_2}}$$

1. $\alpha(\emptyset) = \emptyset$ and $\gamma(\mathcal{Q}_2) = \mathcal{Q}_1$.
2. α distributes over union and γ distributes over intersection.
3. $\alpha \circ \gamma \circ \alpha = \alpha$ and $\gamma \circ \alpha \circ \gamma = \gamma$
4.
   - $\alpha(X) = \sqcap\{Y \mid X \subseteq \gamma(Y)\}$
   - $\gamma(Y) = \sqcup\{X \mid \alpha(X) \sqsubseteq Y\}$

   where $\sqcap$ is the *lub* operator of $2^{\mathcal{Q}_2}$ and $\sqcup$ the *glb* operator of $2^{\mathcal{Q}_1}$.

Boolean abstraction

- General enough to cover many abstraction schemes: data abstraction, abstraction mapping ...
- Natural BDD representation.
- Symbolic verification techniques.
- Abstraction algorithm valid for any boolean algebra.

Computing boolean abstractions

- $\alpha(X) = \wedge\{Y \mid X \Rightarrow \gamma(Y)\}$
- $\gamma(Y) = Y[\varphi_1/B_1, \cdots, \varphi_k/B_k]$

Abstract system

- $Init^A = \alpha(Init)$
- $\tau(q_1^A, q_2^A) : \quad post[\tau](\gamma(q_1^A)) \Rightarrow \gamma(q_2^A) \qquad (\star)$

Construction

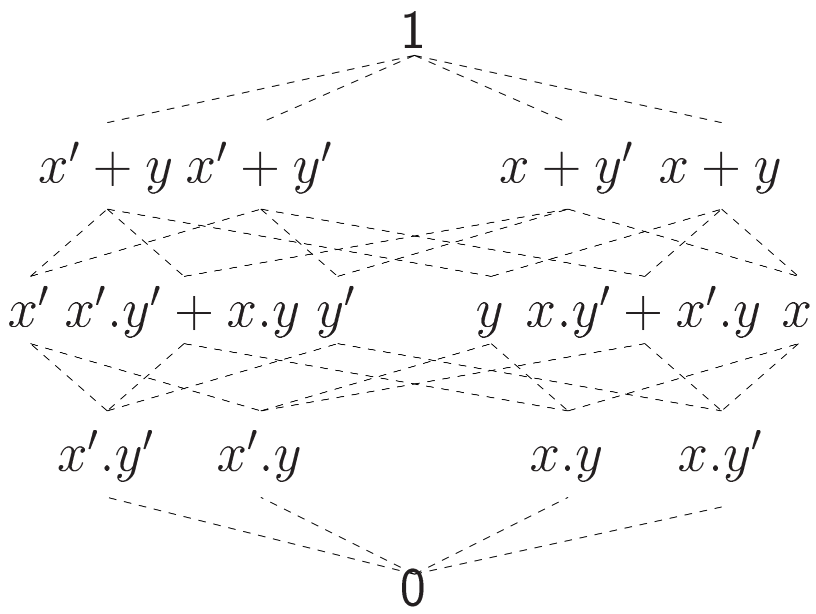
- test $(\star)$ for each τ and each pair (q_1^A, q_2^A)
- test $(\star)$ only when q_1^A is reachable.

$$\tau(q_1^A, q_2^A) : \qquad q_2^A = \alpha(post[\tau](\gamma(q_1^A)))$$

Boolean algebra

$$\alpha(P) = \wedge\{B \mid P \Rightarrow \gamma(B)\}$$

- Enumerate all $B \in \mathcal{D}^A$ and test $P \Rightarrow \gamma(B)$
  2^{2^k} calls to the decision procedure.



Efficient computation of $\alpha(P)$

Theorem 1 *Only* $3^k - 1$ *calls are needed.*

$$P \Rightarrow \gamma(B)$$

Proof: take B *in a CNF form* $D_1 \wedge \cdots \wedge D_l$.

$$\bigwedge P \Rightarrow \gamma(D_i)$$

Corallary 1 *The same algorithm can be used for any sub-algebra.*

$x+y+z$ $\quad$ $x+y'+z$ $\quad$ $x+y+z'$ $\quad$ $x+y'+z'$ $\quad$ $x'+y+z$ $\quad$ $x'+y'+z$ $\quad$ $x'+y+z'$ $\quad$ $x'+y'+z'$

$x+y$ $\quad$ $x'+y$ $\quad$ $x'+y'$ $\quad$ $x+y'$ $\quad$ $[z,y]$ $\quad$ $[x,z]$

x $\quad$ y $\quad$ z $\quad$ z' $\quad$ y' $\quad$ x'

Compositional construction of abstractions

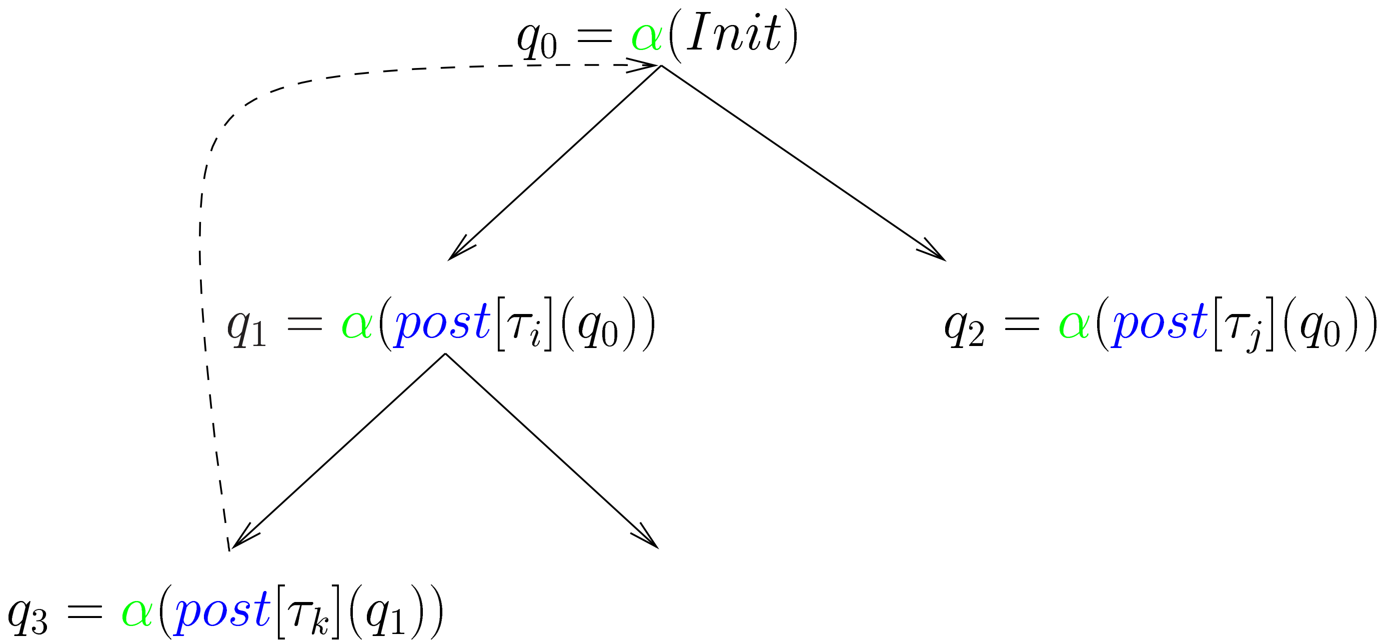
$$\forall (q_1^A, q_2^A),\ \forall \tau :\ post[\tau](\gamma(q_1^A)) \Rightarrow \gamma(q_2^A)$$

$$q_2^A \stackrel{?}{=} \alpha(post[\tau](\gamma(q_1^A))$$

Complexity

$$|\alpha| \ \times \ |\alpha|$$

Construction of abstract state graphs



Complexity

$$|Reach^A| \times |\alpha|$$

The Abstraction algorithm

Initialization $\mathsf{Init} := \alpha(Init)$

For all τ_j $\mathsf{Tau}[j] := \alpha(\mathsf{guard}_j) \wedge I \wedge I' \bigwedge_{\varphi_i \text{ not mod}} (B_i = B_i')$

$\mathsf{To\_Explore} := \mathsf{Init}$; $\mathsf{Reach} := \mathsf{Init}$

Iteration Choose $q_1^A \in \mathsf{To\_Explore}$

$\mathsf{To\_Explore} := \mathsf{To\_Explore} \wedge \neg q_1^A$

For all τ_j : If $q_1^A \Rightarrow \mathsf{guard}[j]$ Then

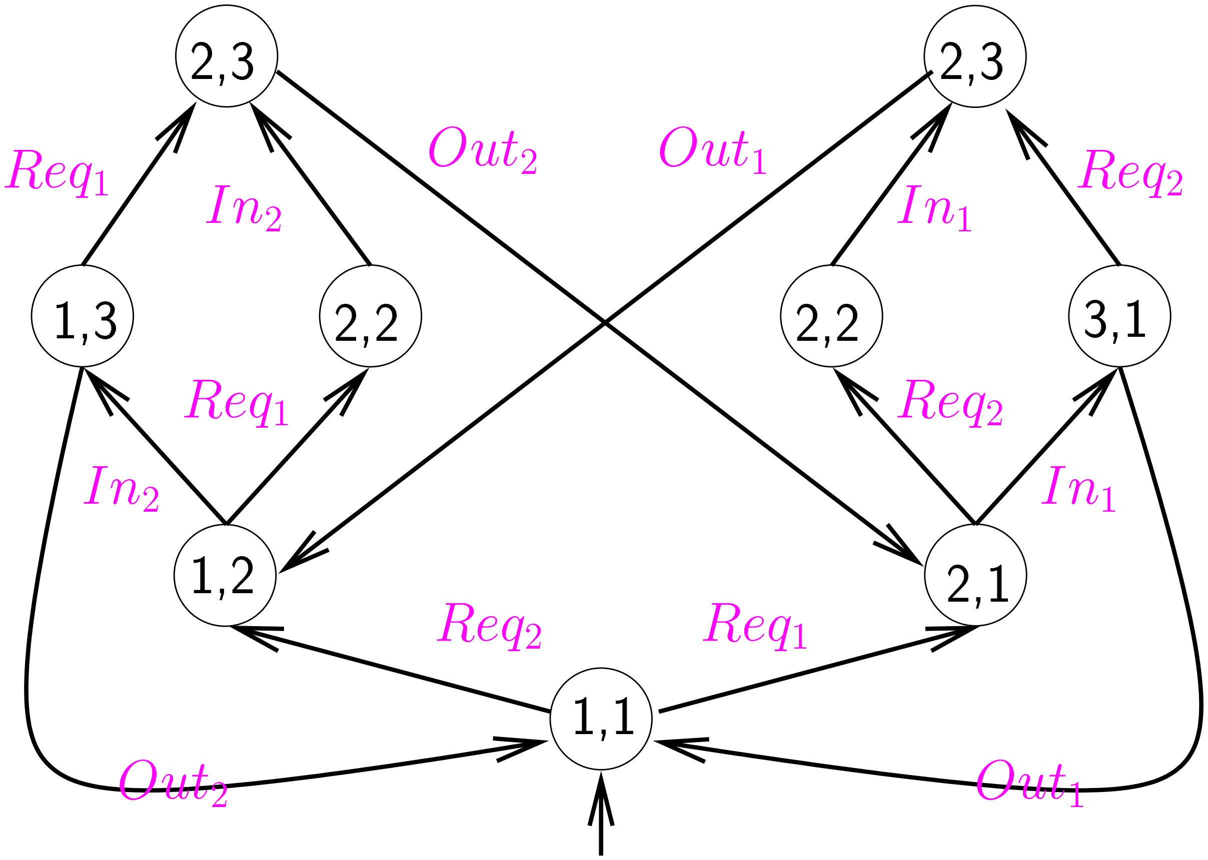
$$\mathrm{Q}_2^A := \alpha(post[\tau](\gamma(q_1^A)))$$

$\mathsf{To\_Explore} := \mathsf{To\_Explore} \vee (\mathrm{Q}_2^A \wedge \neg\mathsf{Reach})$

$\mathsf{Reach} := \mathsf{Reach} \vee \mathrm{Q}_2^A$

$\mathsf{Tau}[j] := \mathsf{Tau}[j] \wedge (q_1^A \Rightarrow \mathrm{Q}_2^A[B_1'/B_1, \cdots, B_k'/B_k])$

Example of abstract state graph



$\varphi_1 \;\equiv\; y_1 = 0$

$\varphi_2 \;\equiv\; y_2 = 0$

$\varphi_3 \;\equiv\; y_1 \leq y_2$

$|Q^A| \;=\; 54$ states

9 reachable

$Bakery^A \models \Box\neg(Pc1 = 3 \wedge Pc2 = 3)$

$Bakery^A \models \Box(Req_i \Rightarrow \Diamond In_i)$

Networks of processes

$$\|_{i=1}^{N} P(i)$$

- *All* the processes satisfy $\varphi_k(i)$

$$\forall i.(\varphi_k(i))$$

- *No* process satisfies $\varphi_k(i)$

$$\forall i.(\neg\varphi_k(i))$$

- *only one* process satisfies $\varphi_k(i)$

$$\exists i.\forall j \neq i.(\varphi_k(i) \wedge \neg\varphi_k(j))$$

- c processes satisfy $\varphi_k(i)$, where c is a constant between 0 and N.

Abstract interpretation as a proof tactic

- Define an abstraction function α.
- Provide an abstract compiler $[\ ]_\alpha$.
- Provide an abstract analyzer.

Theorem 2 (preservation theorem)
Let f be a PVS boolean expression.

$$\vdash [f]_\alpha \ \textit{implies} \ \vdash f$$

The PVS Bdd simplifier

$$
\begin{array}{lcl}
[\ \textsf{TRUE}\ ] & = & \textsf{TRUE} \\
[\ \textsf{FALSE}\ ] & = & \textsf{FALSE} \\
[e_1 \wedge e_2] & = & [e_1] \wedge [e_2] \\
[e_1 \vee e_2] & = & [e_1] \vee [e_2] \\
[\neg e] & = & \neg [e] \\
[\texttt{If}\ c\ \texttt{Then}\ e_1\ \texttt{Else}\ e_2\ \texttt{Endif}] & = & [(c\ \wedge\ e_1)\ \vee\ (\neg c\ \wedge\ e_2)] \\
[a?(x)] & = & [x = a]\ \ (\textsf{normalization}) \\
[e] & = & \textsf{create\_boolean\_variable}(e)
\end{array}
$$

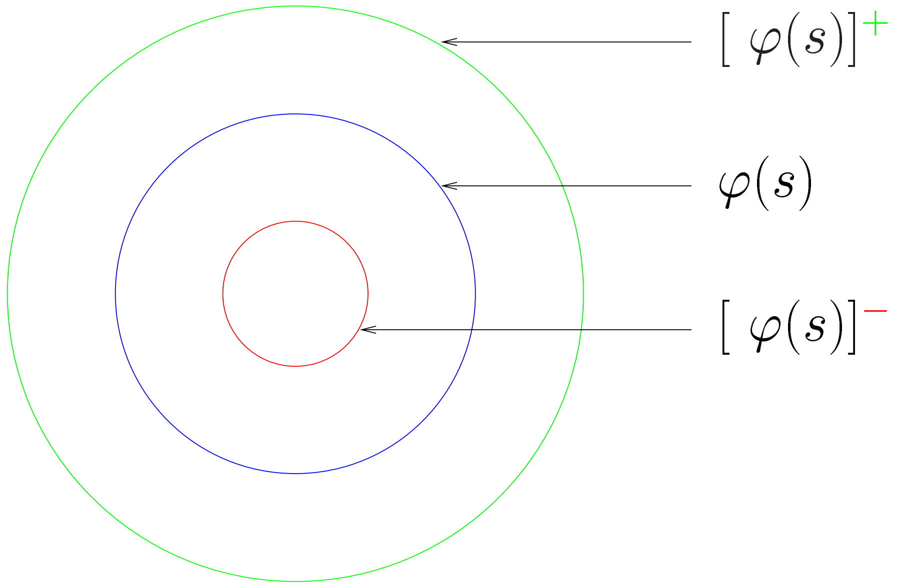
The PVS μ-calculs simplifier

$$
\begin{array}{lcl}
\mathcal{F}(\texttt{bool}) & = & \textsf{TRUE} \\
\mathcal{F}(\texttt{enumtype}) & = & \textsf{TRUE} \\
\mathcal{F}(\texttt{D}- > \texttt{R}) & = & \mathcal{S}(\texttt{D}) \wedge \mathcal{F}(\texttt{R}) \\
\mathcal{F}(\texttt{subtype}(\texttt{D})) & = & \mathcal{S}(\texttt{D}) \vee \mathcal{F}(\texttt{D}) \\
\mathcal{F}(\texttt{recordtype}(\texttt{D}_1 \times \cdots \times \texttt{D}_n)) & = & \mathcal{F}(\texttt{D}_1) \wedge \cdots \wedge \mathcal{F}(\texttt{D}_n) \\
\mathcal{F}(\_) & = & \textsf{FALSE}
\end{array}
$$

$$
\begin{array}{lcl}
[\ \forall/\exists/\lambda\ \ x.e(x)\ ] & = & \forall/\exists/\lambda\ \ \mathcal{B}(x).[\ e(x)\ ] \\
[\ \mu/\nu\ X.e(X)\ ] & = & \mu/\nu\ \mathcal{B}(X).[\ E(x)\ ]
\end{array}
$$

Over and Under approximations

- $[\ \varphi(s)]^{+} \;=\; \bigwedge\{b(s) \mid e \Rightarrow \gamma(b(s))\} \;=\; \alpha_{+}(\varphi(s))$
- $[\ \varphi(s)]^{-} \;=\; \bigvee\{b(s) \mid \gamma(b(s)) \Rightarrow e\} \;=\; \alpha_{-}(\varphi(s))$



$$[\ \neg\varphi(s)]^{+} \;\Leftrightarrow\; \neg[\ \varphi(s)]^{-}$$

Abstract semantics of PVS expressions

$$
\begin{aligned}
[\ e\ ]^{\sigma} &= \ e \qquad \text{if } FV(e) = \emptyset \\
\\
[e_1 \wedge e_2]^{\sigma} &= [e_1]^{\sigma} \wedge [e_2]^{\sigma} \\
[\neg e(s)]^{\sigma} &= \neg [e(s)]^{-\sigma} \\
\\
[\forall(s) : e]^{\sigma} &= \neg\ \exists(\mathcal{R}(s)) : [\neg e]^{\sigma} \\
[\exists(s) : e]^{\sigma} &= \neg\ \forall(\mathcal{R}(s)) : [\neg e]^{\sigma} \\
[\lambda(s) : e]^{\sigma} &= \lambda(\mathcal{R}(s)) : [e]^{\sigma} \\
\\
[\mu(\lambda(Q) : \mathtt{F}(Q))]^{\sigma} &= \mu(\lambda(\mathcal{R}(Q)) : [\mathtt{F}(Q)]^{\sigma}) \\
[\nu(\lambda(Q) : \mathtt{F}(Q))]^{\sigma} &= \nu(\lambda(\mathcal{R}(Q)) : [\mathtt{F}(Q)]^{\sigma}) \\
\\
[e(s)]^{+} &= \alpha_{+}(e(s)) \\
[e(s)]^{-} &= \alpha_{-}(e(s))
\end{aligned}
$$

The tactic abstract

```
  |-------
{1}    (FORALL (state: staterec):
         init(state)
           IMPLIES
             AG(N,
                 LAMBDA state:
                   NOT (critical?(pc1(state))
                          AND
                            critical?(pc2(state))))(state))
Rule? (abstract-and-mc ("lambda(state) :y1(state)=0"
                        "lambda(state) :y2(state)=0"
                        "lambda(state) :y1(state) <= y2(state)"))
```

Theorem 3 (preservation) *Let f be any PVS boolean formula.*

$$\vdash \; [\,[\; f\;]_\alpha\,]_\mu \;\; \textit{implies} \;\; \vdash \; f$$

Conclusions

Results

- Combination of algorithmic and deductive verification.
- Allows fully automatic verification of infinite systems.
- Efficient integration to PVS.
- Combination of several tools.
- Simplification-based methodology.
- Refinement-based methodology.

Things to do

- Improvements of the verification process :
  - – proofs.
  - + static analysis
  - + simplification of formulas
  - + decision procedures
  - + experiments.
- Automatic generation of abstract compilers
  - Express $[\ ]_\alpha$ in PVS.
  - Express $\vdash [\ f\ ]_\alpha$ implies $\vdash f$ in PVS.
  - Generate the lisp code for $[\ ]_\alpha$.