# A tutorial on CAV’13

By Raeeca Narimani

# The main problem in software model checking?

- Finding the right abstraction of a program for a given correctness property.

Safety and Liveness

# A solution?

- Fix the alphabet: the set of statement of the given program

# How to fix the alphabet?

- Run an automaton over it!

# What’s an alphabet?

- alphabet V: a set of non-terminal symbols (variables)
- alphabet ∑ : a set of terminal symbols (non-empty)

Examples:

$\sum 1 = \{ 0, 1 \}$

∑2 = { *, &, hi, 聪, 明}

$\sum 3 = \{ a, b, c, d, e \}$

∑4 = { true, false }

# What’s a word, a string, or a trace?

A a sequence of alphabets

Examples:

Let alphabet $\sum 1$ = { 0, 1 }

Possible words: 0, 1, 01, 110, …

Let alphabet $\sum 3$ = { a, b, c, d, e }

Possible words: a, abcde, ...

# What's Kleene Star, Kleene Closure, or Kleene Operator?

Let alphabet $\sum = \{0,1\}$ then:

$\sum 0 = \varepsilon$

$\sum 1 = \sum = \{ 0, 1 \}$

$\sum 2 = \{ 00, 01, 10, 11 \}$

…

$$\sum^* = \sum 0 \;\cup\; \sum 1 \;\cup\; \sum 2 \;\cup \dots$$

$$\sum + = \sum^* / \sum 0$$

# What's a language (L) ?

- $L \subseteq \Sigma^*$

  let $\Sigma = \{ 0,1 \}$, example languages are

  $L1 = \{ 00 \}$

  $L2 = \{ 1, 01, 100, 01100101, 010100101001 \}$

  $L3 = \{ 1, 01, 001, 0001, \ldots \}$

# What's a Grammar?

A rammar example G is defined by the Tuple $G = \langle V, \Sigma, S, P \rangle$ where

$V$ is an alphabet of non-terminal symbols ("variables").

$\Sigma$ is an alphabet of terminal symbols

$S \in V$ is a start symbol

$P$ is an unordered set of productions (relations) of the form

$A \rightarrow B$ where $A \in V \cup \Sigma+$ and $B \in V \cup \Sigma^*$

# Generating Languages From Grammars

Let grammar $G = \langle$ V, Σ, $S$, $P \rangle$

then language L(G) = {s| s ∈ S /\ S =>* s}        (=>* eventually derives)

# What's an automaton?

- A finite graph
- A grammar $\mathcal{A}$ over the program p:

$\mathcal{A}$ p = ⟨ LOC, δ, ℓ_init, {ℓ_err} ⟩

graph locations (nodes) LOC = { ℓ_init, ℓ_1, ℓ_2, …, ℓ_n }

We assume a fixed set of statements ∑

$\delta \subseteq LOC \times \sum \times LOC$

ℓ_init : the initial location, the starting point of the graph

{ℓ_err} : the set of final locations in the graph

# What is language L($\mathcal{A}$ p)

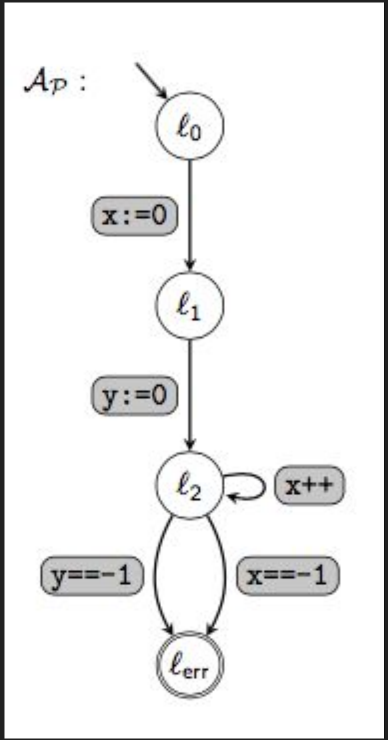
The set of all words accepted by the automaton $\mathcal{A}$ p

is called the language recognized by the automaton $\mathcal{A}$ p,

and is denoted as L($\mathcal{A}$ p).

# Example:

Let $\mathcal{A}_P$:




$\mathcal{A}_P = \langle$ LOC, $\delta$, $\ell_0$, $\{\ell_{err}\}$ $\rangle$

LOC = $\{ \ell_0, \ell_1, \ell_2, \ell_{err} \}$

$\sum = \{$ x := 0 , y := 0 , x++ , x == -1 , y == -1 $\}$

$(\ell_0$, x := 0, $\ell_1) \in \delta$

Let trace $\pi$ = (x := 0) . (y := 0) . (x++) . (x == -1)

Then $\pi \in L(\mathcal{A}_P)$

## Again: the main problem is software model checking?

Finding the right abstraction of the program for a given correctness property

## Solution?

Fixing the program alphabet by running automata over it.

we will use three examples to illustrate how automata over the

alphabet of statements can help to automatically decompose this problem

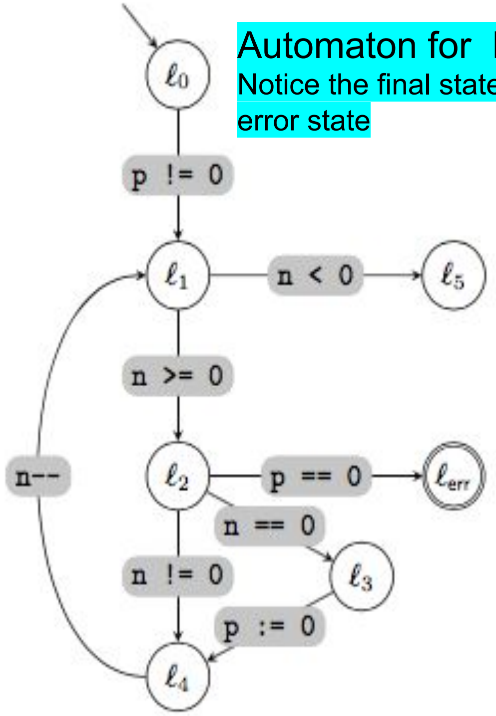
# Example 1: correctness from infeasibility proofs

We show the correctness of program Pex1 by showing the value of variable p Can never be evaluated to zero At location $\ell 2$

Program $P_{ex1}$:

```
ℓ0: assume p != 0;
ℓ1: while(n >= 0)
    {
                exit with error if p == 0
ℓ2:     assert p != 0;
        if(n == 0)
        {
ℓ3:         p := 0;
        }
ℓ4:     n--;
    }
```

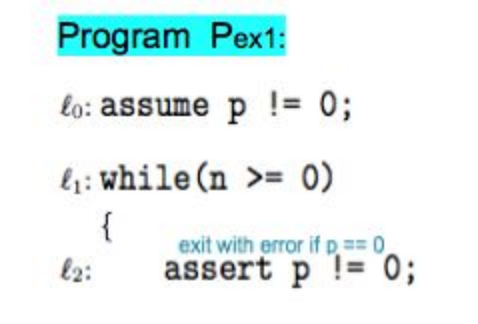
Automaton for $P_{ex1}$:
Notice the final state is the error state

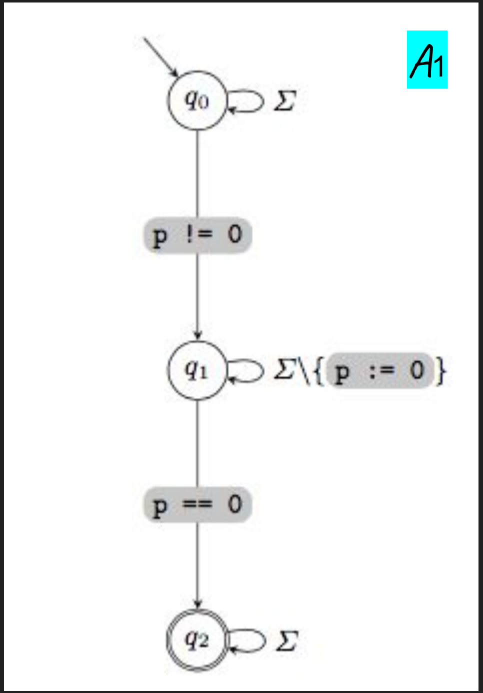


# Example 1 continued

The fastest way to reach error location is if p == 0 immediately after while loop.

This is infeasible (illogical) Because there is no update To value of p between Locations $\ell$ 0 and $\ell$ 2.



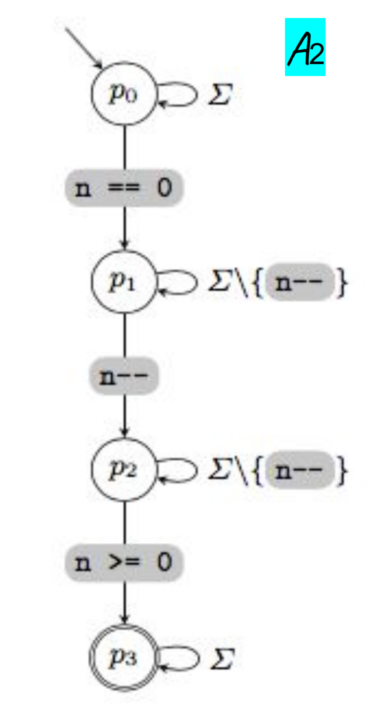
# Example 1 continued

Another possible way to
reach error at location $\ell 2$
Is to have:

- n == 0
- p := 0
- n-- (implying n becomes -1)
- n ≥ 0

```
        if(n == 0)
        {
ℓ3:        p := 0;
        }
ℓ4:     n--;

ℓ1: while(n >= 0)
    {
           exit with error if p == 0
ℓ2:        assert p != 0;
```




This is infeasible because it is impossible to have n ≥ 0 without another update between n == 0, n-- and n--, n ≥ 0

Notice:

$P_{ex1} \subseteq A_1 \times A_2$ hence $P_{ex1}$ is infeasible

# Example 2: automata from the sets of Hoare triples

Consider $P_{ex2}$:

To show the infeasibility of this Automaton, Hoare triples are being used.

```
ℓ0: x := 0;
ℓ1: y := 0;
ℓ2: while(nondet) {x++;}
    assert(x != -1); Error if x == 0
    assert(y != -1); Error if y == 0
```

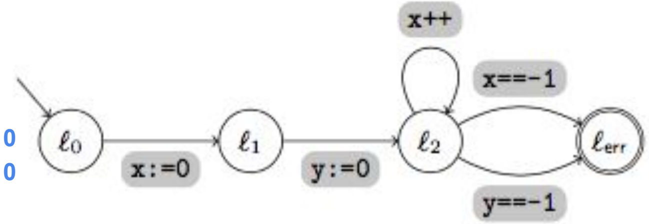
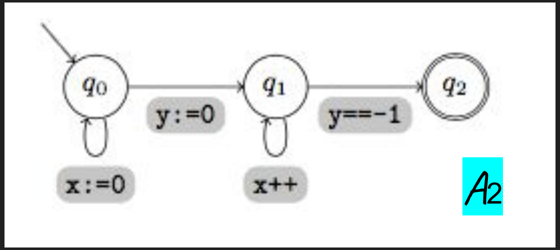


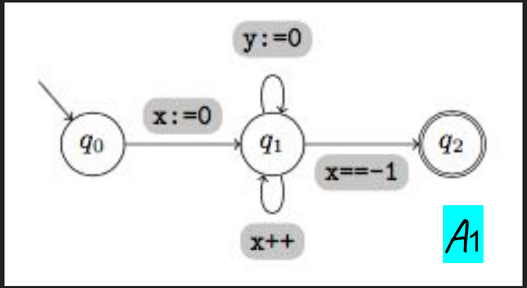

Fig. 3: Example program $\mathcal{P}_{ex2}$

# Example 2 continued

{ true }   x := 0   { true }
{ true }  y := 0    { y = 0 }
{ y = 0 }    x++      { y = 0 }
{ y = 0 }  y == -1  { false }




{ true }   x := 0   { x = 0 }
{ x = 0 }  y := 0    { x = 0 }
{ x = 0 }    x++      { x ≥ 0 }
{ x ≥ 0 }   x == -1  { false }




Again notice that $P_{ex1} \subseteq A_1$ x $A_2$ and hence $P_{ex1}$ is infeasible

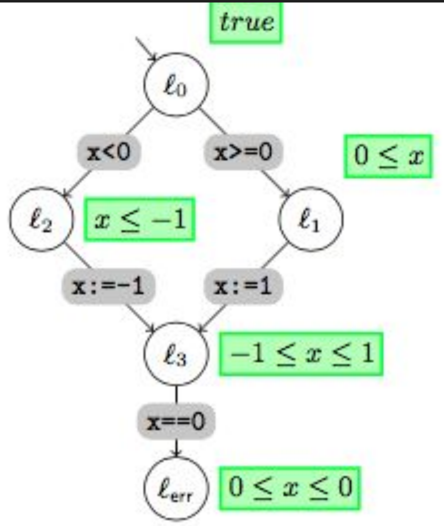
# Example 3: automata for trace partitioning

Let *A1* an automaton with exactly one state for each transition. Traces $P^1_{ex3}$ $P^2_{ex3}$ will be constructed as following:

$$P^1_{ex3} = P_{ex3} \cap A1$$
$$P^2_{ex3} = P_{ex3} / A1$$

$P_{ex3}$

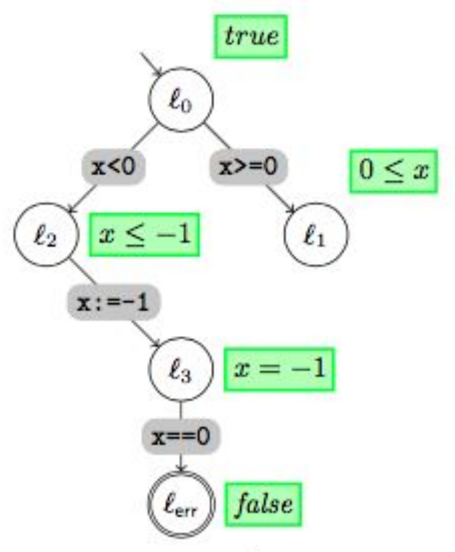
```
ℓ0: if(x>= 0) {
ℓ1:     x := 1;
    } else{
ℓ2:     x := -1;
    }
ℓ3: assert(x != 0);
```



# Example 3 continued

$P^1_{ex3} = P_{ex3} \cap A1$

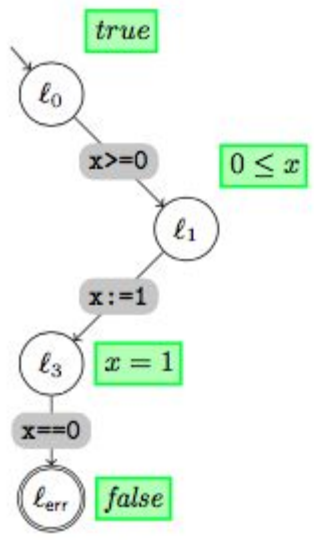
{ x = -1 } x == 0 { false }



(a) $\mathcal{P}^1_{ex3}$

$P^2_{ex3} = P_{ex3} / A1$

{ x = 1 } x == 0 { false }



(b) $\mathcal{P}^2_{ex3}$

# Formal Settings

## Correctness of trace $\tau$:

Let $\tau = st_1 \dots st_n$

We assume a fixed set of assertions $\Phi$ ( contains true, false, and entailment relation $\vDash$)

The Hoare triple $\{ \varphi \}\ \tau\ \{ \psi \}$ is valid for the trace $\tau$ if for some sequence of intermediate assertions $\varphi_1, \dots, \varphi_n$, each of the following Hoare triples is valid: $\{ \varphi \}\ st_1\ \{ \varphi_1 \}, \{ \varphi_1 \}\ st_2\ \{ \varphi_2 \}, \dots, \{ \varphi_{n-1} \}\ st_n\ \{ \psi \}$

If $\tau = \varepsilon$ (empty string) then it must hold that $\varphi \vDash \psi$ for the Hoare triple to hold.

# Formal Settings

## Correctness of trace $\tau$:

Trace $\tau$ is defined to be correct if $\{ \varphi_{pre} \}\ \tau\ \{ \varphi_{post} \}$ is valid for fixed pre/postconditions $\varphi_{pre}$ and $\varphi_{post}$.

# Formal Settings
# Program P:

We formalize a program P as a special kind of graph which we call a control flow graph.

- Recall that $\Sigma^* = \Sigma 0 \cup \Sigma 1 \cup \Sigma 2 \cup \dots$

i.e. $\Sigma^*$ is the set of all traces possible in an alphabet $\Sigma$ (set of statements).

Hence $\tau \in \Sigma^*$.

- The automaton P recognizes a set of traces.

The set of traces accepted by automaton P is called L(P) for language.

Hence $L(P) \subseteq \Sigma^*$. Implying that L(P) = {control flow traces}

# Formal Settings
# Correctness of Program P:

$\{ \varphi_{pre} \}$ P $\{ \varphi_{post} \}$ is valid if P $\subseteq \tau$.

In other words {control flow traces} $\subseteq$ {correct traces}

The language of correct traces is generally not recognizable by a finite automaton.

For every automaton P there exists finite automaton A such that L(A) is the interpolant between {control flow traces} and {correct traces} because if L(P) correct then L(P) = L(A), and L(P) is the smallest automaton that accepts itself

{control flow traces} $\subseteq$ L(A) $\subseteq$ {correct traces}

# Assume Statement

- Used to accommodate control constructs like ‘while’ and ‘if_then_else’
- For every assertion $\psi$ there is a valid Hoare triple $\{ \varphi \}\ \psi\ \{ \varphi' \}$ such that $\varphi'$ is constructed by $\varphi \land \psi$, meaning:

If an execution reaches the statement in a state that satisfies the assertion $\psi$ then the statement is ignored, and if an execution reaches the statement in a state that violates the assertion then the execution is blocked (and the successor location in the control flow graph is not reached).

# Infeasibility ⇒ Correctness

- A trace $\tau$ is infeasible if {true} $\tau$ {false} is valid.
- An infeasible trace thus satisfies every possible pre/postcondition pair.
- In other words, an infeasible trace is correct (for whatever pre/postcondition pair ($\varphi_{pre}$ , $\varphi_{post}$ ) defining the correctness).
- The fact that infeasibility implies correctness is crucial. In general, the set of feasible correct control flow traces is <u>not regular</u>

# Non-reachability of Error Locations

In our setting, this corresponds to the special case where the set F consists of error locations and the postcondition $\varphi\_post$ is the assertion false.

# Validity of assert Statements

It is convenient to express the correctness by the validity of assert statements. Informally, the statement assert(e) is valid if, whenever the statement is reached in an execution of the program, the Boolean expression e evaluates to true.

# Partial Correctness

The partial correctness wrt. ( $\varphi_{pre}$ , $\varphi_{post}$ ) can always be reduced to the partial correctness wrt. the special case ( true , false ) (by modifying the control flow graph of the program: one adds an edge from a new initial location to the old one labeled with the assume statement $\varphi_{pre}$ for the precondition and an edge from each old final location $\varphi_{post}$ to a new final location (an “error location”) labeled with the assume statement $\neg\varphi_{post}$ for the negated postcondition).

# Floyd_Hoare Automata, Definition 1

Automaton $\mathcal{A} = \langle Q, \delta, \boldsymbol{q0}, Q_{final} \rangle$ over the alphabet of statements $\sum$ is a Floyed_Hoare Automaton if there exists a mapping:

$$q \in Q \mapsto \varphi_q \in \Phi$$

that assigns to each state q an assertion $\varphi_q$ such that:

- for every transition (q, st, q0) from state q to state q0 reading the letter st, the Hoare triple $\{ \varphi_q \}$ st $\{ \varphi_{q0} \}$ is valid for the assertions $\varphi_q$ and $\varphi_{q0}$ assigned to q and q0, respectively

$$(q,st,q0) \in \delta \Rightarrow \{ \varphi_q \}\ st\ \{ \varphi_{q0} \} \text{ is valid}$$

# Floyd_Hoare Automata, Definition 1

- the precondition $\varphi_{pre}$ entails the assertion assigned to the initial state $q_0$,

$$q = q_0 \Rightarrow \varphi_{pre} \vDash \varphi_q$$

- the assertion assigned to a final state entails the postcondition $\varphi_{post}$:

$$q \in Q_{final} \Rightarrow \varphi_q \vDash \varphi_{post}$$

The mapping $q \mapsto \Phi_q$ from states to assertions in the definition above is called an annotation of the automaton A.

# Theorem 1

A Floyd-Hoare automaton A accepts only correct traces,

$$L(A) \subseteq \{\text{correct traces}\}$$

i.e., if the trace $\tau$ is accepted by A then the Hoare triple $\{ \varphi_{pre} \}\ \tau\ \{ \varphi_{post} \}$ is valid.

# Theorem 2

If the Floyd-Hoare automata $A_1$, ..., $A_n$ cover the set of control flow traces of the program P (i.e., $P \subseteq A_1 \cup \ldots \cup A_n$ ) then P is correct.

# Construction of Floyd-Hoare Automaton

Let H a set such that if ( $\varphi$, st, $\psi$ ) $\in$ H, then Hoare triple $\{\varphi\}$ st $\{\psi\}$ is valid.

Let $\Phi_H$ the finite set of assertions occurring in H.

Assume $\varphi_{pre}$ , $\varphi_{post}$ $\in$ $\Phi_H$.

We construct the Floyd-Hoare automaton $\mathcal{A}_H$ as follows:

$\mathcal{A}_H = ( Q_H, \delta, q_0, Q_{final} )$ where:

the set of states $Q_H = \{ q_\varphi \mid \varphi \in \Phi_H \}$

( $\forall \varphi \in \Phi_H \; \exists q_\varphi \in Q_H$ i.e. bijective relation between $Q_H$ and $\Phi_H$)

transition relation $\delta = \{ (q_\varphi, st, q_\psi) \mid (\varphi, st, \psi) \in H \}$

precondition state $q_{\varphi pre} = q_0$

postcondition state (unique here) $Q_{final} = \{ q_{\varphi post} \}$

$\mathcal{A}_H$ is a Floyd_Hoare Automaton since there's mapping $q_\varphi \mapsto \varphi$

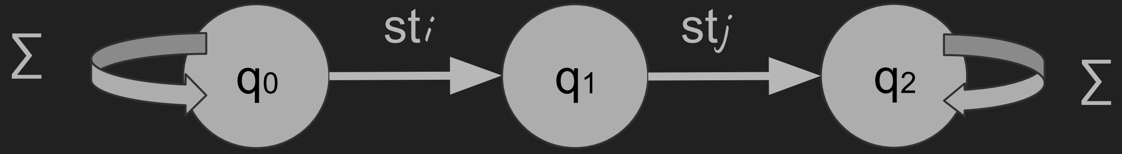
# Construction of Automaton from Infeasibility Proof

Assume that the trace

$$\tau = st_1 \ldots st_i \ldots st_j \ldots st_n$$

is infeasible and that we have a proof of the form: the sequence of the two statements $st_i$ and $st_j$ is infeasible and the statements in between do not modify any variable used in $st_i$ and $st_j$.

We can construct an automaton with the set of states $Q = \{ q_0, q_1, q_2 \}$ as follows:



# Construction of Automaton from Infeasibility Proof

...

# Construction of a correctness proof for the program P

Recalling theorem 2: if the Floyd-Hoare automata $A_1, ..., A_n$ cover the set of control flow traces of the program P (i.e., $P \subseteq A_1 \cup \ldots \cup A_n$ ) then P is correct.

we can construct an automaton $A = A_1 \cup . . . \cup A_n$ for a correctness proof for the program P such that:

$$\{\text{control flow traces}\} \subseteq L(A) \subseteq \{\text{correct traces}\} \qquad (1)$$

# Construction of a correctness proof for the program P

- The construction of $A_1$, ..., $A_n$ can be done in parallel from the n correctness proofs for some choice of traces $\tau_1, \ldots, \tau_n$ (Construction of automata prom infeasibility proofs).
- The construction of A as the union $A = A_1 \cup \ldots \cup A_n$ can also be done incrementally (for n = 0,1,2,...) until (1) holds. Namely, if the inclusion does not yet hold (which is the case initially, when n = 0), then there exists a control flow trace $\tau_{n+1}$ which is not in A. We then construct the automaton $A_{n+1}$ from the proof for the trace $\tau_{n+1}$ and add it to the union, i.e., $A = A \cup A_{n+1}$.

# Construction of a correctness proof for the program P

If (1) holds, i.e. $L(P) \subseteq L(A) \subseteq \{\text{correct traces}\}$ for $A = A_1 \cup \ldots \cup A_n$,
then the programs $P_1, \ldots, P_n$ where $P_i = P \cup A_i$ ( for $i = 1, \ldots, n$ )
define a decomposition of the program P (i.e., $P = P_1 \cup \ldots \cup P_n$ ).

This is how the examples in the introduction were proceeded.

This decomposition is constructed automatically from correctness proofs (in contrast with an approach where one constructs correctness proofs for the modules of a given decomposition).

# Conclusion and Future Work

A new angle of attack at the problem of finding the right abstraction of a program for a given correctness property:

- Existing approaches the techniques amount to constructing a cover (often a partitioning) of the set of control flow traces by automata $A_1, \ldots, A_n$. The construction is restricted in that the automata must be merged into one automaton and, moreover, the states and transitions of the resulting automaton must be in direct correspondence with the nodes and edges of the control flow graph ensure that all control flow traces are indeed covered.

# Conclusion and Future Work

- The approach presented in this paper allows one to remove this restriction.

  In this example, you see how it is useful to remove the restriction that the correctness argument must be based on (an unfolding of) the control flow graph.