A Framework for Static and Runtime Verification of Data- and Control-Oriented Properties

GÖTEBORG UNIVERSITY

Gerardo Schneider
Dept. of Computer Science and Eng.
Sweden
gerardo@cse.gu.se
http://www.cse.chalmers.se/~gersch/

CHALMERS

Joint work with

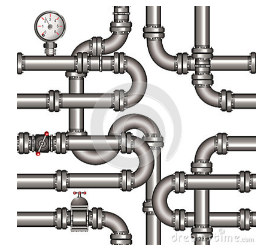
Wolfgang Ahrendt, Mauricio Chimento and Gordon Pace

Vetenskapsrådet

IFIP WG 1.3 meeting
Binz, January 2016

We are interested in proving properties…

Each time we call method *m1* if *Pre* holds (e.g. *x* positive) then *Post* should hold (e.g. *y* positive)



Control flow

Data

Whenever *m1* is executed,
then *m2* should be executed before *m3*

After calling method *m1* and then *m2*,
if *Pre2* holds (for *m2*) then *Post2* should hold (for *m2*)
Otherwise (no *m1*), if *Pre2'* then *Post2'*…

… and *m2* should be called no later than 30 sec after *m1*

Statically

Time

Runtime

A Simplified View of Formal Verification Techniques

Model Checking

Theorem Proving

Deductive (Program) Verification

Runtime Trace Checking

Runtime Assertion Checking

High Abstraction

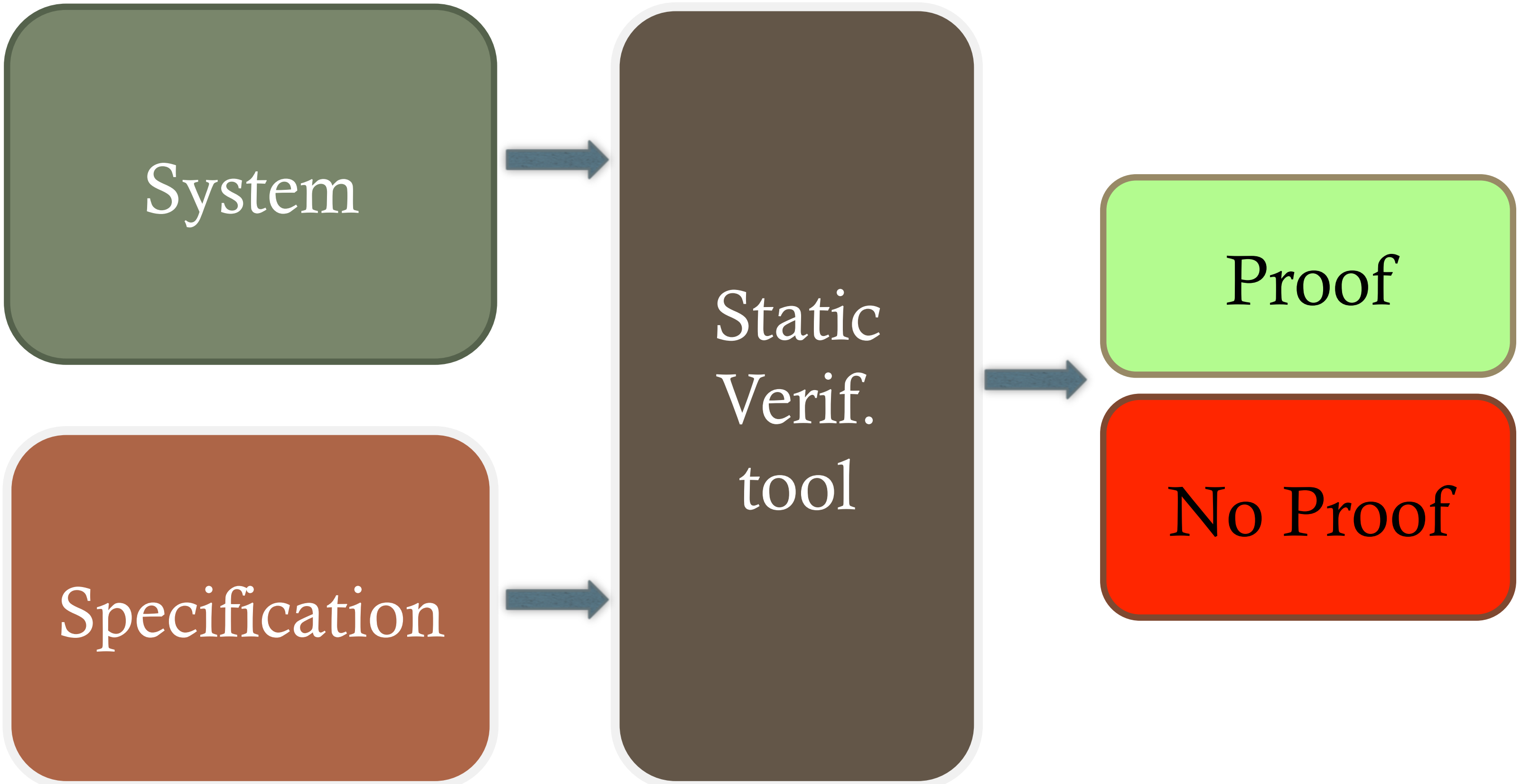
Low Abstraction

Static

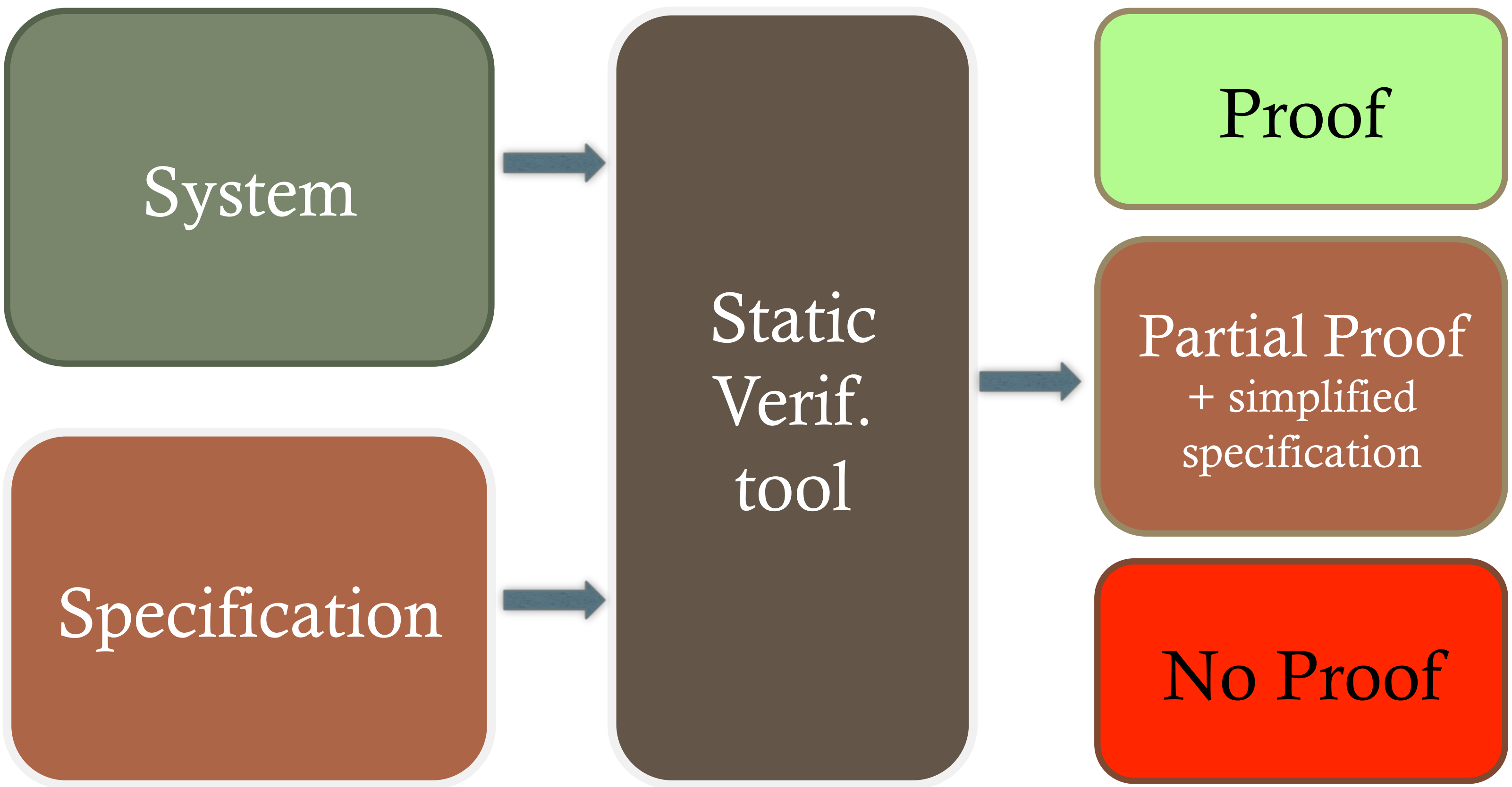
Runtime

Leaving out many other techniques, e.g. Abstract Interpretation, etc…

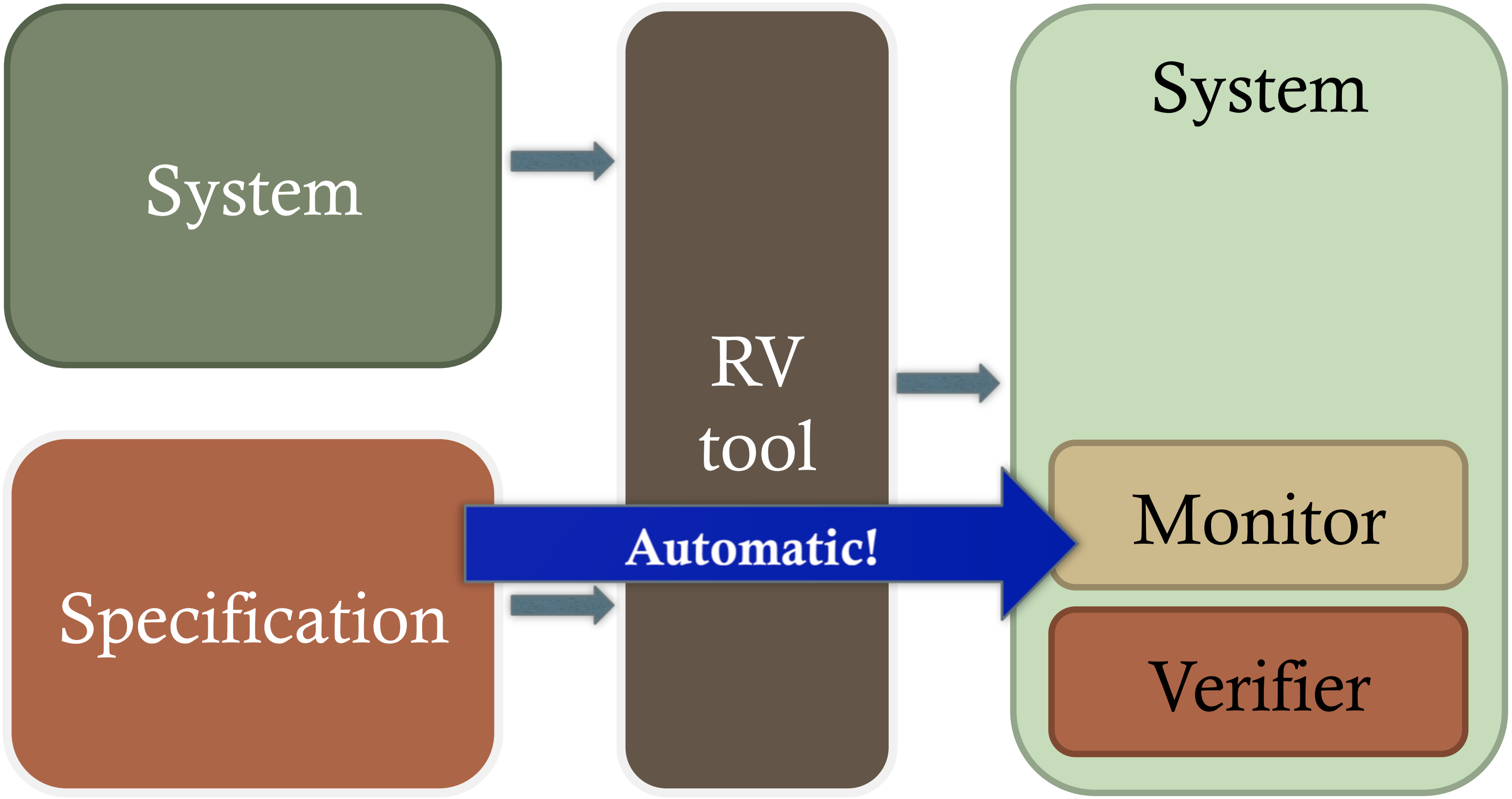
Static (Program) Verification



Static (Program) Verification



Runtime Verification



Static vs Runtime Verification

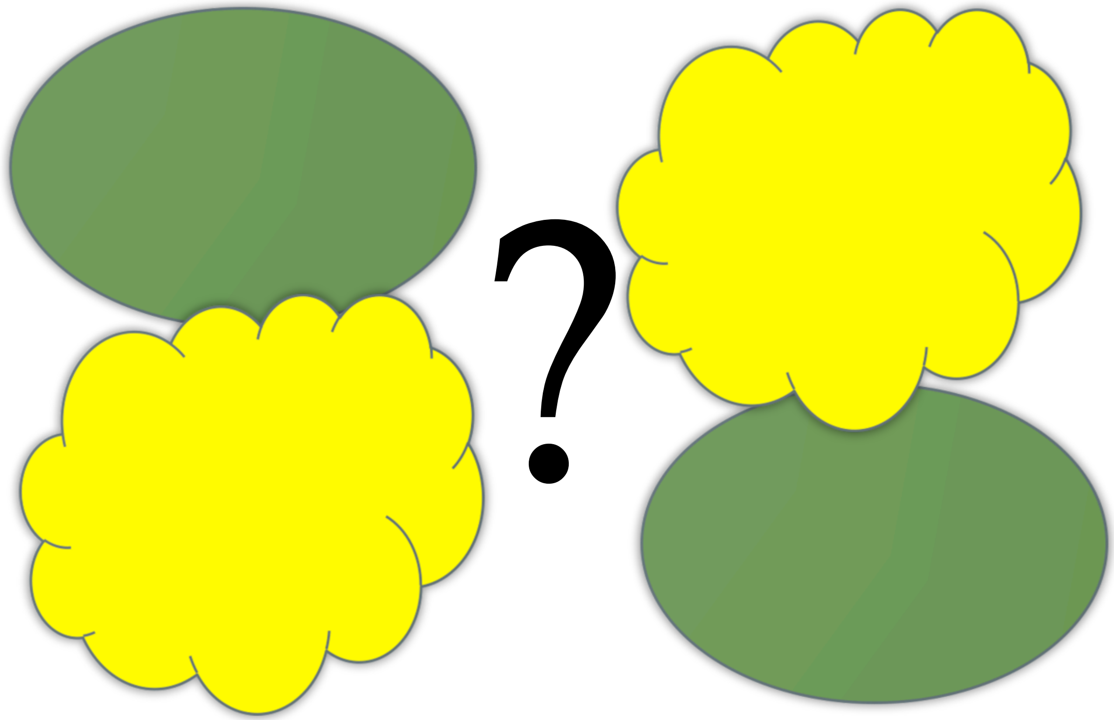
- Static verification
  - \+ Reason about properties of *all possible runs*
  - \+ High precision
  - \+ (-) Often on a model / abstractions for automation
  - – Hard to achieve full automation (e.g. invariants)
  - – Loosing aspects of concrete runs
- Runtime verification
  - \+ Full precision (for *current run*)
  - \+ Full automation (from property)
  - – Cannot judge future runs
  - – Runtime overhead

Not the same kind of properties…

Different Approaches to Different Problems…

Data-Oriented Properties

Control-Oriented Properties



Static

Runtime

Different Approaches to Different Problems…

Data-Oriented Properties

Control-Oriented Properties

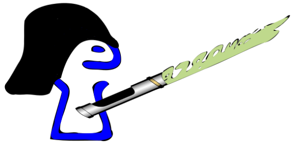
Static **Runtime**

Our Work

- Combine the best of static and dynamic verification
  - Data + Control
- Combine different techniques but not too many specification languages

How to achieve that?

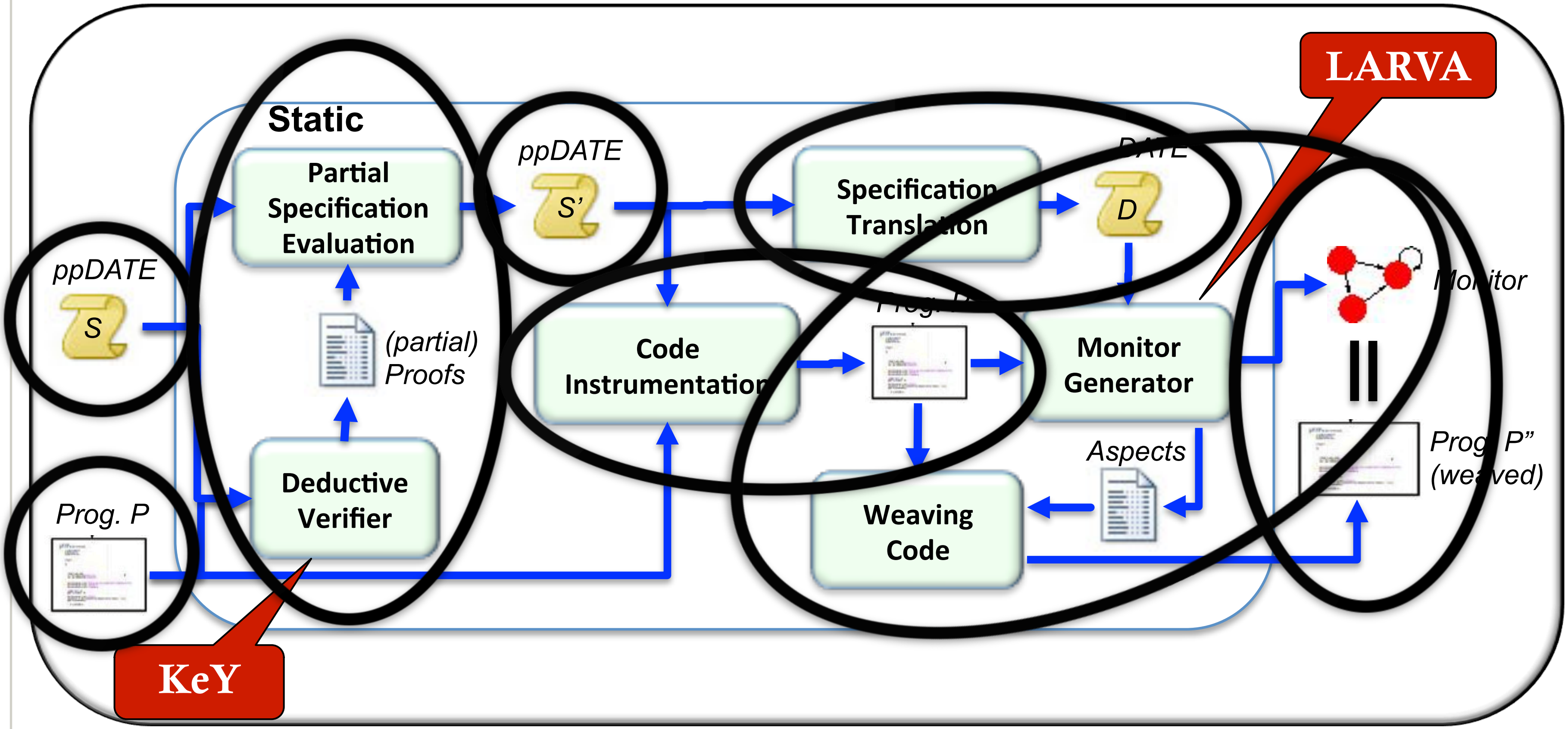
- We ask *The Force* and get it!



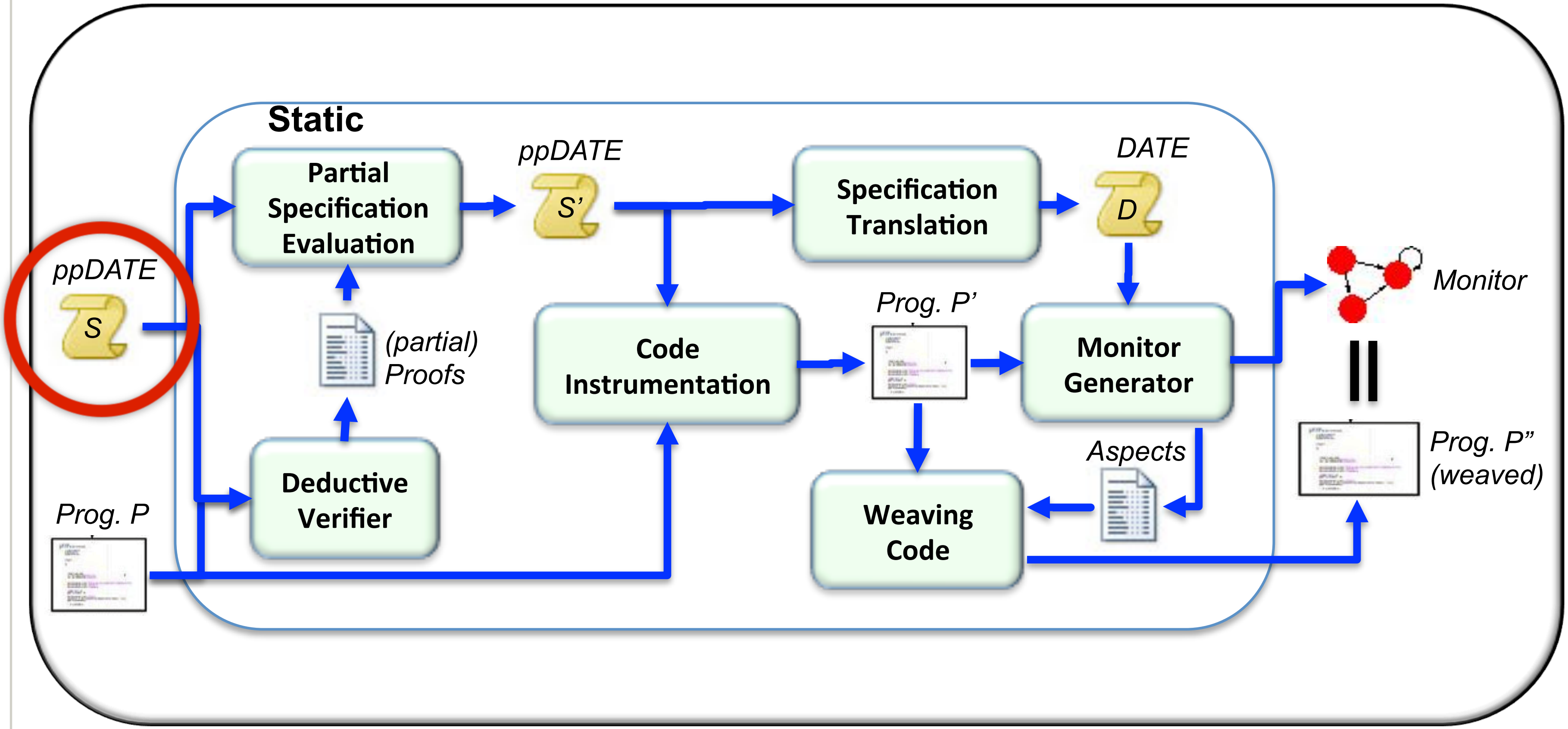
- STARVOORS : Unified Static and Runtime Verification of Object-Oriented Software
  - A specification language: ppDATE
  - A tool based on top of KeY and LARVA

Vetenskapsrådet

StaRVOOrS


LARVA
Static
Partial Specification Evaluation
ppDATE
S'
Specification Translation
D
ppDATE
S
(partial) Proofs
Code Instrumentation
Monitor Generator
Aspects
Deductive Verifier
Prog. P
Weaving Code
KeY

StaRVOOrS



Static
Partial Specification Evaluation
ppDATE
S'
Specification Translation
DATE
D
ppDATE
S
(partial) Proofs
Code Instrumentation
Prog. P'
Monitor Generator
Monitor
Deductive Verifier
Aspects
Weaving Code
Prog. P"
(weaved)
Prog. P

DATE

Dynamic Automata with Timers and Events

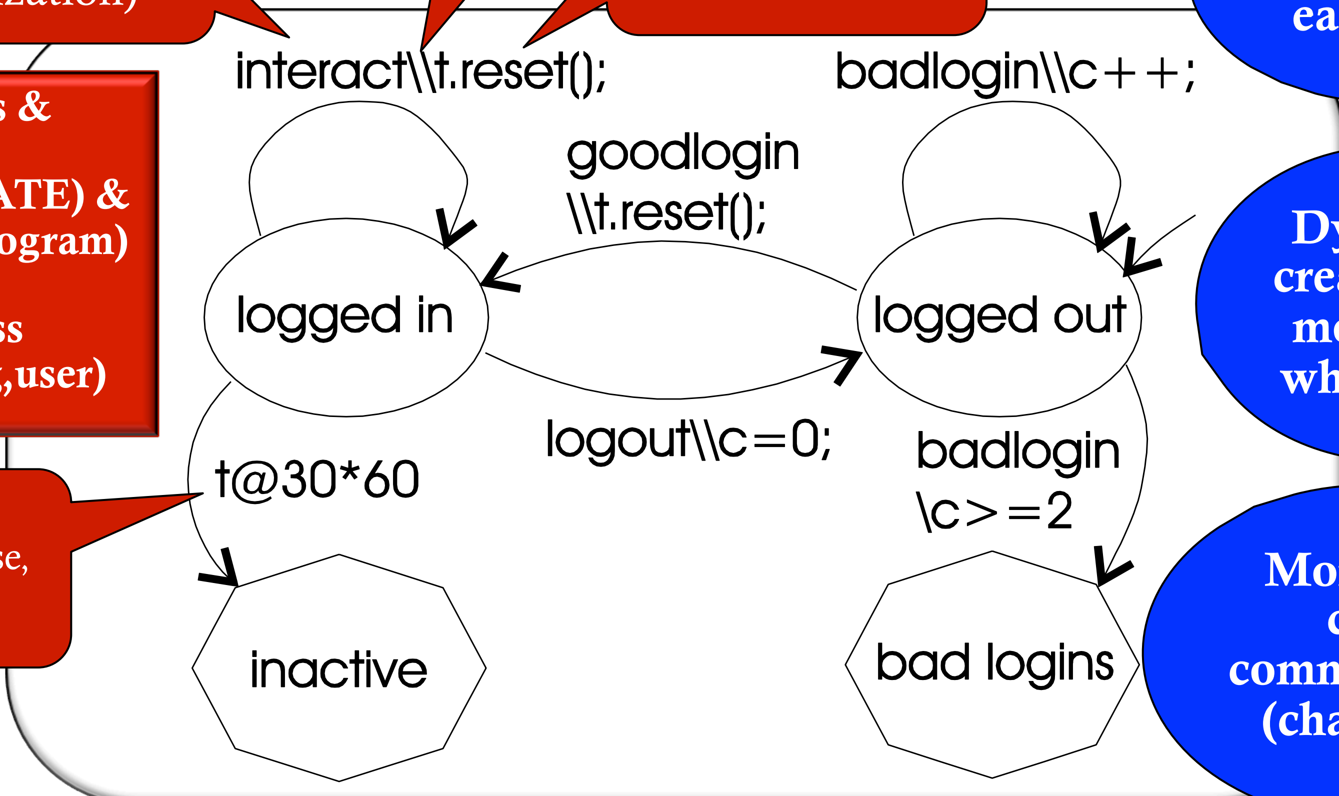
Trigger (Entry/ Exit points to code, Synchronization)

Condition

Action (create DATEs; execute Java programs)

One monitor for each user

Conditions & Actions:
- Local (DATE) & System (program) variables
- Can access context (eg,user)



Timer (reset, pause, resume)

Dynamic creation of monitors when new users

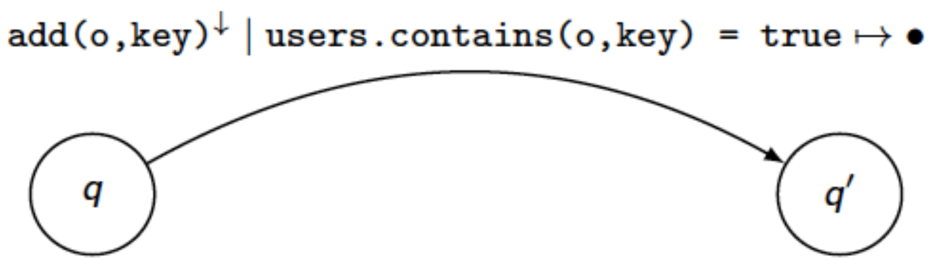
Monitors can communicate (channels)

* C. Colombo, G.J. Pace, and G. Schneider. *Dynamic event-based runtime monitoring of real-time and contextual properties*. In FMICS'08, vol 5596 of LNCS, pp 135-149, 2009

*pp*DATE

DATE with Pre/Post-conditions (roughly!)

Hoare triple associated with state *q*



$$\tau(q) = \{\, \{\texttt{users.size} < \texttt{users.capacity}\}\ \mathrm{add}\ \{\mathit{post}\}\, \}$$

$$\mathit{post} \equiv (\exists\ \texttt{int}\ i;\ i \geq 0\ \&\&\ i < \texttt{users.capacity}\,;\texttt{users.h}[i] = \texttt{o}\,;)$$

- Part of a ppDATE of adding a user in a login system

Can we write interesting properties?

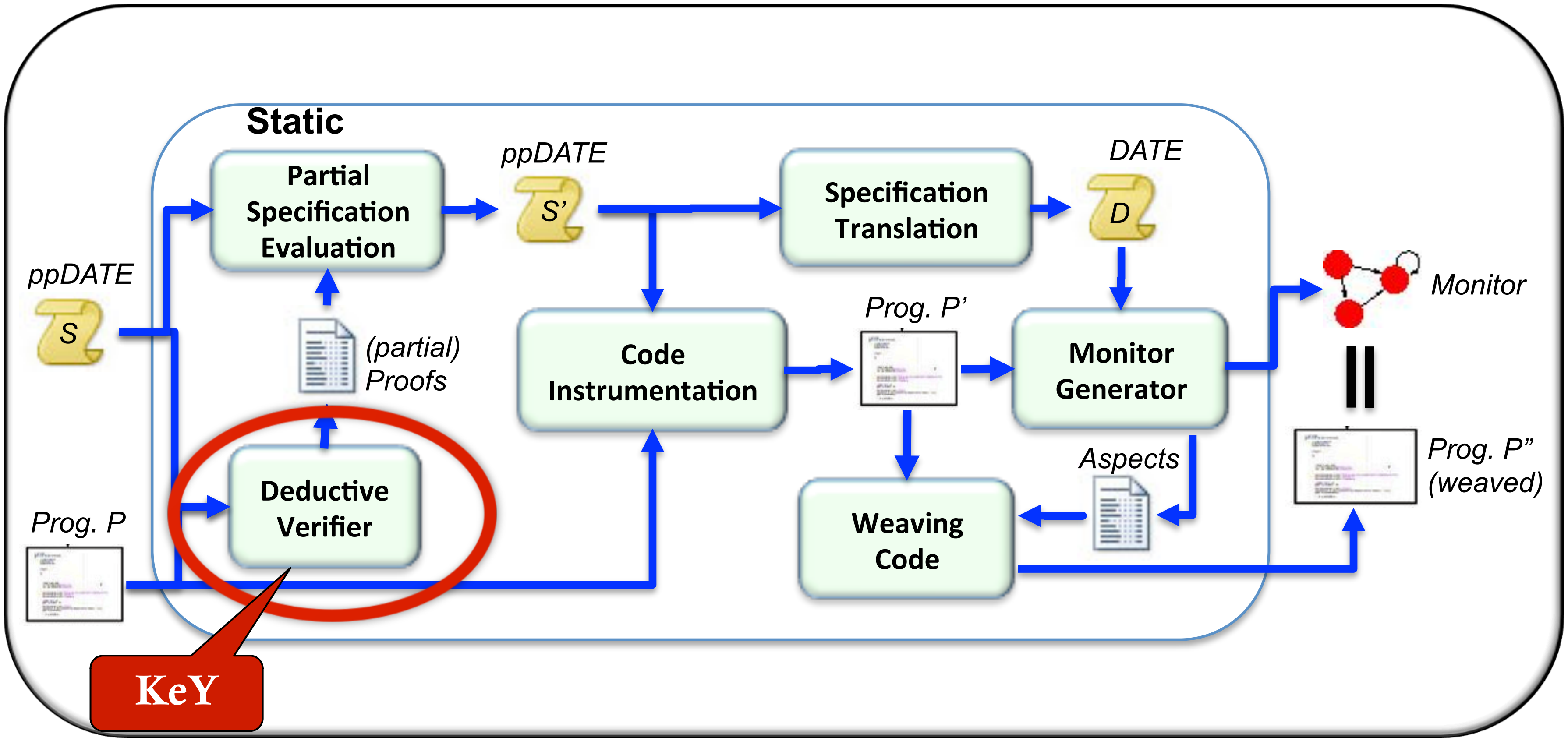
Expressiveness:

- ppDATEs are equivalent to DATES (encoding)
  - Data + Control-oriented
  - Context-dependent properties (identifiers help distinguishing different calls of a method)
  - Properties about recursive calls (matching entry/exit points of same call)
  - Real-time properties …

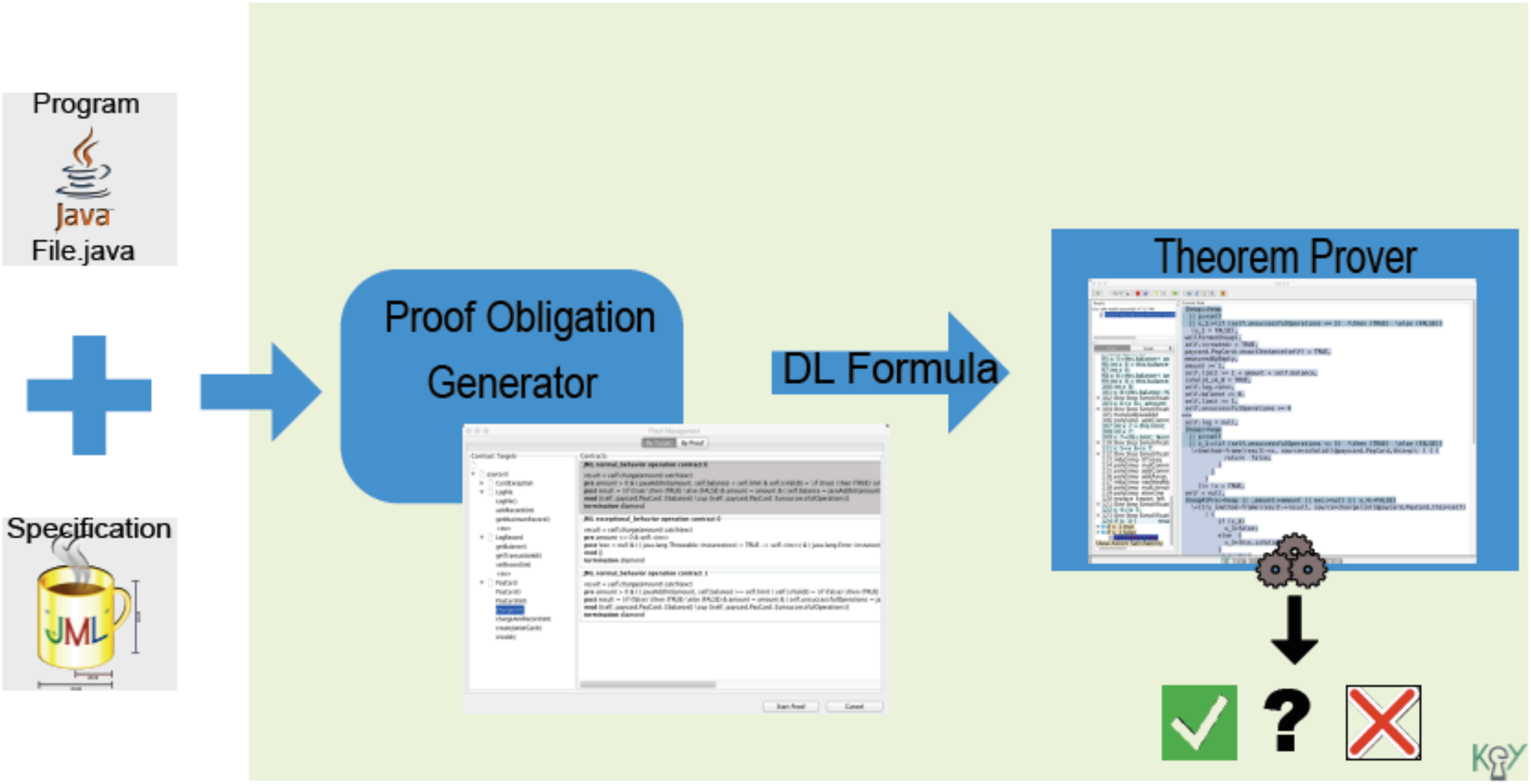
Why a new language?

- Separation of concerns between data and control
  - No need to encode event history in data
  - No need to encode data properties in automata

StaRVOOrS


Static
Partial Specification Evaluation
ppDATE
S'
Specification Translation
DATE
D
ppDATE
S
(partial) Proofs
Code Instrumentation
Prog. P'
Monitor Generator
Monitor
Deductive Verifier
Aspects
Prog. P
Weaving Code
Prog. P" (weaved)
KeY

KeY



Program
Java
File.java
Specification
JML
Proof Obligation Generator
DL Formula
Theorem Prover
?
KeY

JML Example

Precondition

Postcondition

Also: Assertions, Invariants, etc

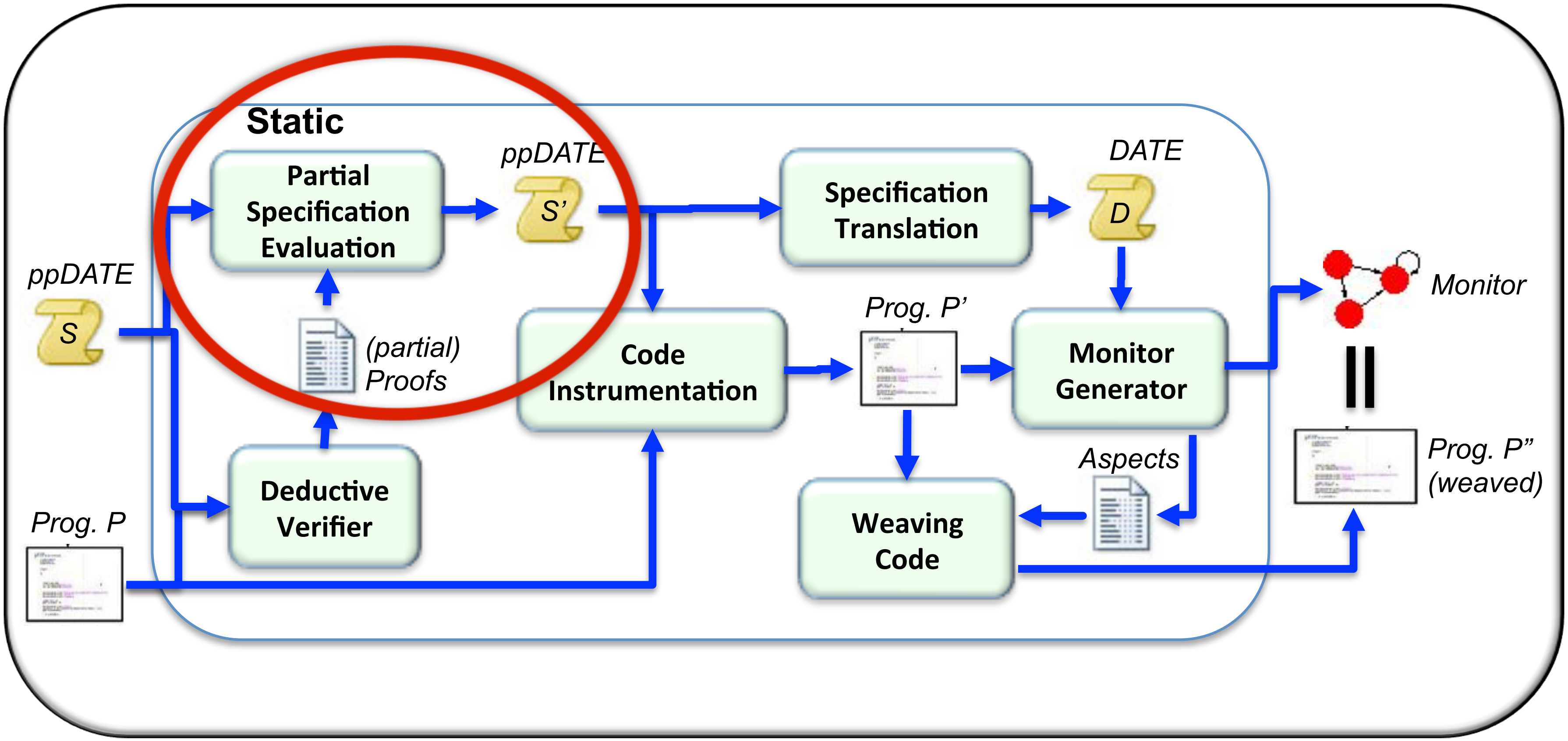
```
/*@ public normal_behavior
  @ requires a != null;
  @ ensures (\forall int j; j >= 0 && j < a.length;
  @                         \result >= a[j]);
  @ ensures a.length > 0 ==>
  @         (\exists int j; j >= 0 && j < a.length;
  @                         \result == a[j]);
  @*/
public static int max(int[] a) {
    int max = a[0], i = 1;
    while ( i < a.length ) {
      if ( a[i] > max ) max = a[i];
      ++i;
    }
    return max;
}
```

JML Translated to Java Dynamic Logic

Precondition ➔ <Prog> Postcondition

```
  a != null
->
  <
    int max = 0;
    if ( a.length > 0 ) max = a[0];
    int i = 1;
    while ( i < a.length ) {
      if ( a[i] > max ) max = a[i];
      ++i;
    }
  >
    \forall int j; (j >= 0 & j < a.length -> max >= a[j])
    &
    (a.length > 0 ->
      \exists int j; (j >= 0 & j < a.length & max = a[j]))
```

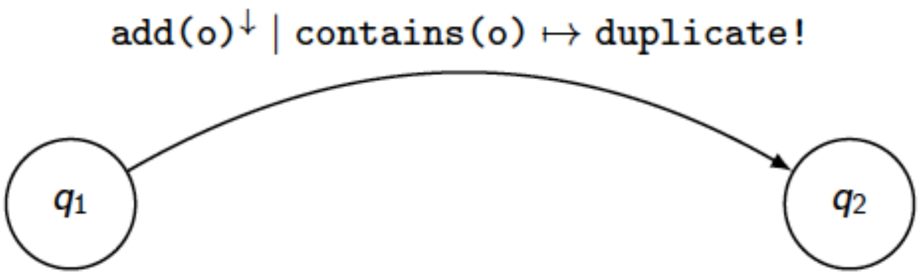
StaRVOOrS



Static
Partial Specification Evaluation
ppDATE
S'
Specification Translation
DATE
D
ppDATE
S
(partial) Proofs
Code Instrumentation
Prog. P'
Monitor Generator
Monitor
Deductive Verifier
Aspects
Weaving Code
Prog. P
Prog. P"
(weaved)

Partial Specification Evaluator

Example: Part of adding an element to an array



$$\tau(q_1) = \{ \quad \{\texttt{size} < \texttt{capacity}\}\, \texttt{add(o)}\, \{\exists\, i.\texttt{arr}[i] = \texttt{o}\} \quad \}$$

Partial Specification Evaluator

Example: Part of adding an element to an array

- KeY tries to prove:

  $$\{\mathtt{size} < \mathtt{capacity}\}\,\mathtt{add(o)}\,\{\exists\, i.\ \mathtt{arr}[i] = \mathtt{o}\}$$

- KeY cannot fully prove (automatically)
- proof branch

  $$\ldots, \mathtt{arr[key\%capacity]} = \mathtt{null} \vdash \ldots$$

  closed (automatically)
- proof branch

  $$\ldots, \neg\,\mathtt{arr[key\%capacity]} = \mathtt{null} \vdash \ldots$$

  not closed (automatically)

Partial Specification Evaluator

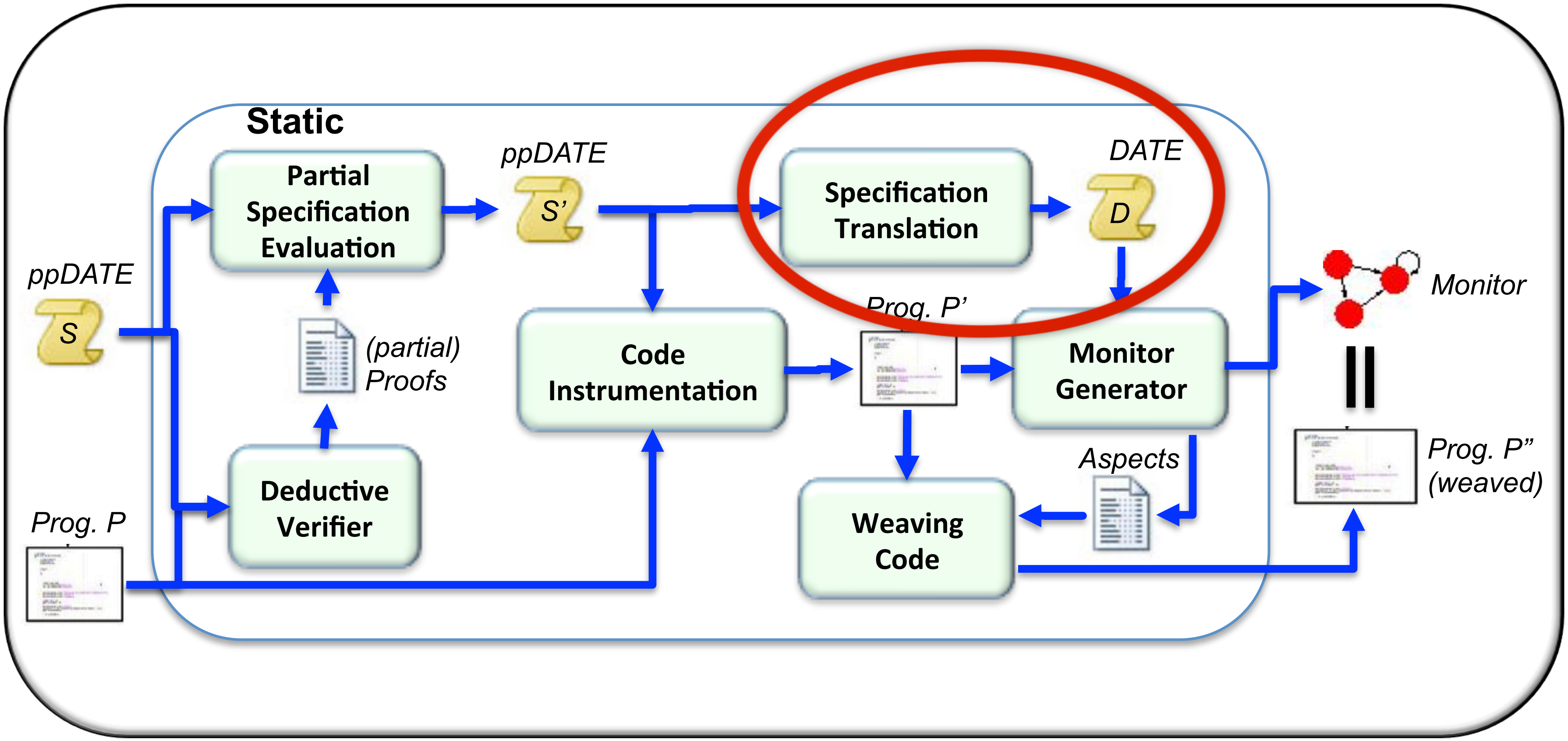
Example: Part of adding an element to an array

- partial proof analysis synthesises additional pre-conditions, here

$$\neg\, \texttt{arr[key\%capacity]} = \texttt{null}$$

$$\tau(q_1) =$$
$$\{ \quad \{\mathit{pre} \wedge \neg\, \texttt{arr[key\%capacity]} = \texttt{null}\}\, \texttt{add(o)}\, \{\mathit{post}\} \quad \}$$

StaRVOOrS



Static
Partial Specification Evaluation
ppDATE
S'
Specification Translation
DATE
D
ppDATE
S
(partial) Proofs
Code Instrumentation
Prog. P'
Monitor Generator
Monitor
Deductive Verifier
Aspects
Prog. P
Weaving Code
Prog. P"
(weaved)

Translation from ppDATE to DATE

given transition

$$q \xrightarrow{e \,\mid\, \mathit{cond} \mapsto \mathtt{act}} q'$$

and Hoare triple

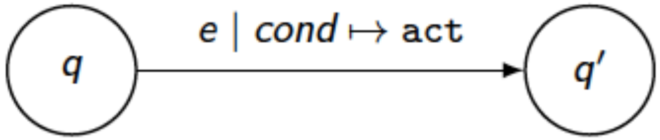
$$\tau(q) = \{\, \ldots, \{\mathit{pre}\}\, \mathtt{m}(\overline{\mathtt{a}})\, \{\mathit{post}\}, \ldots \,\}$$

There are 2 cases:

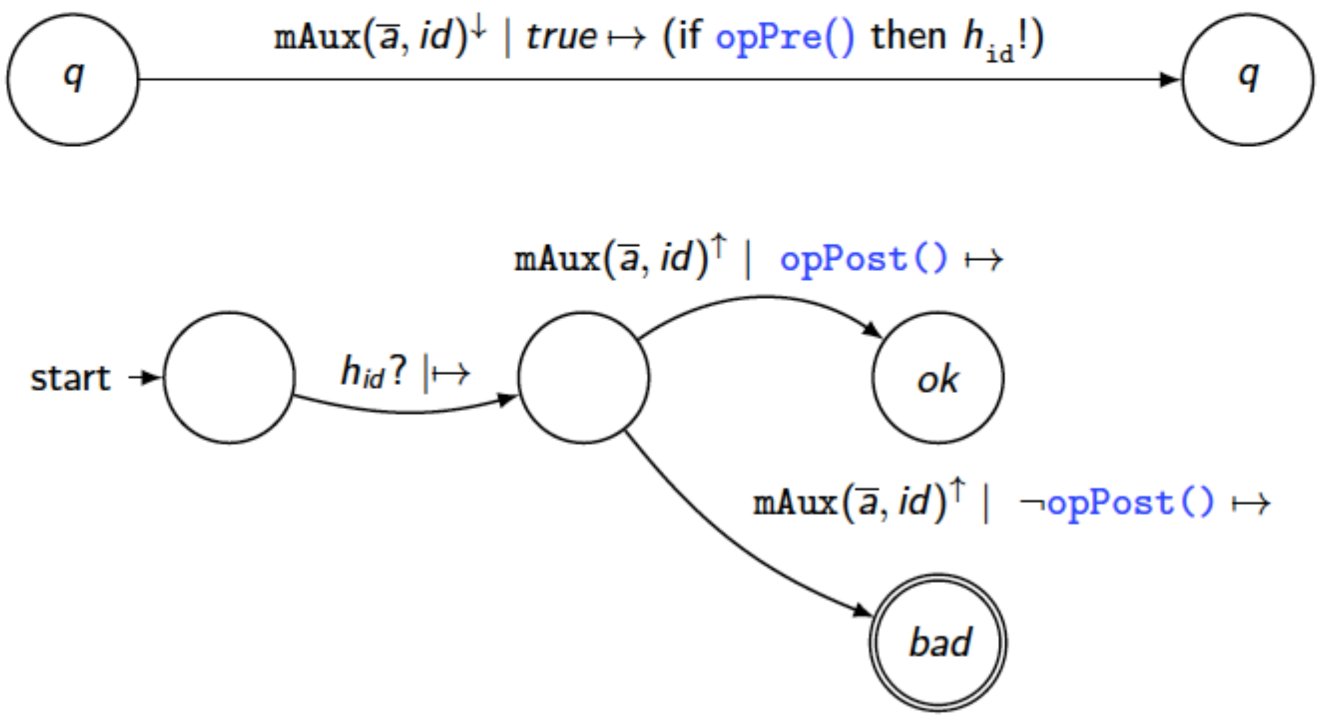
$$e \neq \mathtt{m}(\overline{a})^{\downarrow}$$

$$e = \mathtt{m}(\overline{a})^{\downarrow}$$

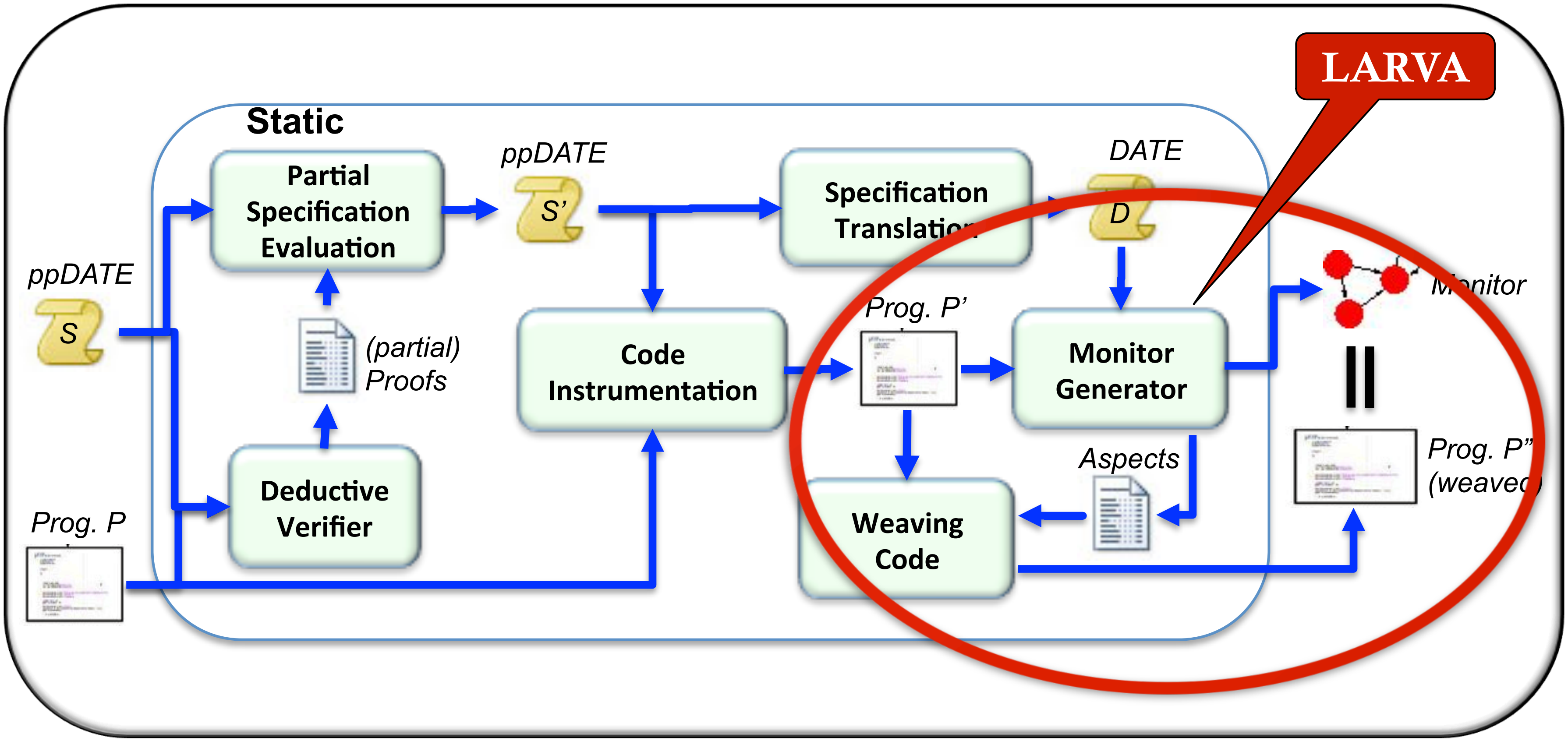
Translation from ppDATE to DATE



Plus 2 additional DATEs:



StaRVOOrS



LARVA
Static
Partial Specification Evaluation
ppDATE
S'
Specification Translation
DATE
D
ppDATE
S
(partial) Proofs
Code Instrumentation
Prog. P'
Monitor Generator
Monitor
Deductive Verifier
Aspects
Prog. P''
(weaved)
Prog. P
Weaving Code

LARVA

Monitor Generation and Runtime Verification

```
public static boolean
      add_ok_post(HashTable hasht, Object o, int key)
{ boolean r = false;
  for (int i = 0 ; i <= users.capacity - 1 ; i++) {
    if (users.h[i] == o) { r = true ; break; }
  }
  return r;
}
```

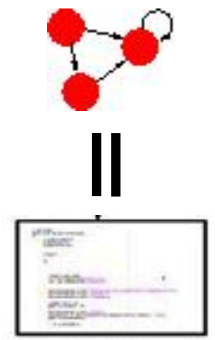
Java Program *P'*

```
EVENTS {
 add_entry(Object o,int key) = {HashTable users.add(o, key)}
}
PROPERTY add {
 STATES {  NORMAL{q2;}  STARTING{q1 (add_ok);}  }
 TRANSITIONS { q1 -> q2 [add_entry\users.contains(o, key) < 0] }
}
CINVARIANTS {
 HashTable {h.length == capacity}
 HashTable {h != null}
 HashTable {size >= 0 && size <= capacity}
 HashTable {capacity >= 1}
}
CONTRACTS {
 CONTRACT add_ok {
   PRE {size < capacity && key > 0 }
   METHOD {HashTable.add}
   POST {(\exists int i; i>= 0 && i < capacity; h[i] == o)}
   ASSIGNABLE {size, h[*]}}
}
```

DATE description *D*

LARVA (AspectJ: *P'&D*, compile)

System



Weaved Program *P''*

Besides….

- Formal semantics for ppDATEs (SOS) -> Complex!
  - Rich structure
  - Try to be close to implementation (LARVA)
- Proof of correctness of the translation ppDATEs to DATEs
  - Trace semantics (counter-examples and violating traces)
- Two case studies
  - Mondex: an electronic purse
  - SoftSlate: open source Java shopping cart web application

Conclusion

- Approach to combine static and runtime verification
  - Expressive language for data- and control-oriented
  - Verification tool
  - Formal semantics and correctness of translation

ON-GOING

- Optimize the monitor (using static analysis techniques)

FUTURE:

- Feedback from RV to improve static verification
- User-friendly interface to write properties
- Distributed setting

References

- W. Ahrendt, G.J. Pace, and G. Schneider. *A Unified Approach for Static and Runtime Verification: Framework and Applications.* In ISoLA'12, vol 7609 of LNCS, pp.312-326, 2012.

- W. Ahrendt, M. Chimento, G. Pace and G. Schneider. *A Specification Language for Static and Runtime Verification of Data and Control Properties.* In FM'15, vol. 9109 of LNCS, pp.108-125, 2015.

- W. Ahrendt, M. Chimento, G. Pace and G. Schneider. *StaRVOOrS: A Tool for Combined Static and Runtime Verification of Java.* In *RV'15, vol. 9333 of LNCS, pp.297-305, 2015.*

- W. Ahrendt, G. Pace, and G. Schneider. *Starvoors - Episode II, Strengthen and Distribute the Force.* In ISoLA'16, vol 9952 of LNCS, pp.402-415, 2016.

- W. Ahrendt, M. Chimento, G. Pace and G. Schneider. *Combined Static and Runtime Verification of Data- and Control-Oriented Properties.* Submitted

http://www.cse.chalmers.se/~chimento/starvoors

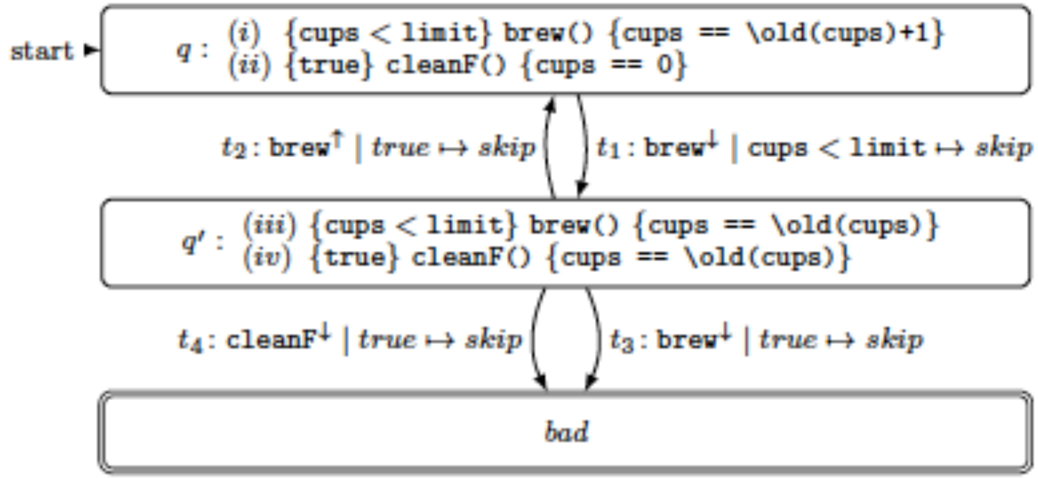
Questions?

Thank
You
Vinaka
köszönöm
Kiitos
Maake
Asante
Shukria
Dhanyavadagalu
Manana
Dankon
Terima Kasih
Matondo
감사합니다
Dankscheen
спасибо
Kam Sah Hammida
Mauruuru
Biyan
Diolch i Chi
Taiku
Tack
Dank Je
Blagodaram
Ngiyabonga
Dziekuje
Chokrane
Grazie
Juspaxar
Arigato
Mochchakkeram
Gracias
Tingki
நன்றி
Bedankt
Dakujem
धन्यवाद
cảm ơn bạn
Paldies
Gratias Tibi
Ua Tsaug Rau Koj
Grazas
Di Ou Mèsi
Kia Ora
Kop Khun Khap
Obrigado
Děkuji
Nirringrazzjak
ありがとう
Suksama
Matur Nuwun
Rahmat
谢谢
хвала
Hvala
Welalin
Djiere Dieuf
Misaotra
Danke
Merci
Go Raibh Maith Agat
Eskerrik Asko
Najis Tuke
Salamat
ขอขอบคุณคุณ

Auxiliary Slides

Context-Dependency in ppDATEs

A coffee machine



- Hoare triples makes no reference to the state of the machine (there is no info about whether the machine is active or not)
- The state of the machine is implicitly defined by the states of the ppDATE
- If the ppDATE is in state q, the machine is not active.
- If it is in state q', then it is active.
- On each state the Hoare triples are context-dependent
- This is why we can describe properties with the same precondition, but with different post-conditions, depending on which state of the ppDATE they are

*pp*DATE Definition LARVA Script

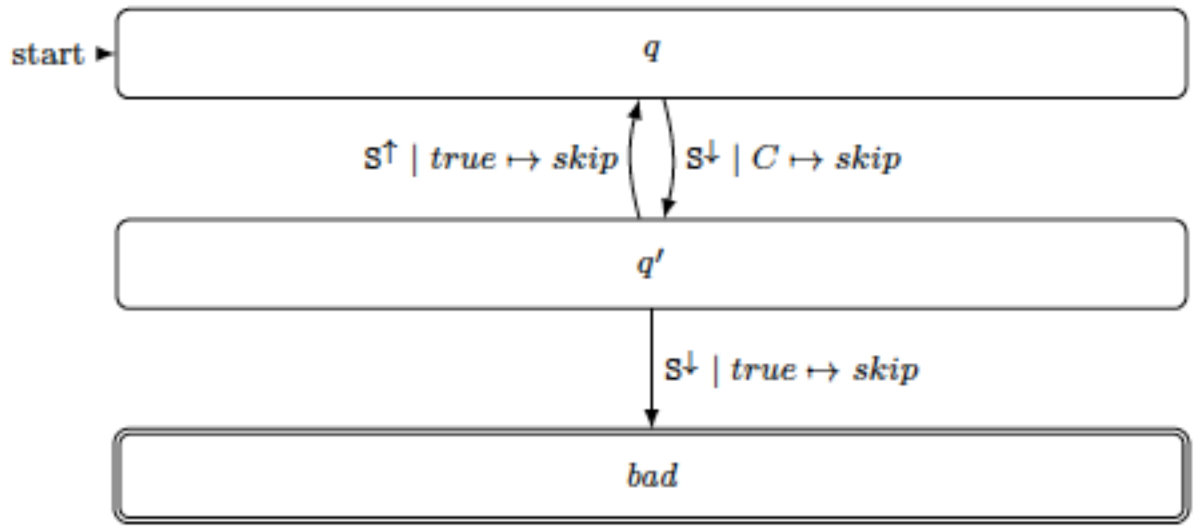
```
IMPORTS { main.UserInterface ; main.Hashtable ; }

GLOBAL {
  PROPERTY prop-deposit {
      PINIT { (prop-deposit-temp, UserInterface) }
  }
}
TEMPLATES {
 TEMPLATE prop-deposit-temp (UserInterface uf) {
   TRIGGERS {
     login_exit(String un, int pwd)
        = {UserInterface f.login(un, pwd)exit()} where {uf = f}
     logout_entry()
        = {UserInterface f.logout()entry} where {uf = f}
     deposit_entry(int val)
        = {UserInterface f.deposit(val)entry} where {uf = f}
   }
   PROPERTY prop_deposit {
     STATES {
       ACCEPTING { q2 ; }
       BAD { bad ; }
       STARTING { q1 (add_ok) ; }
     }
     TRANSITIONS {
       q1 -> q2 [login_exit \ f.getUser() != null]
       q1 -> bad [deposit_entry]
       q2 -> q1 [logout_entry \ f.getUser() != null ]
       q2 -> q2 [deposit_entry \ f.getUser() != null]
     }
   }
  }
}
```

```
CINVARIANTS {
  HashTable {\typeof(h) == \type(Object[])}
  HashTable {arr.length == capacity}
  HashTable {arr != null}
  HashTable {size >= 0 && size <= capacity}
  HashTable {capacity >= 1}
}
HTRIPLES {
  HT add_ok {
    PRE {size < capacity}
    METHOD {Hashtable.add}
    POST  {(\exists int i; i>= 0 && i < capacity; arr[i] == o)}
    ASSIGNABLE {size, arr[*]}
  }
}
```

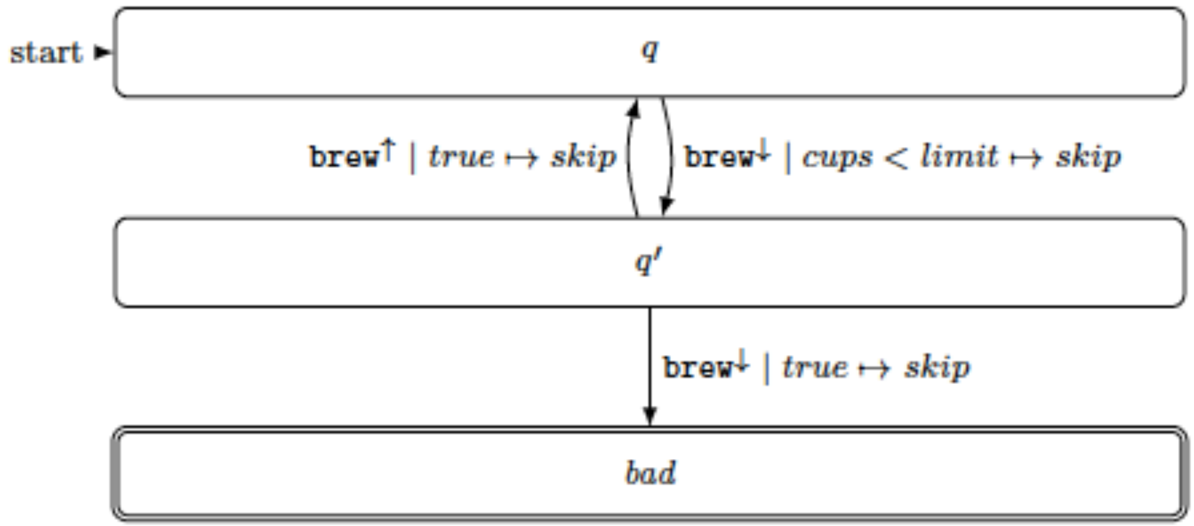
*pp*DATE Templates
Definition

$$one\text{-}at\text{-}a\text{-}time = \lambda\ C, S : cond, trigger.$$



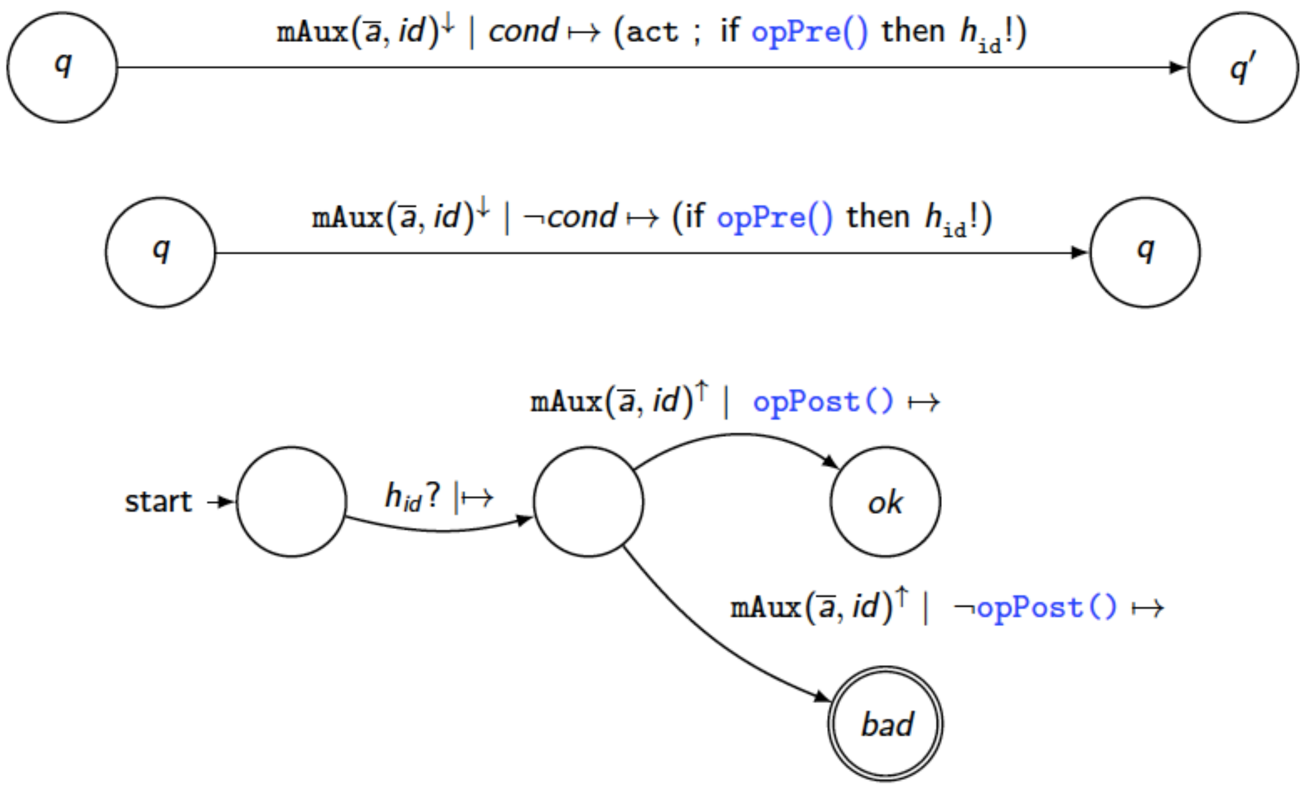
*pp*DATE Templates
Instantiation

$inst(one\text{-}at\text{-}a\text{-}time, cups < limit, \texttt{brew}) =$



Translation from ppDATE to DATE

DATE : if $e = \mathtt{m}(\bar{a})^{\downarrow}$



Case Study: SoftSlate Commerce
(Shopping cart web application)

LOGIN - LOGOUT

(i) A user has to be logged in the application in order to perform a purchase, i.e., the checkout of a purchase can only happen between a login and a logout.

(ii) If a user is logged in, then that user cannot successfully log in again in the application until she logs out from it.

(iii) If a user is not logged-in, then that user cannot successfully log out from the application.

(iv) A user can only proceed to the checkout section if her status is a valid one.

(v) A user who is not a costumer cannot proceed to the checkout section.

PURCHASE CHECKOUT

(1) The checkout of a purchase should be performed following the four required steps.

(2) It is not be possible to buy zero or less items.

(3) The expiration date of the credit card should not earlier than the current date.

(4) The price of a product should be positive.

(5) Before a purchase is completed, taxes should be processed.

(6) The total cost should be equal to the sum of the prices of all the products to be purchased.

(7) If the price of an item changes, then its price in the order of the user should be updated.

Case Study: SoftSlate Commerce
(Shopping cart web application)

- Found a strange design decision: each user associated with one session generated two instances of class User for a given real user (prop (iii) thus violated)
- Violation of property (4)
- Violation of property (7): prices modified by administrator propagated to DB but not the user cart

Purchases	(a) no monitoring	(b) monitoring S	(c) monitoring S'
1	800 ms	1,300 ms	1,100 ms
10	10,500 ms	15,500 ms	13,000 ms
100	120,000 ms	190,000 ms	150,000 ms

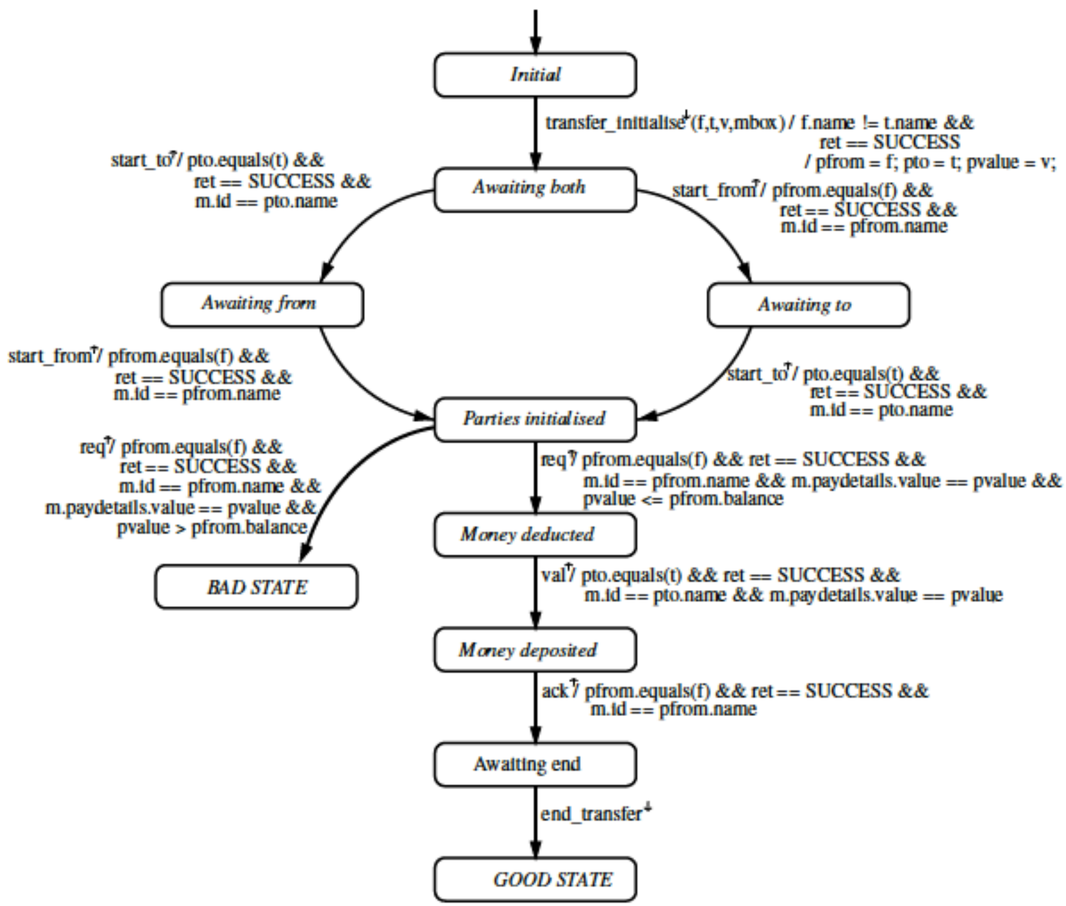
Case Study: Mondex
(An electronic purse application)

- ppDATE: 10 states and 25 transitions
- 25 DATEs: 106 states and 196 transitions

Transactions	*no* monitoring	monitoring without static verif.	monitoring using static verif.
10	8 ms	120 ms	15 ms
100	50 ms	3,500 ms	90 ms
1000	250 ms	330,000 ms	375 ms

- Overhead: Postcondition monitoring
- KeY proves 2 Hoare triples fully -> **not** checked at runtime
- KeY proves 24 Hoare triples partially -> *conditionally* checked at runtime
- Why the gain? Preconditions were false -> no postcondition checking

ppDATE for Mondex



ppDATE

*pp*DATE
Formally…

Set of *states*

A function tagging each state with
Hoare triples
$\Pi \in Q \longrightarrow \mathcal{P}(cond_{Sys} \times \Sigma \times postcond_{Sys})$

$$(Q, t, B, q_0, \Pi)$$

Initial state

Transition relation
$t \subseteq Q \times trigger \times cond_{Sys \cup V} \times action \times Q.$

Set of *bad states*

ppDATE
Transitions

$$trigger ::= systemtrigger \mid \mathsf{actevent?}$$

$$systemtrigger ::= \mathsf{methodname}^{\downarrow} \mid \mathsf{methodname}^{\uparrow}$$

$$action ::= \mathsf{skip} \mid v = e \mid \mathsf{actevent!} \mid \mathsf{create}(template, \overline{args}) \mid action \ ; action \mid \mathsf{if}\ cond_{Sys \cup V}\ \mathsf{then}\ action \mid \mathsf{Program}$$

Conditions are BJMLE (Boolean JML Expressions)

- For the sake of presentation just think of them as normal boolean conditions

Terminating, side-effect free
(no system events, no writing on system variables)

*pp*DATE

BJMLE

- any side-effect free Boolean *Java* expression is a BJMLE,
- if `a` and `b` are BJMLEs, and `x` is a variable of type `t`, the following expressions are BJMLEs:
  - `!a`, `a&&b`, and `a||b`
  - `a ==> b` ("`a` implies `b`")
  - `a <==> b` ("`a` is equivalent to `b`")
  - `(\forall t x; a)` ("for all `x` of type `t`, `a` holds")
  - `(\exists t x; a)` ("there exists `x` of type `t` such that `a`")
  - `(\forall t x; a; b)` ("for all `x` of type `t` fulfilling `a`, `b` holds")
  - `(\exists t x; a; b)` ("there exists an `x` of type `t` fulfilling `a`, such that `b`")
- replacing any sub-expression `e` in a BJMLE with `\old(e)` gives a BJMLE,
- replacing any sub-expression in a BJMLE with `\result` gives a BJMLE, (well-typedness is context dependent, see Def.5)

*pp*DATE Network

Set of ppDATES

Set of ppDATE templates

$$pn = (M, V, \nu_0, T_{ppd})$$

Templates are created with the action *create*

Set of ppDATE (global) variables

Initial valuation of variables in *V*

*pp*DATE
Formal Semantics

Will not go into details…

- SOS semantics - Complex!
- Rich structure
  - Communicating "automata" (channel broadcasting)
  - Program (*system*) and monitor (*ppDATE*) variables
  - Actions are arbitrary programs with side effects
  - Dynamic creation of ppDATEs (*templates*)
- [Try to be] close to the implementation (LARVA)

*pp*DATE
Semantics

Every time the system generates an event (entry or exit of a method):

- All ppDATEs with enabled transitions execute the associated actions, *simultaneously*
- Action events (*h!*) will be stored in a buffer
- After all enabled transitions are fired, every transition becoming enabled by events in the buffer, are fired
- The buffer is emptied and the procedure is repeated until no more transitions are enabled

*pp*DATE
Semantics

- Small steps for local configurations
- Big steps for global configurations

*pp*DATE
Local Configurations

- Given a ppDATE *m*, a *local configuration* is a tuple (m, q, ρ)
  - q is the current state
  - ρ allows to monitor potential violations of Hoare triples
    - Stores which exit event (*systemevent*) should cause a check of which postconditions, under the given system variable valuation

*pp*DATE
Big Step Semantics for Global Configurations

- Given a ppDATE network $pn = (M, V, \nu_0, T_{ppd})$ a *global configuration* is a tuple (L, ν) such that:
  - L is the set of *local configurations*
  - ν is a *ppDATE variable valuation* with domain V

Extended global configuration

Transitive closure of relation *small step global*

$$\textit{shift}\ \frac{(L, \nu, \{e\}, \theta) \rightarrowtail^* (L', \nu', \emptyset, \theta)}{(L, \nu) \xRightarrow{(e,\theta)} (L', \nu')}$$

(system event, system variable valuation)

*pp*DATE
Small Step Rule for Extended Global Configurations

All local conf. s.t. *m* has enabled transition whose trigger is activated by events in *E*

Local conf. that will not change

Local conf. changing after small step of L_{en}

$$
iter \frac{
\begin{array}{c}
L_{en} = \{l \mid l \in L, enabled(l, e, \theta, \nu), e \in E\} \\
L_{nch} = L \backslash L_{en} \\
L_{ch} = \{l' \mid l \in L_{en}, l \xrightarrow{(e,\theta,\nu)} l', e \in E\} \\
Acts = \{a \mid l \in L_{en}, toBeExecuted(l, e, \theta, \nu, a), e \in E\} \\
mergeParalActs_{\nu}(\{[\![a]\!]_{\theta,\nu} | a \in Acts\}) = (\nu', E', New') \\
L_{new} = \{(m, q_{0m}, \emptyset) \mid m \in New'\} \\
L' = L_{ch} \cup L_{nch} \cup L_{new}
\end{array}
}{(L, \nu, E, \theta) \rightarrowtail (L', \nu', E', \theta)}
$$

Execute all actions to be executed in parallel. If undefined (conflict in parallel read/write) then execution aborts

*pp*DATE
Small Step Rules for Local Configurations

$$entry_1 \; \frac{\begin{array}{c} checkOnExit((m,q,\rho),\sigma^{\downarrow}_{id},\theta,\pi') \\ nextState((m,q,\rho),\sigma^{\downarrow}_{id},\theta,\nu,q') \end{array}}{(m,q,\rho) \xhookrightarrow{(\sigma^{\downarrow}_{id},\theta,\nu)} (m,q',\rho \cup \{(\sigma^{\uparrow}_{id},\pi',\theta)\})}$$

$$entry_2 \; \frac{\begin{array}{c} \nexists\, \pi' \cdot checkOnExit((m,q,\rho),\sigma^{\downarrow}_{id},\theta,\pi') \\ nextState((m,q,\rho),\sigma^{\downarrow}_{id},\theta,\nu,q') \end{array}}{(m,q,\rho) \xhookrightarrow{(\sigma^{\downarrow}_{id},\theta,\nu)} (m,q',\rho)}$$

$$entry_3 \; \frac{\begin{array}{c} checkOnExit((m,q,\rho),\sigma^{\downarrow}_{id},\theta,\pi') \\ \nexists\, q' \cdot nextState((m,q,\rho),\sigma^{\downarrow}_{id},\theta,\nu,q') \end{array}}{(m,q,\rho) \xhookrightarrow{(\sigma^{\downarrow}_{id},\theta,\nu)} (m,q,\rho \cup \{(\sigma^{\uparrow}_{id},\pi',\theta)\})}$$

$$exit \; \frac{nextState((m,q,\rho),\sigma^{\uparrow}_{id},\theta,\nu,q')}{(m,q,\rho) \xhookrightarrow{(\sigma^{\uparrow}_{id},\theta,\nu)} (m,q',\rho)}$$

$$act \; \frac{\begin{array}{c} e \in \textsf{actevent} \\ nextState((m,q,\rho),e,\theta,\nu,q') \end{array}}{(m,q,\rho) \xhookrightarrow{(e,\theta,\nu)} (m,q',\rho)}$$