Runtime Verification of Component-based Systems

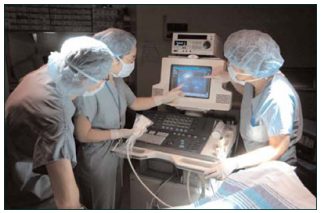
Yliès Falcone[1], Mohamad Jaber[2], Thanh-Hung Nguyen[2], Marius Bozga[2], and Saddek Bensalem[2]

[1]LIG – [2]Verimag
-
Université Joseph Fourier

SEMBA - Valence - October 21, 2011

Challenges

Computer systems are everywhere

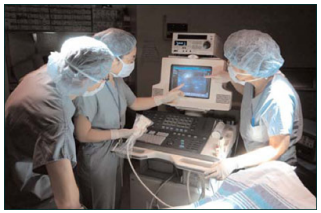




Bugs too!!

How to build complex systems that behave in a correct manner?

Challenges

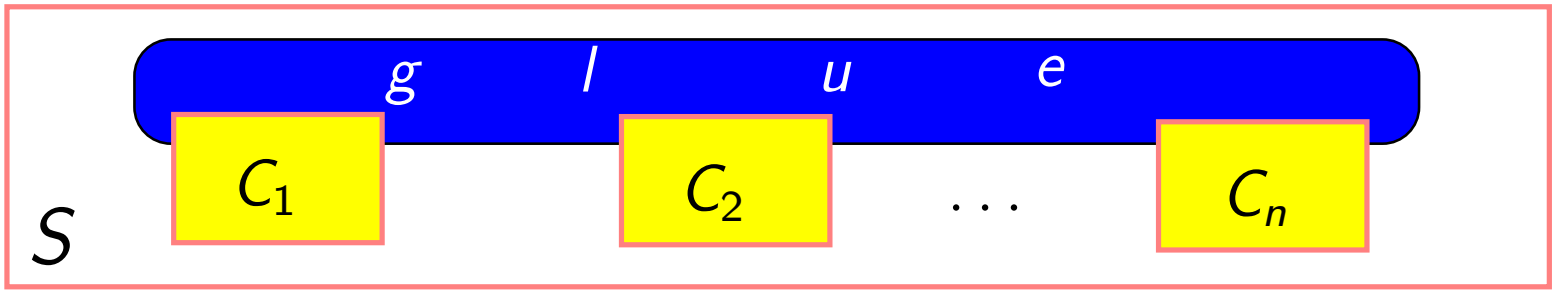
Computer systems are everywhere


Bugs too!!

How to build complex systems that behave in a correct manner?

Component-based Approach for System Design

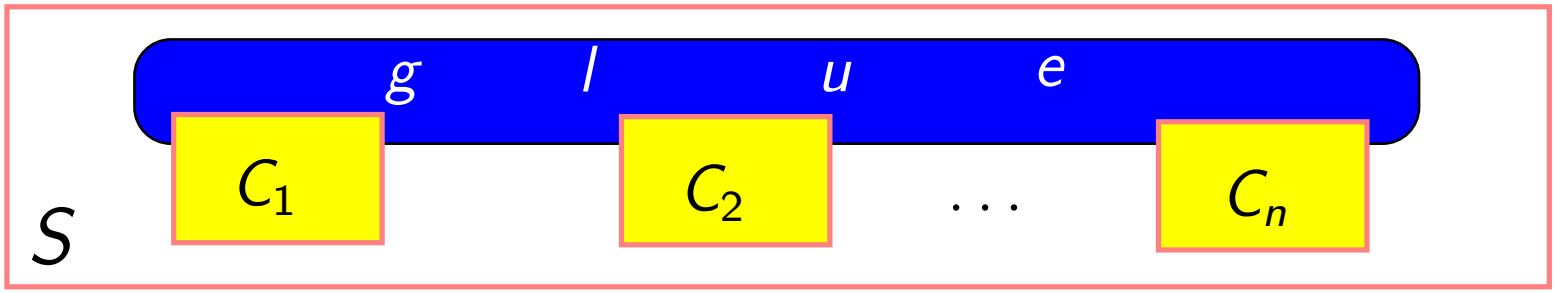


Principles:

- builds complex systems by assembling components
- reuses existing components, known properties
- provides flexibility in the construction phase
- helps to cope with complexity

How to verify/check : $S = glue(C_1, \ldots, C_n) \models P?$

Component-based Approach for System Design



$\models P?$

Principles:

- builds complex systems by assembling components
- reuses existing components, known properties
- provides flexibility in the construction phase
- helps to cope with complexity

How to verify/check : $S = glue(C_1, \ldots, C_n) \models P?$

How to Ensure/Check Correctness?

Static Verification Techniques

Mathematical techniques to prove or disprove the correctness of a design w.r.t. a given property
(e.g., static analysis, model checking, . . .)

Limitations (for Component-Based Systems)

- State space explosion!
- Black box component
- Interaction with unknown environment (x = read();)

Adopted Solution: Runtime Verification (RV)

- Checking P while the system is running
- "Bridges the gap" between static verification and testing

How to Ensure/Check Correctness?

Static Verification Techniques

Mathematical techniques to prove or disprove the correctness of a design w.r.t. a given property
(e.g., static analysis, model checking, ...)

Limitations (for Component-Based Systems)

- State space explosion!
- Black box component
- Interaction with unknown environment (x = read();)

Adopted Solution: Runtime Verification (RV)

- Checking P while the system is running
- "Bridges the gap" between static verification and testing

How to Ensure/Check Correctness?

Static Verification Techniques

Mathematical techniques to prove or disprove the correctness of a design w.r.t. a given property
(e.g., static analysis, model checking, . . .)

Limitations (for Component-Based Systems)

- State space explosion!
- Black box component
- Interaction with unknown environment (x = read();)

Adopted Solution: Runtime Verification (RV)

- Checking P while the system is running
- "Bridges the gap" between static verification and testing

Outline

1. The BIP Framework
2. An RV Framework for Component-Based Systems
3. Verifying the Runtime Behavior of BIP Systems
4. Experimental Results
5. Discussion

Outline

1. The BIP Framework
2. An RV Framework for Component-Based Systems
3. Verifying the Runtime Behavior of BIP Systems
4. Experimental Results
5. Discussion

The BIP Framework

BIP is a component framework for modeling heterogeneous systems

Layered Component Model

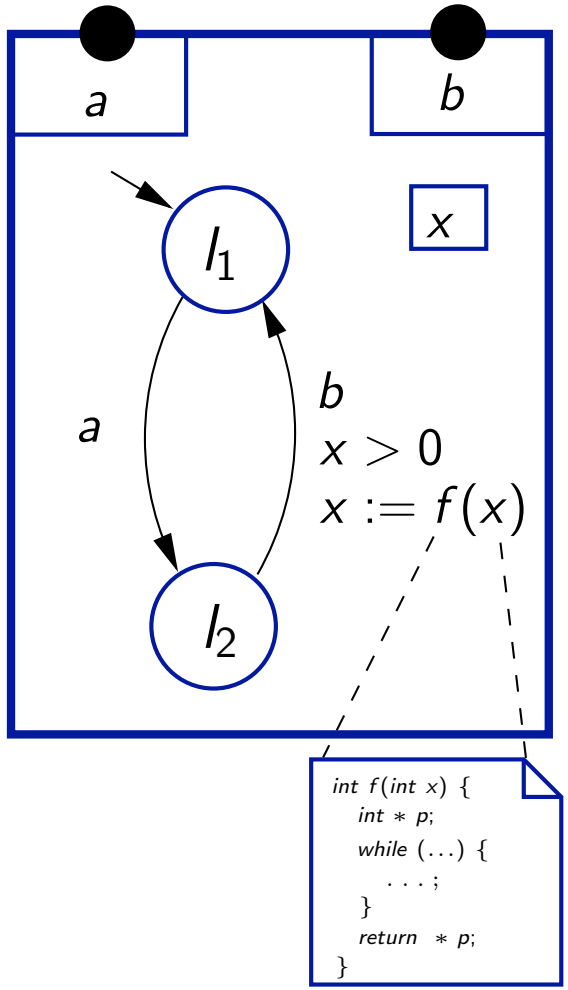
- **Behavior** - automata extended with data and communication ports
- **Interactions** - set of interactions
- **Priorities** - partial order on interactions

Behavior

Atomic Component

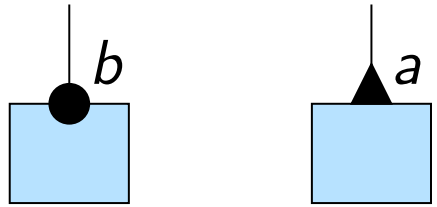
Labelled Transition System with data:

- *ports*, e.g, $\{a, b\}$
- *control locations*, e.g, $\{l_1, l_2\}$
- *variables*, e.g, $\{x\}$
- *transitions*
  - guards, e.g., $x > 0$
  - variable modifications (with external functions), e.g., $x := f(x)$



Interactions and Connectors: **Communication**

Ports can be either *synchron* (•) or *trigger* (▲)



Interaction (= "a communication/collaboration")

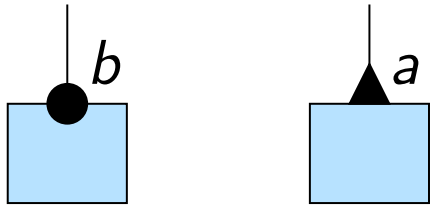
An interaction is defined as a set of ports

Connector

A connector is used to specify a set of interactions

Interactions and Connectors: **Communication**

Ports can be either *synchron* (•) or *trigger* (▲)



Interaction (= "a communication/collaboration")

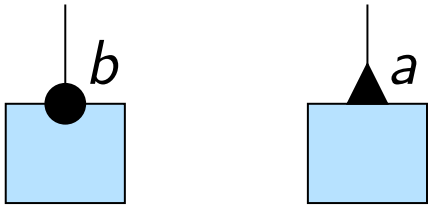
An interaction is defined as a set of ports

Connector

A connector is used to specify a set of interactions

Interactions and Connectors: **Communication**

Ports can be either *synchron* (•) or *trigger* (▲)

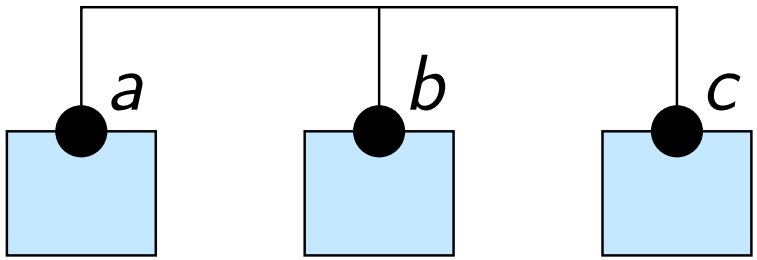


Interaction (= "a communication/collaboration")

An interaction is defined as a set of ports

Connector

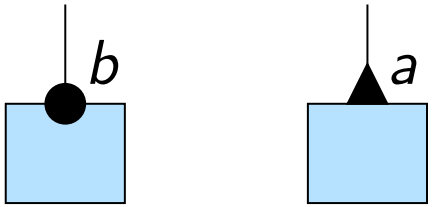
A connector is used to specify a set of interactions



RendezVous, the only possible interaction is: *abc*

Interactions and Connectors: **Communication**

Ports can be either *synchron* (•) or *trigger* (▲)

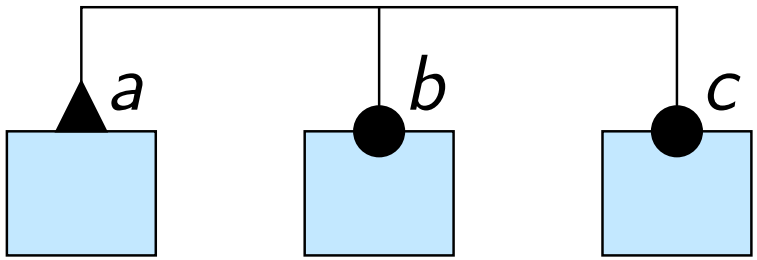


Interaction (= "a communication/collaboration")

An interaction is defined as a set of ports

Connector

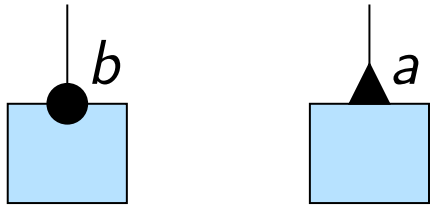
A connector is used to specify a set of interactions



Broadcast, possible interactions are: *a*,*ab*,*ac*,*abc*

Interactions and Connectors: **Communication**

Ports can be either *synchron* (•) or *trigger* (▲)

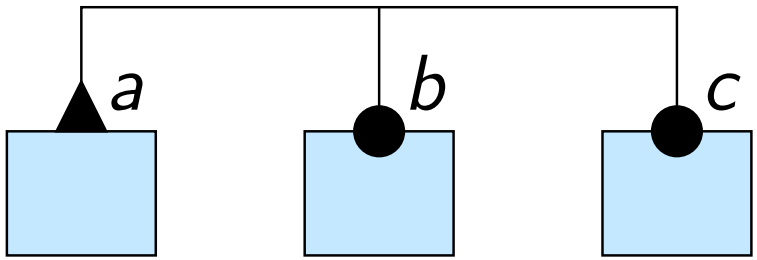


Interaction (= "a communication/collaboration")

An interaction is defined as a set of ports

Connector

A connector is used to specify a set of interactions



Bliudze and Sifakis

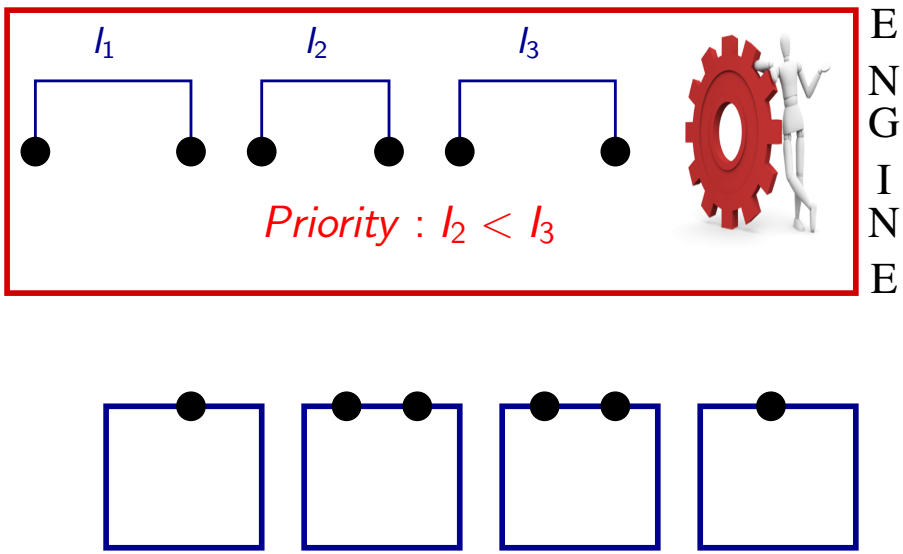
- Strong formalization of the Algebra of Connectors
- Interactions and priorities encompass the universal glue

Broadcast, possible interactions are: *a*,*ab*,*ac*,*abc*

Composite component

Composite component

available atomic components + connectors + priority rules



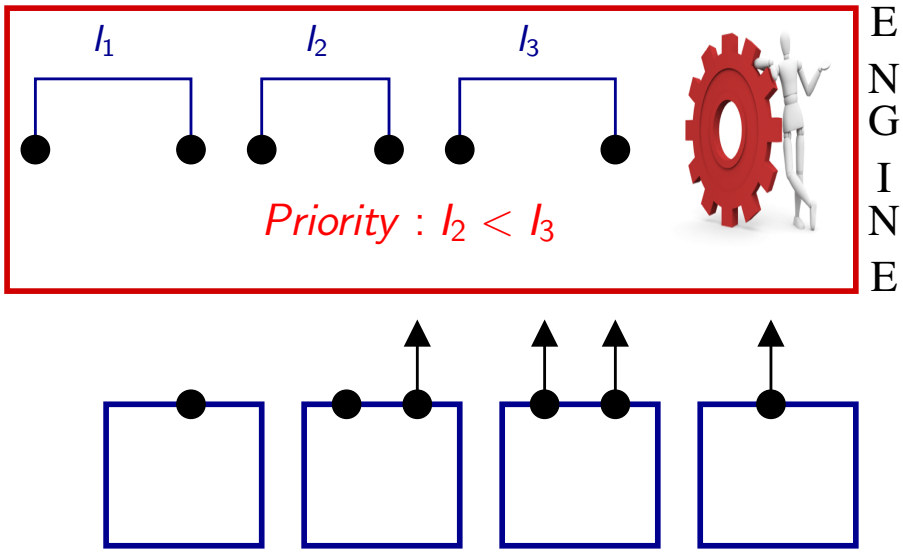
Engine Protocol

1. Atoms notify the engine of their active ports
2. The engine enumerates the allowed interactions
3. Filters out low priority ones
4. Picks one among those left
5. Notifies the atoms

Composite component

Composite component

available atomic components + connectors + priority rules



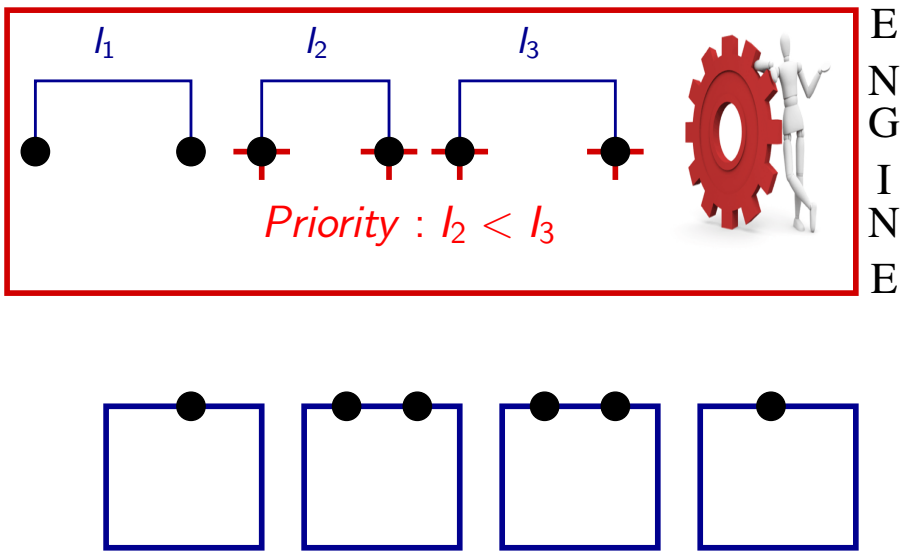
Engine Protocol

1. Atoms notify the engine of their active ports
2. The engine enumerates the allowed interactions
3. Filters out low priority ones
4. Picks one among those left
5. Notifies the atoms

Composite component

Composite component

available atomic components + connectors + priority rules



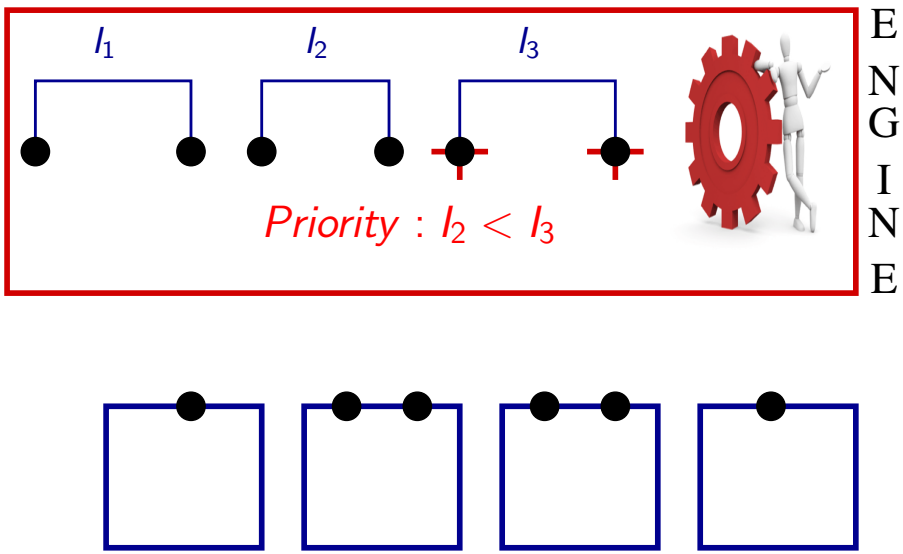
Engine Protocol

1. Atoms notify the engine of their active ports
2. The engine enumerates the allowed interactions
3. Filters out low priority ones
4. Picks one among those left
5. Notifies the atoms

Composite component

Composite component

available atomic components + connectors + priority rules



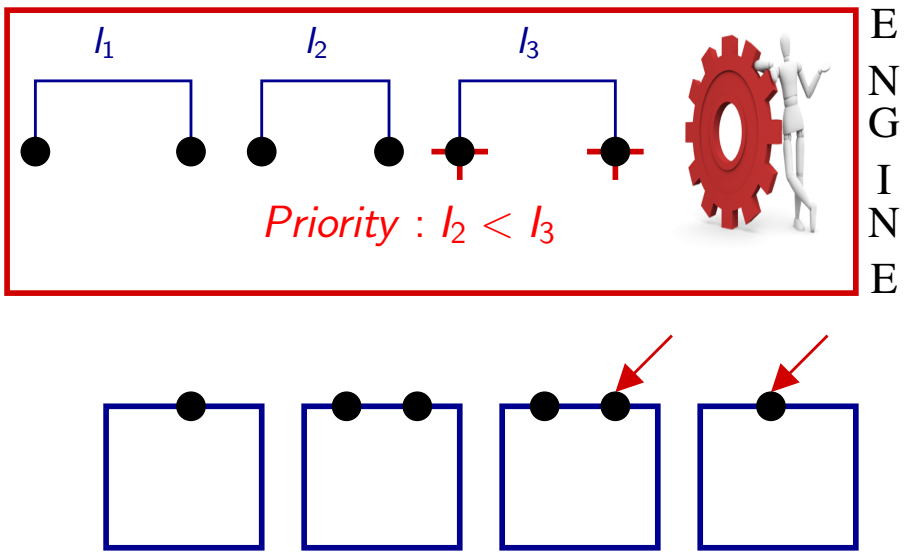
Engine Protocol

1. Atoms notify the engine of their active ports
2. The engine enumerates the allowed interactions
3. Filters out low priority ones
4. Picks one among those left
5. Notifies the atoms

Composite component

Composite component

available atomic components + connectors + priority rules



Engine Protocol

1. Atoms notify the engine of their active ports
2. The engine enumerates the allowed interactions
3. Filters out low priority ones
4. Picks one among those left
5. Notifies the atoms

Outline

1 The BIP Framework

2 An RV Framework for Component-Based Systems

3 Verifying the Runtime Behavior of BIP Systems

4 Experimental Results

5 Discussion

Monitoring a specification in the context of BIP systems

Specification = a desired behavior

- state-based
- linear-time

e.g., "in component B_1, $x > 0$ if in component B_2 $y < 0$"

Verification Monitor

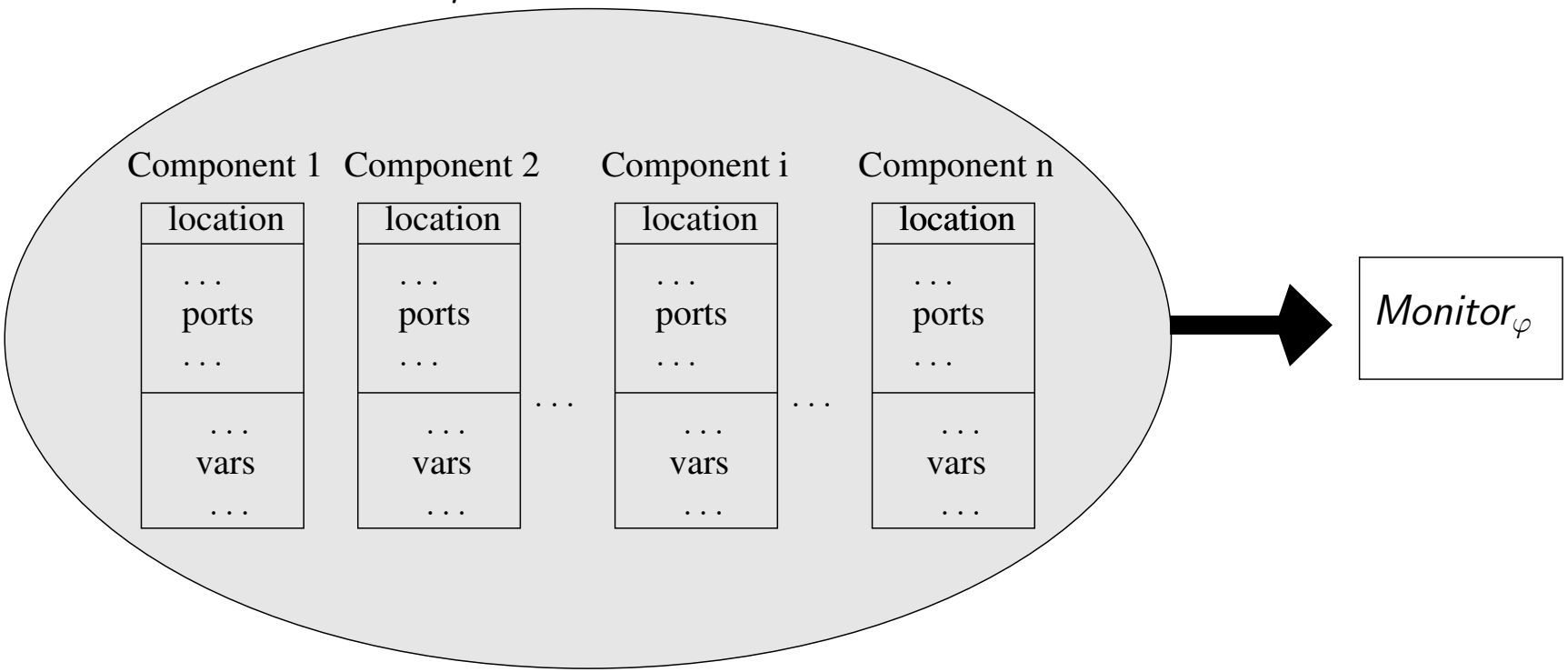
- *finite-state machine* with an *output function*
- *expressive:* any linear-time spec using 4-valued truth-domain
- *generic:* one can plug its own monitor synthesis algorithm
- execute in a *lock-step manner* with monitored system
- *abstracts* the original execution

Abstracting the original execution of the system

$$\varphi \leftrightarrow M_{\varphi}(C_2.port5, C_i.var6, \ldots, C_n.location)$$

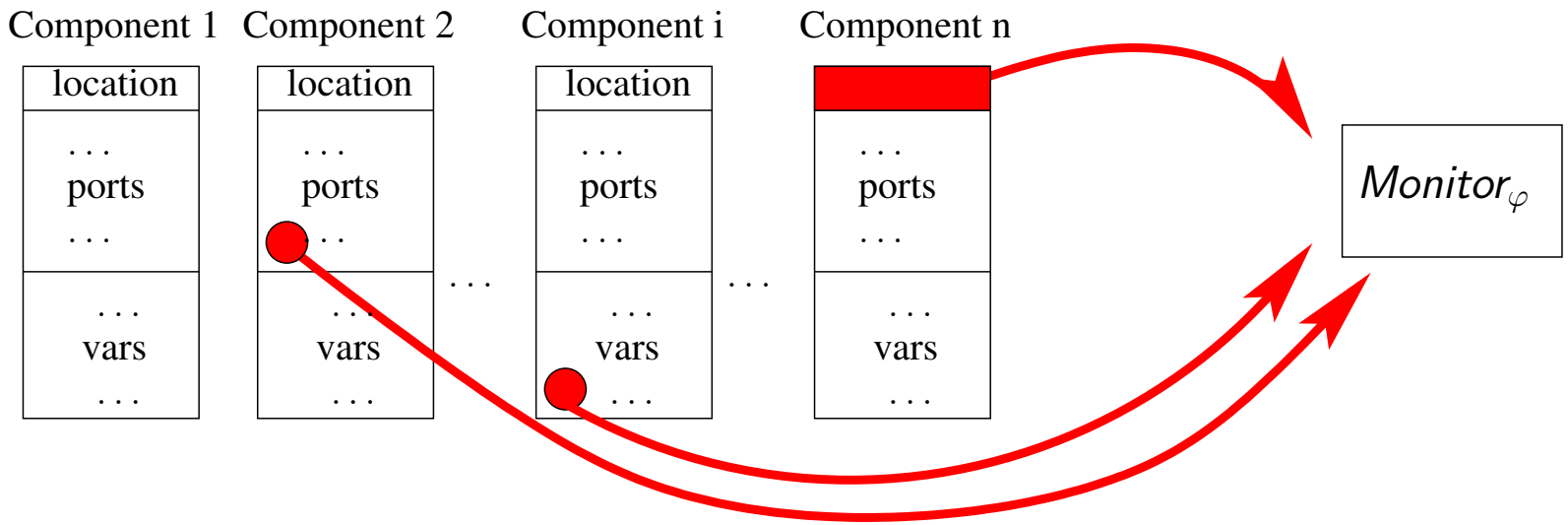
Abstracting the original execution of the system

$$\varphi \leftrightarrow M_\varphi(C_2.port5, C_i.var6, \ldots, C_n.location)$$



Abstracting the original execution of the system

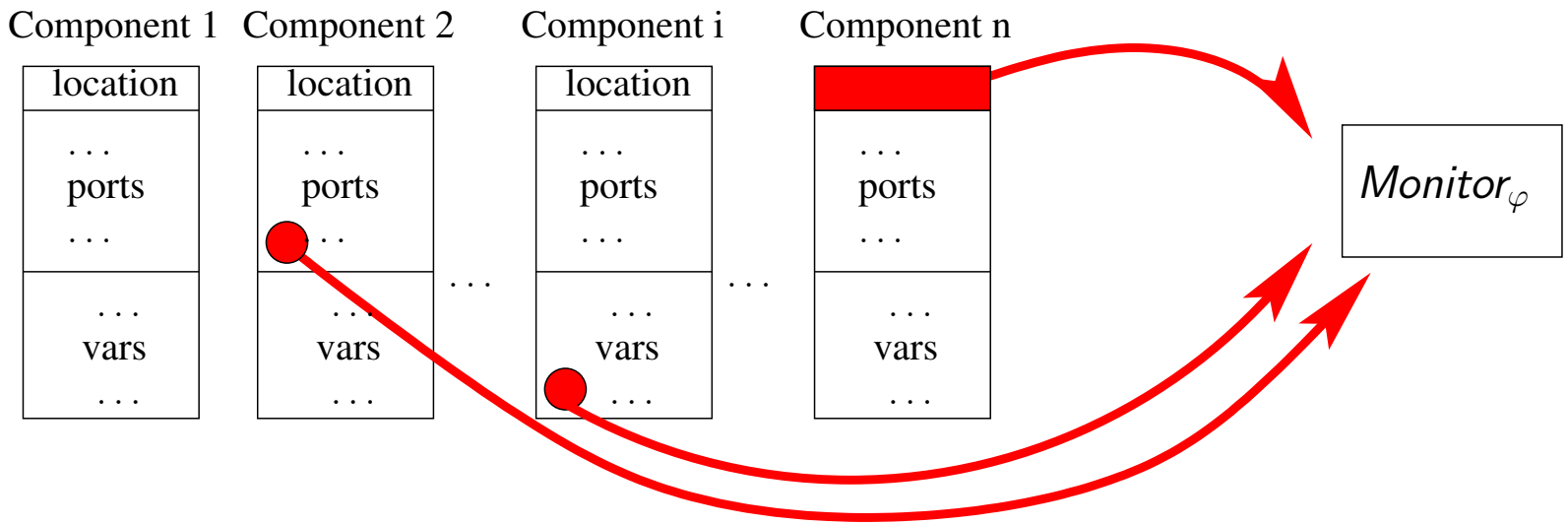
$$\varphi \leftrightarrow M_\varphi(C_2.port5, C_i.var6, \ldots, C_n.location)$$



The *least informative* function in
$\bigcup_{C_i \in Components} C_i.vars \cup C_i.ports \cup \{location_i\} \rightarrow Data$

Abstracting the original execution of the system

$$\varphi \leftrightarrow M_\varphi(C_2.port5, C_i.var6, \ldots, C_n.location)$$



Preserves property evaluation: sound and complete

Outline

1 The BIP Framework

2 An RV Framework for Component-Based Systems

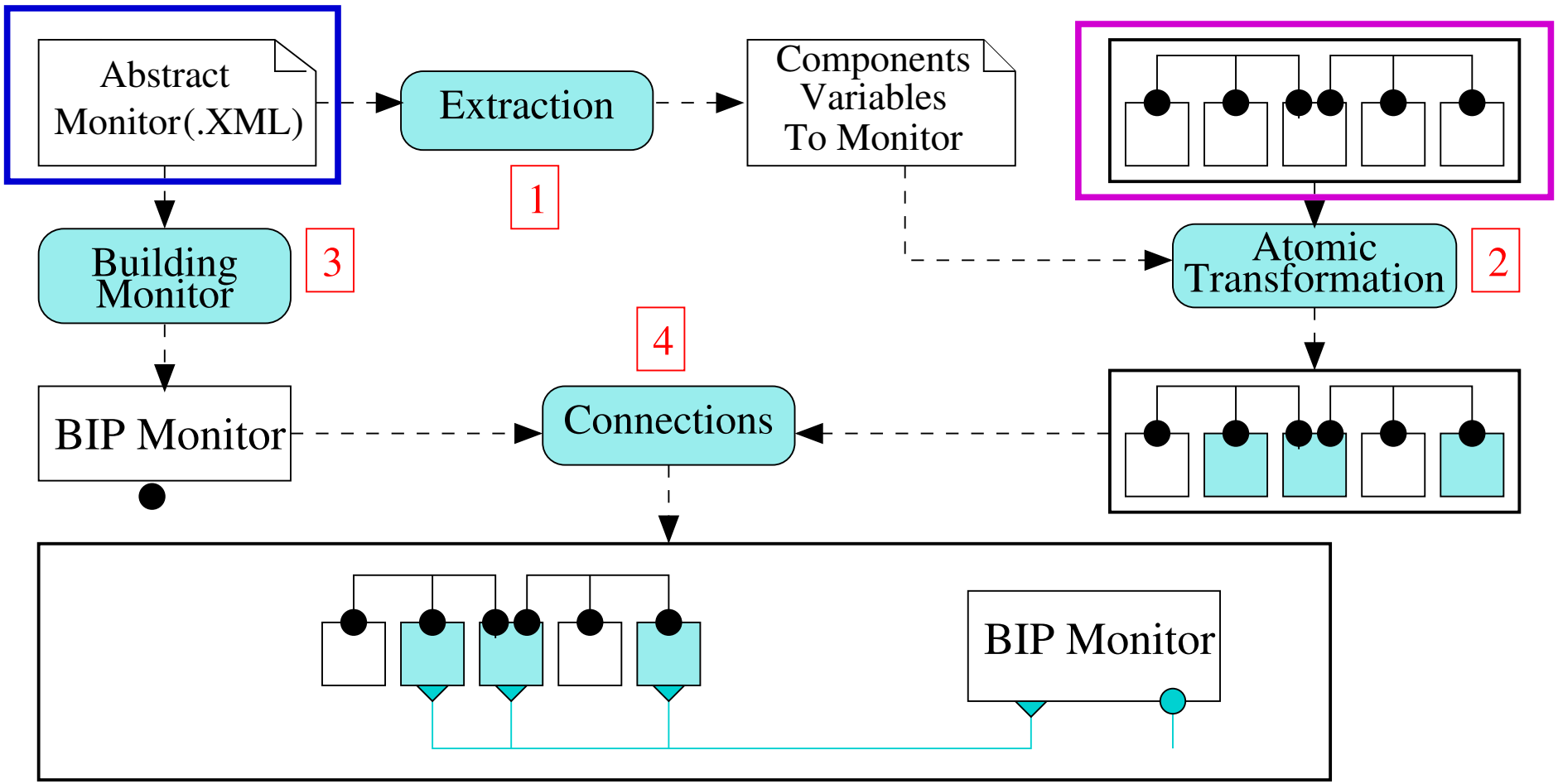
3 Verifying the Runtime Behavior of BIP Systems

4 Experimental Results

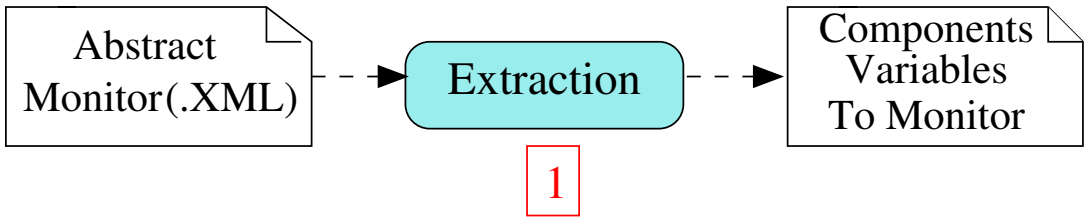
5 Discussion

Overview

Integrating an **abstract monitor** into a **BIP system**



Step 1: Extraction of needed information



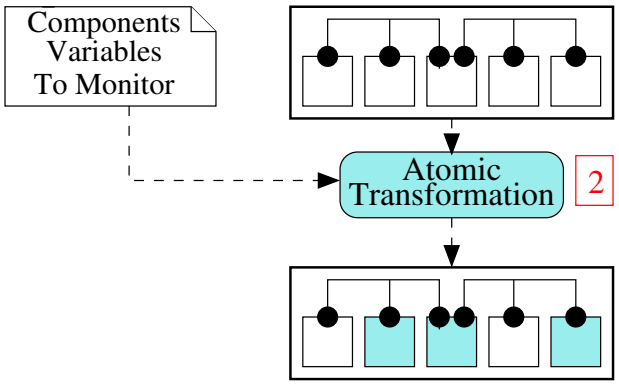
Retrieve from the monitor definition the needed information for each component to monitor the specification

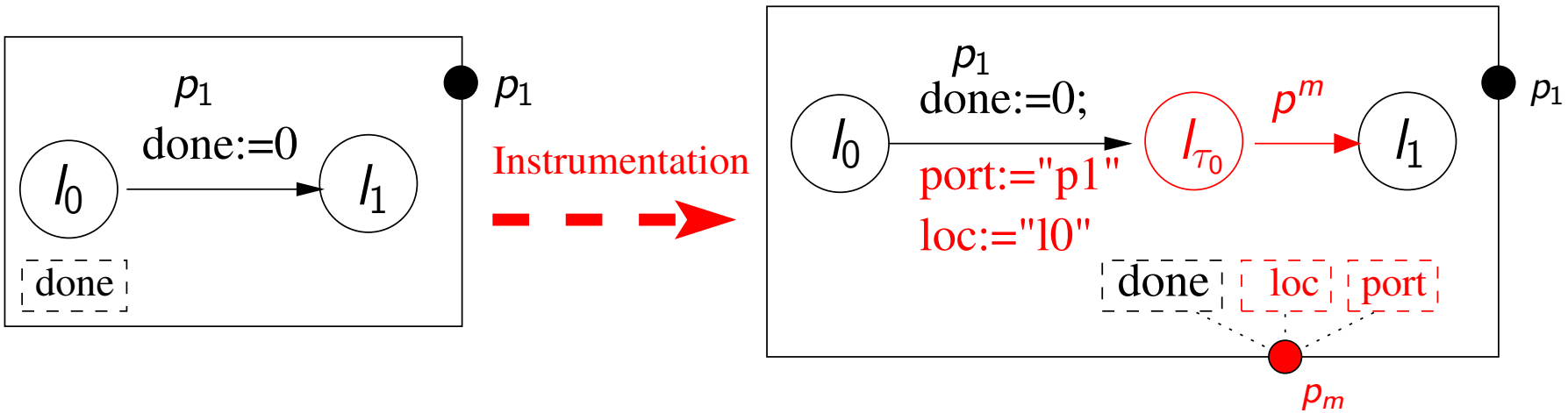
First step in the implementation of the abstraction function

Outcome: a mapping Components $\rightarrow \{vars, state, \ldots\}$

Step 2: Instrumentation of atomic components

Instrument each component so that it can interact with the monitor



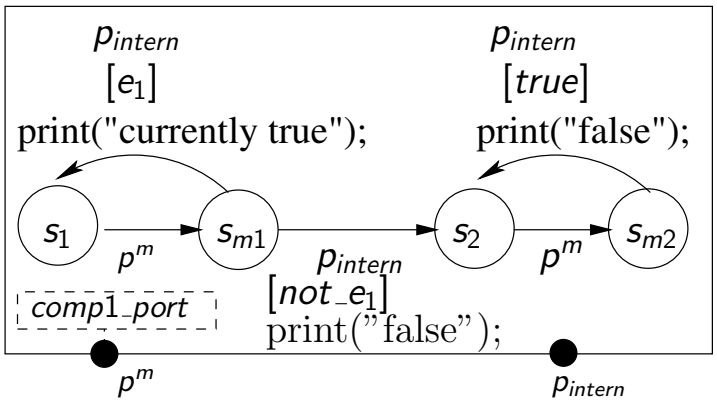


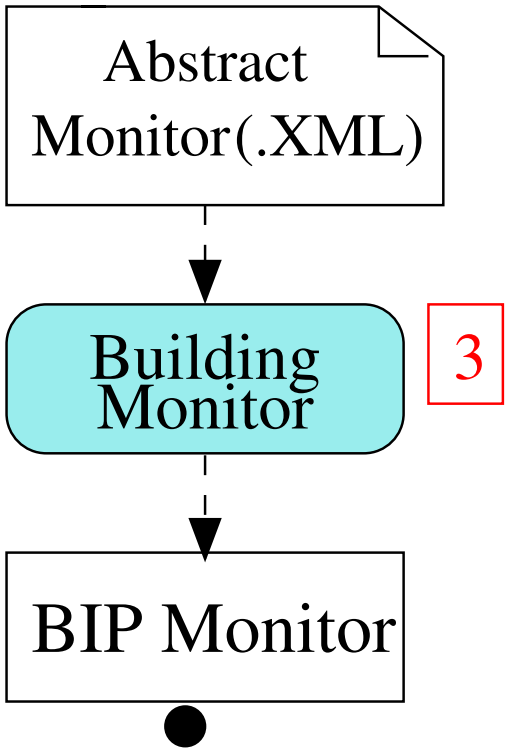
Step 3: Turning an abstract monitor into a BIP monitor

Abstract monitor (.XML) $\rightarrow$ Atomic component

- interacts with other instrumented components
- produces verdicts following the behavior of the original monitor

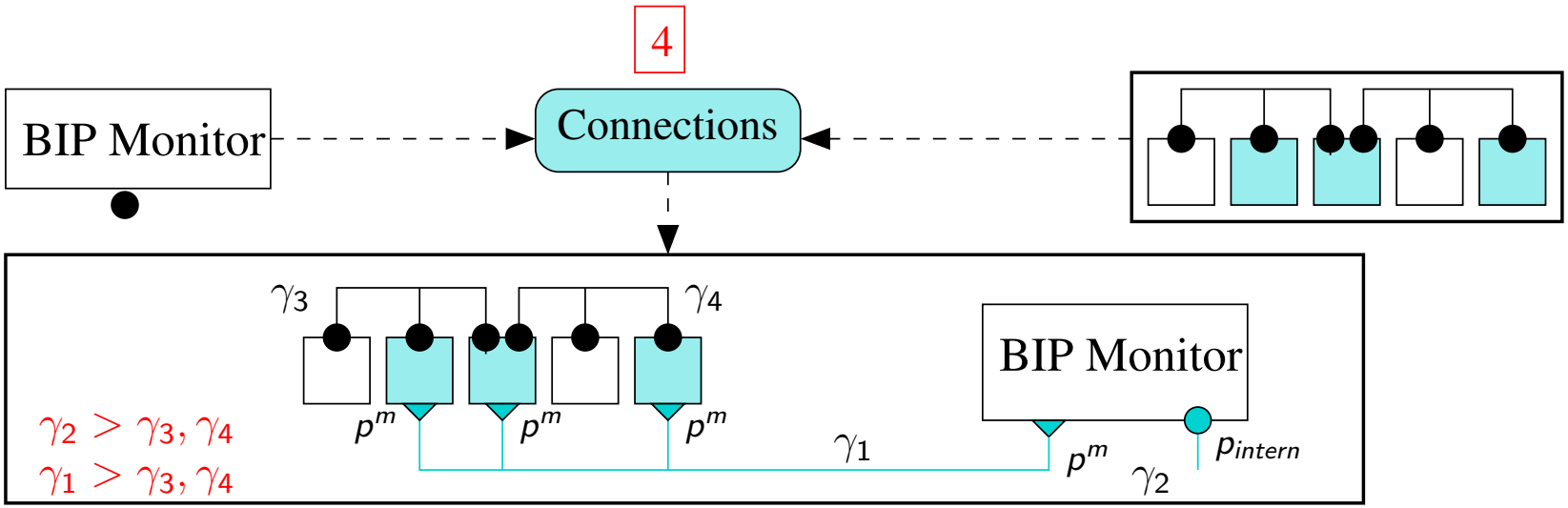
```
<VerificationMonitor>
    <State id="s1" initial="true">
       <Transition event="e1" nextState="s1" output="currently_true"/>
       <Transition event="not_e1" nextState="s2" output="false"/>
    </State>
    <State id="s2">
       <Transition event="true" nextState="s2" output="false"/>
    </State>
 </VerificationMonitor>
```





Step 4: Connections

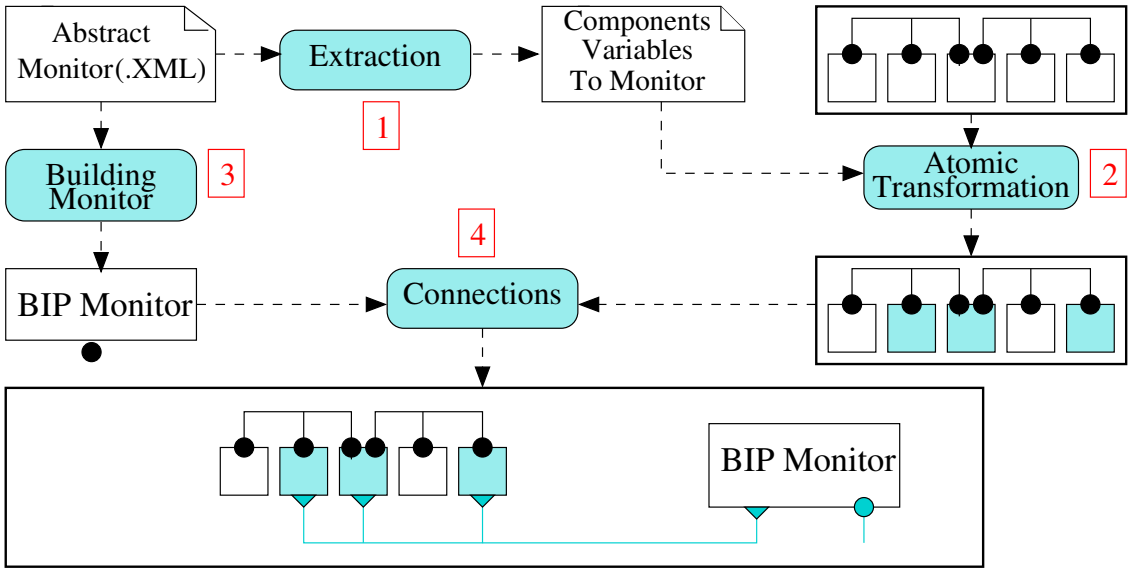
Connects instrumented components and the BIP monitor



Connectors containing p_m and p_{intern} have more priority

Summary/Discussion

4-stage approach to introduce an abstract monitor into a BIP system



Correctness

- Transformation do not modify the data nor the behavior
- No deadlock is introduced (monitor always ready)
- Fresh-data for the monitor (latest system state)

Outline

1. The BIP Framework
2. An RV Framework for Component-Based Systems
3. Verifying the Runtime Behavior of BIP Systems
4. Experimental Results
5. Discussion

Dala Robot

Dala Robot

- A large and realistic interactive system
- An infinite system
  - states
  - transitions

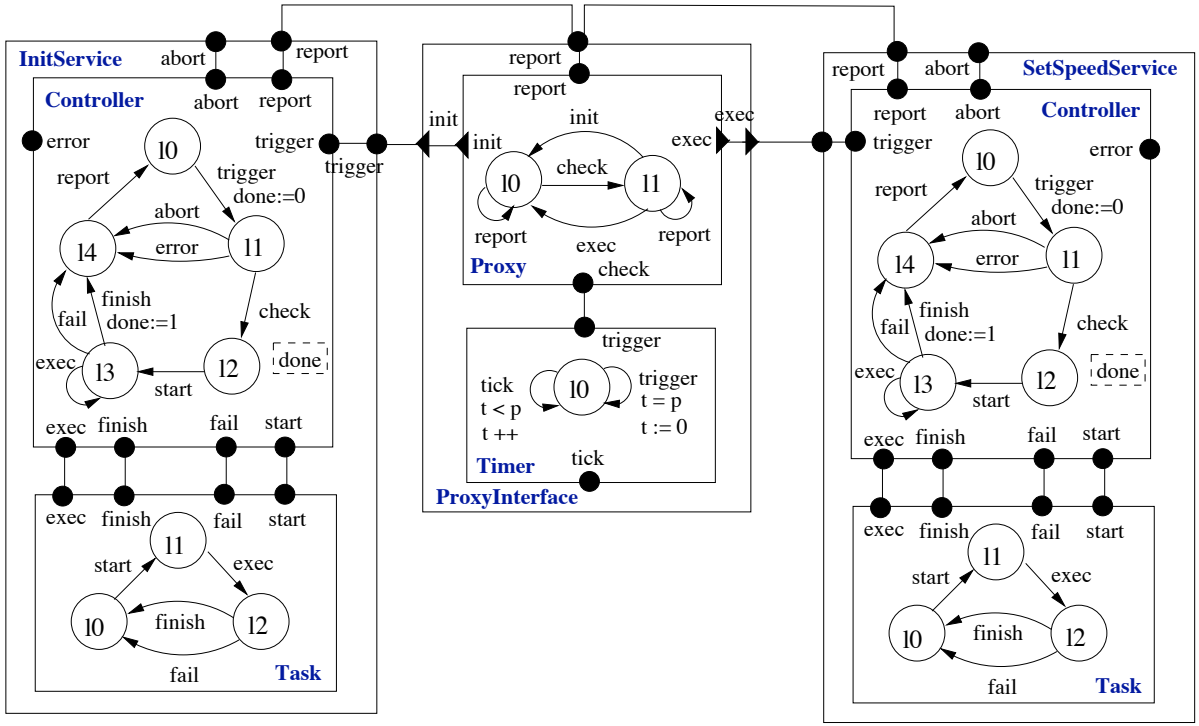
$\hookrightarrow$ cannot be model-checked!

- Functional level of Dala = a set of modules
- Each module:
  - a set of services (different tasks)
  - a set of posters (to exchange data between modules).

Simplified BIP Model of Dala Robot

Components

- ProxyInterface: communicates with the control layer using the mailbox
- InitService: responsible for the initialization of the module
- SetSpeedService: performs the main task of the module



Experiments: Monitoring some properties on Dala

- **Execution order**: InitService initializes the robot and should be successfully executed before SetSpeedService sets the speed parameter of the robot
- **Data freshness**: the data read by a service of a module (Reader) must be fresh enough compared to the moment it has been written (Writer)

	time-no-monitor	specification	optimized		not-optimized	
			time (s)	ovhd (%)	time (s)	ovhd (%)
Ordering violated	1.896	φ_1	2.045	7.8	9.163	383
		φ_2	1.953	3	9.192	384
Data freshness violated	1.638	φ_3	1.684	2.8	4.337	164
		φ_4	1.682	2.6	3.773	130

	time-no-monitor	specification	optimized		not-optimized	
			time (s)	ovhd (%)	time (s)	ovhd (%)
Ordering guaranteed	18.36	φ_1	19.84	8.0	89	384
		φ_2	18.89	2.8	88.96	384
Data freshness guaranteed	16.34	φ_3	16.78	2.6	43.83	168
		φ_4	16.90	3.4	37.82	131

Abstraction technique is *effective*

Monitoring only information in the specification using the abstraction technique *reduces* the overhead significantly!

Experiments: Monitoring some properties on Dala

- **Execution order**: InitService initializes the robot and should be successfully executed before SetSpeedService sets the speed parameter of the robot
- **Data freshness**: the data read by a service of a module (Reader) must be fresh enough compared to the moment it has been written (Writer)

	time-no-monitor	specification	optimized		not-optimized	
			time (s)	ovhd (%)	time (s)	ovhd (%)
Ordering violated	1.896	φ_1	2.045	7.8	9.163	383
		φ_2	1.953	3	9.192	384
Data freshness violated	1.638	φ_3	1.684	2.8	4.337	164
		φ_4	1.682	2.6	3.773	130

	time-no-monitor	specification	optimized		not-optimized	
			time (s)	ovhd (%)	time (s)	ovhd (%)
Ordering guaranteed	18.36	φ_1	19.84	8.0	89	384
		φ_2	18.89	2.8	88.96	384
Data freshness guaranteed	16.34	φ_3	16.78	2.6	43.83	168
		φ_4	16.90	3.4	37.82	131

Abstraction technique is *effective*

Monitoring only information in the specification using the abstraction technique *reduces* the overhead significantly!

Experiments: Monitoring some properties on Dala

- **Execution order**: InitService initializes the robot and should be successfully executed before SetSpeedService sets the speed parameter of the robot
- **Data freshness**: the data read by a service of a module (Reader) must be fresh enough compared to the moment it has been written (Writer)

	time-no-monitor	specification	optimized		not-optimized	
			time (s)	ovhd (%)	time (s)	ovhd (%)
Ordering violated	1.896	φ_1	2.045	7.8	9.163	383
		φ_2	1.953	3	9.192	384
Data freshness violated	1.638	φ_3	1.684	2.8	4.337	164
		φ_4	1.682	2.6	3.773	130

	time-no-monitor	specification	optimized		not-optimized	
			time (s)	ovhd (%)	time (s)	ovhd (%)
Ordering guaranteed	18.36	φ_1	19.84	8.0	89	384
		φ_2	18.89	2.8	88.96	384
Data freshness guaranteed	16.34	φ_3	16.78	2.6	43.83	168
		φ_4	16.90	3.4	37.82	131

Abstraction technique is *effective*

Monitoring only information in the specification using the abstraction technique *reduces* the overhead significantly!

Experiments: Monitoring some properties on Dala

- **Execution order**: InitService initializes the robot and should be successfully executed before SetSpeedService sets the speed parameter of the robot
- **Data freshness**: the data read by a service of a module (Reader) must be fresh enough compared to the moment it has been written (Writer)

	time-no-monitor	specification	optimized		not-optimized	
			time (s)	ovhd (%)	time (s)	**ovhd (%)**
Ordering violated	1.896	φ_1	2.045	7.8	9.163	**383**
		φ_2	1.953	3	9.192	**384**
Data freshness violated	1.638	φ_3	1.684	2.8	4.337	**164**
		φ_4	1.682	2.6	3.773	**130**

	time-no-monitor	specification	optimized		not-optimized	
			time (s)	ovhd (%)	time (s)	**ovhd (%)**
Ordering guaranteed	18.36	φ_1	19.84	8.0	89	**384**
		φ_2	18.89	2.8	88.96	**384**
Data freshness guaranteed	16.34	φ_3	16.78	2.6	43.83	**168**
		φ_4	16.90	3.4	37.82	**131**

Abstraction technique is *effective*

Monitoring only information in the specification using the abstraction technique *reduces* the overhead significantly!

Outline

1 The BIP Framework

2 An RV Framework for Component-Based Systems

3 Verifying the Runtime Behavior of BIP Systems

4 Experimental Results

5 Discussion

Discussion

Summary

- *Another validation (complementary) technique* for CBS (BIP systems)
- Based on a *general and expressive RV framework*
- *Abstraction* of the current system state
- Technique *scales* (realistic and large example)
- Monitored *real programs* (with BIP C-code generator)

Perspectives

- Using *combination with static analysis* for performance optimization
- *Dynamic instrumentation* technique
- More elaborated *abstraction techniques*
- *Runtime enforcement* (going beyond verdict detection)