Formal Verification of Data-Intensive Applications through Model Checking Modulo Theories

Marcello M. Bersani\textsuperscript{1}  \textbf{Mădălina Erașcu}\textsuperscript{2}  Francesco Marconi\textsuperscript{1}  
Silvio Ghilardi\textsuperscript{3}  Matteo Rossi\textsuperscript{1}

\textsuperscript{1}DEIB, Politecnico di Milano, Milano, Italy
\textsuperscript{2}West University of Timișoara and Institute eAustria, Timișoara, Romania
\textsuperscript{3}Università degli Studi di Milano, Milano, Italy

madalina.erascu@e-uvt.ro

July 13, 2017

This research was partially supported by Horizon 2020 project no. 644869 (DICE).
Big Data is everywhere!

Software market switches to Big Data: popular technologies such as Spark, Storm, Hadoop, and NoSQL stimulates Big Data adoption.

*Business issue:* 65% of Big Data projects still fail (*Capgemini Report 2015*)

**Solution:**
Integrating Quality Assurance (QA) practices in application development


**Prediction of quality properties of DIAs:**
- Performance, reliability, *safety properties*
- Helpful early in the DIA design
- Assess the potential impact of architectural changes (iteratively)
Big Data is everywhere!

Software market switches to Big Data: popular technologies such as Spark, Storm, Hadoop, and NoSQL stimulates Big Data adoption.

*Business issue:* 65% of Big Data projects still fail (*Capgemini Report 2015*)

**Solution:**

Integrating Quality Assurance (QA) practices in application development


Prediction of quality properties of DIAs:

- Performance, reliability, *safety properties*
- Helpful early in the DIA design
- Assess the potential impact of architectural changes (iteratively)
Big Data is everywhere!

Software market switches to Big Data: popular technologies such as Spark, Storm, Hadoop, and NoSQL stimulates Big Data adoption.

*Business issue:* 65% of Big Data projects still fail *(Capgemini Report 2015)*

**Solution:**
Integrating Quality Assurance (QA) practices in application development

DICE project (http://www.dice-h2020.eu/) aims to define methods and tools for the data-aware *quality*-driven development of Date Intensive Applications (DIAs).

Prediction of quality properties of DIAs:
- Performance, reliability, *safety properties*
- Helpful early in the DIA design
- Assess the potential impact of architectural changes (iteratively)
Big Data is everywhere!

Software market switches to Big Data: popular technologies such as Spark, Storm, Hadoop, and NoSQL stimulates Big Data adoption.

Business issue: 65% of Big Data projects still fail (Capgemini Report 2015)

Solution:
Integrating Quality Assurance (QA) practices in application development

DICE project (http://www.dice-h2020.eu/) aims to define methods and tools for the data-aware quality-driven development of Date Intensive Applications (DIAs).

Prediction of quality properties of DIAs:

- Performance, reliability, safety properties
- Helpful early in the DIA design
- Assess the potential impact of architectural changes (iteratively)
Big Data is everywhere!

Software market switches to Big Data: popular technologies such as Spark, Storm, Hadoop, and NoSQL stimulates Big Data adoption.

Business issue: 65% of Big Data projects still fail (Capgemini Report 2015)

Solution:
Integrating Quality Assurance (QA) practices in application development

DICE project (http://www.dice-h2020.eu/) aims to define methods and tools for the data-aware quality-driven development of Date Intensive Applications (DIAs).

Prediction of quality properties of DIAs:

- Performance, reliability, safety properties
- Helpful early in the DIA design
- Assess the potential impact of architectural changes (iteratively)
Big Data is everywhere!

Software market switches to Big Data: popular technologies such as Spark, Storm, Hadoop, and NoSQL stimulates Big Data adoption.

*Business issue:* 65% of Big Data projects still fail (Capgemini Report 2015)

**Solution:**

Integrating Quality Assurance (QA) practices in application development

DICE project (http://www.dice-h2020.eu/) aims to define methods and tools for the data-aware quality-driven development of Date Intensive Applications (DIAs).

Prediction of quality properties of DIAs:

- Performance, reliability, *safety properties*
- Helpful early in the DIA design
- Assess the potential impact of architectural changes (iteratively)
Big Data is everywhere!

Software market switches to Big Data: popular technologies such as Spark, Storm, Hadoop, and NoSQL stimulates Big Data adoption.

*Business issue:* 65% of Big Data projects still fail *(Capgemini Report 2015)*

**Solution:**
Integrating Quality Assurance (QA) practices in application development

DICE project (http://www.dice-h2020.eu/) aims to define methods and tools for the data-aware *quality*-driven development of Date Intensive Applications (DIAs).

**Prediction of quality properties of DIAs:**
- Performance, reliability, *safety properties*
- Helpful early in the DIA design
  - Assess the potential impact of architectural changes (iteratively)
Big Data is everywhere!

Software market switches to Big Data: popular technologies such as Spark, Storm, Hadoop, and NoSQL stimulates Big Data adoption.

*Business issue:* 65% of Big Data projects still fail (Capgemini Report 2015)

**Solution:**
Integrating Quality Assurance (QA) practices in application development

DICE project (http://www.dice-h2020.eu/) aims to define methods and tools for the data-aware *quality*-driven development of Date Intensive Applications (DIAs).

**Prediction of quality properties of DIAs:**
- Performance, reliability, *safety properties*
- Helpful early in the DIA design
- Assess the potential impact of architectural changes (iteratively)
Focus: Apache Storm technology

Apache Storm (storm.apache.org) technology – used in applications that need efficient processing of unbounded streams of data, such as event log monitoring, real-time data analytics and data normalization.

Applications that use Storm: Yahoo, Twitter, Spotify, The Weather Channel, etc.

Key concepts:
- **streams** – infinite sequences of **tuples** that are processed by the application
- **topologies** – directed graphs
  - nodes represent operations performed over the application data
  - edges indicate how such operations are combined

Types of nodes:
- **input nodes** bring information into the application from the environment: spouts
- **computational nodes** implement the logic of the application by processing information and producing a result: bolts
Focus: Apache Storm technology

Apache Storm (storm.apache.org) technology – used in applications that need efficient processing of unbounded streams of data, such as event log monitoring, real-time data analytics and data normalization.

Applications that use Storm: Yahoo, Twitter, Spotify, The Weather Channel, etc.

Key concepts:

▷ streams – infinite sequences of tuples that are processed by the application
▷ topologies – directed graphs
  ▷ nodes represent operations performed over the application data
  ▷ edges indicate how such operations are combined

Types of nodes:

▷ input nodes bring information into the application from the environment: spouts
▷ computational nodes implement the logic of the application by processing information and producing a result: bolts
Focus: Apache Storm technology

Apache Storm (storm.apache.org) technology – used in applications that need efficient processing of unbounded streams of data, such as event log monitoring, real-time data analytics and data normalization.

Applications that use Storm: Yahoo, Twitter, Spotify, The Weather Channel, etc.

Key concepts:

▶ streams – infinite sequences of tuples that are processed by the application
▶ topologies – directed graphs
  ▶ nodes represent operations performed over the application data
  ▶ edges indicate how such operations are combined

Types of nodes:

▶ input nodes bring information into the application from the environment: spouts
▶ computational nodes implement the logic of the application by processing information and producing a result: bolts
Focus: Apache Storm technology

Apache Storm (storm.apache.org) technology – used in applications that need efficient processing of unbounded streams of data, such as event log monitoring, real-time data analytics and data normalization.

Applications that use Storm: Yahoo, Twitter, Spotify, The Weather Channel, etc.

Key concepts:

- **streams** – infinite sequences of **tuples** that are processed by the application
- **topologies** – directed graphs
  - **nodes** represent operations performed over the application data
  - **edges** indicate how such operations are combined

Types of nodes:

- **input nodes** bring information into the application from the environment: **spouts**
- **computational nodes** implement the logic of the application by processing information and producing a result: **bolts**
Focus: Apache Storm technology

Apache Storm (storm.apache.org) technology – used in applications that need efficient processing of unbounded streams of data, such as event log monitoring, real-time data analytics and data normalization.

Applications that use Storm: Yahoo, Twitter, Spotify, The Weather Channel, etc.

Key concepts:
- **streams** – infinite sequences of **tuples** that are processed by the application
- **topologies** – directed graphs
  - **nodes** represent operations performed over the application data
  - **edges** indicate how such operations are combined

Types of nodes:
- **input nodes** bring information into the application from the environment: spouts
- **computational nodes** implement the logic of the application by processing information and producing a result: bolts
Focus: Apache Storm technology

Apache Storm (storm.apache.org) technology – used in applications that need efficient processing of unbounded streams of data, such as event log monitoring, real-time data analytics and data normalization.

Applications that use Storm: Yahoo, Twitter, Spotify, The Weather Channel, etc.

Key concepts:

- **streams** – infinite sequences of **tuples** that are processed by the application
- **topologies** – directed graphs
  - **nodes** represent operations performed over the application data
  - **edges** indicate how such operations are combined

Types of nodes:

- **input nodes** bring information into the application from the environment: spouts
- **computational nodes** implement the logic of the application by processing information and producing a result: bolts
Focus: Apache Storm technology

Apache Storm (storm.apache.org) technology – used in applications that need efficient processing of unbounded streams of data, such as event log monitoring, real-time data analytics and data normalization.

Applications that use Storm: Yahoo, Twitter, Spotify, The Weather Channel, etc.

Key concepts:
- **streams** – infinite sequences of *tuples* that are processed by the application
- **topologies** – directed graphs
  - **nodes** represent operations performed over the application data
  - **edges** indicate how such operations are combined

Types of nodes:
- *input nodes* bring information into the application from the environment: spouts
- *computational nodes* implement the logic of the application by processing information and producing a result: bolts
Focus: Apache Storm technology

Apache Storm (storm.apache.org) technology – used in applications that need efficient processing of unbounded streams of data, such as event log monitoring, real-time data analytics and data normalization.

Applications that use Storm: Yahoo, Twitter, Spotify, The Weather Channel, etc.

Key concepts:

- **streams** – infinite sequences of **tuples** that are processed by the application
- **topologies** – directed graphs
  - **nodes** represent operations performed over the application data
  - **edges** indicate how such operations are combined

Types of nodes:

- **input nodes** bring information into the application from the environment: spouts
- **computational nodes** implement the logic of the application by processing information and producing a result: bolts
Focus: Apache Storm technology

Apache Storm (storm.apache.org) technology – used in applications that need efficient processing of unbounded streams of data, such as event log monitoring, real-time data analytics and data normalization.

Applications that use Storm: Yahoo, Twitter, Spotify, The Weather Channel, etc.

Key concepts:

▶ **streams** – infinite sequences of **tuples** that are processed by the application
▶ **topologies** – directed graphs
  ▶ **nodes** represent operations performed over the application data
  ▶ **edges** indicate how such operations are combined

Types of nodes:

▶ **input nodes** bring information into the application from the environment: **spouts**
▶ **computational nodes** implement the logic of the application by processing information and producing a result: **bolts**
Focus: Apache Storm technology

Apache Storm ([storm.apache.org](http://storm.apache.org)) technology – used in applications that need efficient processing of unbounded streams of data, such as event log monitoring, real-time data analytics and data normalization.

Applications that use Storm: Yahoo, Twitter, Spotify, The Weather Channel, etc.

Key concepts:

- **streams** – infinite sequences of **tuples** that are processed by the application
- **topologies** – directed graphs
  - **nodes** represent operations performed over the application data
  - **edges** indicate how such operations are combined

Types of nodes:

- **input nodes** bring information into the application from the environment: **spouts**
- **computational nodes** implement the logic of the application by processing information and producing a result: **bolts**
Focus: Apache Storm technology

Apache Storm (storm.apache.org) technology – used in applications that need efficient processing of unbounded streams of data, such as event log monitoring, real-time data analytics and data normalization.

Applications that use Storm: Yahoo, Twitter, Spotify, The Weather Channel, etc.

Key concepts:

- **streams** – infinite sequences of **tuples** that are processed by the application
- **topologies** – directed graphs
  - **nodes** represent operations performed over the application data
  - **edges** indicate how such operations are combined

Types of nodes:

- *input nodes* bring information into the application from the environment: **spouts**
- *computational nodes* implement the logic of the application by processing information and producing a result: **bolts**
Example of Storm topology

Finite state automata describing the states of a spout (left) and bolt (right)

Features of the topology:
- parametric in the number of nodes and processes
- the number of nodes is known at design-time, hence fixed

**Infinite-state model checking!**

Suitable abstraction: array-based systems. (Ghilardi et al.)

Safety verification of Storm topologies: given queue(s) bound(s) defined by the designer, “all bolt queues have a limited occupation level”.

Example of Storm topology

Finite state automata describing the states of a spout (left) and bolt (right)

Features of the topology:
- parametric in the number of nodes and processes
- the number of nodes is known at design-time, hence fixed

Infinite-state model checking!

Suitable abstraction: array-based systems. (Ghilardi et al.)

Safety verification of Storm topologies: given queue(s) bound(s) defined by the designer, “all bolt queues have a limited occupation level”.
Example of Storm topology

Finite state automata describing the states of a spout (left) and bolt (right)

Features of the topology:

- **parametric** in the number of nodes and processes
- the number of nodes is known at design-time, hence fixed

**Infinite-state model checking!**

Suitable abstraction: **array-based systems.** (Ghilardi et al.)

Safety verification of Storm topologies: given queue(s) bound(s) defined by the designer, “all bolt queues have a limited occupation level”.
Example of Storm topology

Finite state automata describing the states of a spout (left) and bolt (right)

Features of the topology:
- parametric in the number of nodes and processes
- the number of nodes is known at design-time, hence fixed

Infinite-state model checking!

Suitable abstraction: array-based systems. (Ghilardi et al.)

Safety verification of Storm topologies: given queue(s) bound(s) defined by the designer, “all bolt queues have a limited occupation level”.
Example of Storm topology

Finite state automata describing the states of a spout (left) and bolt (right)

Features of the topology:
- parametric in the number of nodes and processes
- the number of nodes is known at design-time, hence fixed

Infinite-state model checking!

Suitable abstraction: array-based systems. (Ghilardi et al.)

Safety verification of Storm topologies: given queue(s) bound(s) defined by the designer, “all bolt queues have a limited occupation level”. 
Example of Storm topology

Finite state automata describing the states of a spout (left) and bolt (right)

Features of the topology:
- parametric in the number of nodes and processes
- the number of nodes is known at design-time, hence fixed

Infinite-state model checking!

Suitable abstraction: array-based systems. (Ghilardi et al.)

Safety verification of Storm topologies: given queue(s) bound(s) defined by the designer, “all bolt queues have a limited occupation level”.
Example of Storm topology

Finite state automata describing the states of a spout (left) and bolt (right)

Features of the topology:
- parametric in the number of nodes and processes
- the number of nodes is known at design-time, hence fixed

Infinite-state model checking!

Suitable abstraction: array-based systems. (Ghilardi et al.)

Safety verification of Storm topologies: given queue(s) bound(s) defined by the designer, “all bolt queues have a limited occupation level”.

Array-based Systems

Formalizing an array-based system means specifying:

- the set of initial states
- the ordering of the actions (by means of a transition relation)
- the set of unsafe states (the negation of the formula we want to check)

Examples (Cubicle syntax):

- `init (i x) { T = 0.0 && B[i,x]=I && ... } \leadsto \forall i, x \ldots`

- `transition spout.emit(i j) \leadsto \exists i, j \ldots`
  requires `{Tsmin<Stime[i] && SubscribedBS[j,i]=True && ...}
  `{L[l]:=case
    | l=j : L[l]+1.0
    | _ : L[l];

- `unsafe(i) { L[i]>1.5 } \leadsto \exists i \ldots`

Symbolic representation of array-based systems: quantified first-order logic formulae.
Verification of array-based systems: decision procedure based on backward reachability.

Termination of the decision procedure:

- the current set of reachable states has a non-empty intersection with the set of initial states (safety check) \( \Rightarrow \) system is unsafe
- the current set has reached a fix-point (fix-point check) \( \Rightarrow \) system is safe
Array-based Systems

Formalizing an array-based system means specifying:

- the set of initial states
- the ordering of the actions (by means of a transition relation)
- the set of unsafe states (the negation of the formula we want to check)

Examples (Cubicle syntax):

- `init (i x) { T = 0.0 && B[i,x]=I && ... } ⇝ ∀ i, x ...`
- `transition spout.emit(i j) ⇝ ∃ i, j ...`
  
  requires `{Tsmin<Stime[i] && SubscribedBS[j,i]=True && ...}`
  
  `{ L[1]:=case
        | 1=j : L[1]+1.0
        | _ : L[1];
  }

- `unsafe(i) { L[i]>1.5 } ⇝ ∃ i ...

Symbolic representation of array-based systems: quantified first-order logic formulae. Verification of array-based systems: decision procedure based on backward reachability. Termination of the decision procedure:

- the current set of reachable states has a non-empty intersection with the set of initial states (safety check) ⇒ system is unsafe
- the current set has reached a fix-point (fix-point check) ⇒ system is safe
Array-based Systems

Formalizing an array-based system means specifying:

- the set of initial states
- the ordering of the actions (by means of a transition relation)
- the set of unsafe states (the negation of the formula we want to check)

Examples (Cubicle syntax):

- `init (i x) { T = 0.0 && B[i,x]=I && ... } \Rightarrow \forall i, x ...`

- `transition spout.emit(i j) \Rightarrow \exists i, j ...`
  
  `requires {Tsmin<Stime[i] && SubscribedBS[j,i]=True && ...} { L[1]:=case
  | 1=j : L[1]+1.0
  | _ : L[1];
  }

- `unsafe(i) { L[i]>1.5 } \Rightarrow \exists i ...

Symbolic representation of array-based systems: quantified first-order logic formulae. Verification of array-based systems: decision procedure based on backward reachability. Termination of the decision procedure:

- the current set of reachable states has a non-empty intersection with the set of initial states (safety check) \( \Rightarrow \) system is unsafe
- the current set has reached a fix-point (fix-point check) \( \Rightarrow \) system is safe
Array-based Systems

Formalizing an array-based system means specifying:

- the set of initial states
- the ordering of the actions (by means of a transition relation)
- the set of unsafe states (the negation of the formula we want to check)

Examples (Cubicle syntax):

- `init (i x) { T = 0.0 && B[i,x]=I && ... } \Rightarrow \forall i,x ...
- transition spout.emit(i j) \Rightarrow \exists i,j ...
- requires {Tsmin<Stime[i] && SubscribedBS[j,i]=True && ...} 
  { 
    L[1]:=case 
    | 1=j : L[1]+1.0 
    | _ : L[1]; 
  }
- unsafe(i) { L[i]>1.5 } \Rightarrow \exists ...

Symbolic representation of array-based systems: quantified first-order logic formulae. Verification of array-based systems: decision procedure based on backward reachability. Termination of the decision procedure:

- the current set of reachable states has a non-empty intersection with the set of initial states (safety check) \Rightarrow system is unsafe
- the current set has reached a fix-point (fix-point check) \Rightarrow system is safe
Array-based Systems

Formalizing an array-based system means specifying:

- the set of initial states
- the ordering of the actions (by means of a transition relation)
- the set of unsafe states (the negation of the formula we want to check)

Examples (Cubicle syntax):

- `init (i x) { T = 0.0 && B[i,x]=I && ... } \Rightarrow \forall i, x ...
- transition spout.emit(i j) \Rightarrow \exists i, j ...
  requires {Tsmin<Stime[i] && SubscribedBS[j,i]=True && ...
  \{ 
    L[l]:=case 
    | l=j : L[l]+1.0 
    | _ : L[l];
  \}
- unsafe(i) { L[i]>1.5 } \Rightarrow \exists i ...

Symbolic representation of array-based systems: quantified first-order logic formulae. Verification of array-based systems: decision procedure based on backward reachability. Termination of the decision procedure:

- the current set of reachable states has a non-empty intersection with the set of initial states (safety check) \(\Rightarrow\) system is unsafe
- the current set has reached a fix-point (fix-point check) \(\Rightarrow\) system is safe
Array-based Systems

Formalizing an array-based system means specifying:

▶ the set of initial states
▶ the ordering of the actions (by means of a transition relation)
▶ the set of unsafe states (the negation of the formula we want to check)

Examples (Cubicle syntax):

▶ init (i x) { T = 0.0 && B[i,x]=I && ... } \implies \forall_{i,x} ...

▶ transition spout.emit(i j) \implies \exists_{i,j} ...

requires {Tsmin<Stime[i] && SubscribedBS[j,i]=True && ...}

{ 
    L[1]:=case 
    | l=j : L[1]+1.0 
    | _ : L[1];
}

▶ unsafe(i) { L[i]>1.5 } \implies \exists_{i} ...

Symbolic representation of array-based systems: quantified first-order logic formulae. Verification of array-based systems: decision procedure based on backward reachability. Termination of the decision procedure:

▶ the current set of reachable states has a non-empty intersection with the set of initial states (safety check) \Rightarrow system is unsafe
▶ the current set has reached a fix-point (fix-point check) \Rightarrow system is safe
Array-based Systems

Formalizing an array-based system means specifying:

- the set of initial states
- the ordering of the actions (by means of a transition relation)
- the set of unsafe states (the negation of the formula we want to check)

Examples (Cubicle syntax):

- init \( (i, x) \) \{ \( T = 0.0 \) \&\& \( B[i, x] = I \) \&\& ... \} \implies \forall i, x ...

- transition spout_emit\( (i, j) \) \implies \exists i, j ...

  requires \{Tsmin<Stime[i] \&\& \text{SubscribedBS}[j,i]=True \&\& ...\}

  \{ \begin{align*}
  L[1] &:= \text{case} \\
  | l=j : L[1]+1.0 \\
  | _ : L[1];
  \end{align*} \}

- unsafe\( (i) \) \{ \( L[i] > 1.5 \) \} \implies \exists i ...


Termination of the decision procedure:

- the current set of reachable states has a non-empty intersection with the set of initial states (safety check) \( \Rightarrow \) system is unsafe
- the current set has reached a fix-point (fix-point check) \( \Rightarrow \) system is safe
Array-based Systems

Formalizing an array-based system means specifying:

▶ the set of initial states
▶ the ordering of the actions (by means of a transition relation)
▶ the set of unsafe states (the negation of the formula we want to check)

Examples (Cubicle syntax):

▶ init (i x) { T = 0.0 && B[i,x]=I && ... } ⇝ \( \forall i, x \ldots \)

▶ transition spout.emit(i j) ⇝ \( \exists i, j \ldots \)

requires \{ Tsmin<Stime[i] && SubscribedBS[j,i]=True && ... \}

\{ 
  L[1]:=case 
  | l=j : L[1]+1.0 
  | _ : L[1]; 
\}

▶ unsafe(i) { L[i]>1.5 } ⇝ \( \exists i \ldots \)

Symbolic representation of array-based systems: quantified first-order logic formulae.
Verification of array-based systems: decision procedure based on backward reachability.

Termination of the decision procedure:

▶ the current set of reachable states has a non-empty intersection with the set of initial states (safety check) ⇒ system is unsafe
▶ the current set has reached a fix-point (fix-point check) ⇒ system is safe
Array-based Systems

Formalizing an array-based system means specifying:

- the set of initial states
- the ordering of the actions (by means of a transition relation)
- the set of unsafe states (the negation of the formula we want to check)

Examples (Cubicle syntax):

- \( \text{init} \ (i \ x) \{ \ T = 0.0 \ \&\& \ B[i,x]=I \ \&\& \ldots \} \ \Rightarrow \ \forall_{i,x} \ldots \)

- \( \text{transition spout_emit}(i \ j) \ \Rightarrow \ \exists_{i,j} \ldots \)

  \[ \text{requires} \ \{\ Tsmin<Stime[i] \ \&\& \ SubscribedBS[j,i]=True \ \&\& \ldots \} \]

  \[ \{\ L[l]:=\text{case} \]
  \[ \quad \mid \ l=j : \ L[l]+1.0 \]
  \[ \quad \mid \ _ : \ L[l]; \]

- \( \text{unsafe}(i) \ \{\ L[i]>1.5 \} \ \Rightarrow \ \exists_{i} \ldots \)


Termination of the decision procedure:

- the current set of reachable states has a non-empty intersection with the set of initial states (safety check) \( \Rightarrow \) system is unsafe
- the current set has reached a fix-point (fix-point check) \( \Rightarrow \) system is safe
Array-based Systems

Formalizing an array-based system means specifying:

- the set of initial states
- the ordering of the actions (by means of a transition relation)
- the set of unsafe states (the negation of the formula we want to check)

Examples (Cubicle syntax):

- **init** (i x) \{ T = 0.0 \&\& B[i,x]=I \&\& ... \} \implies \forall_{i,x} ...
- **transition spout.emit**(i j) \implies \exists_{i,j} ...
  requires \{ Tsmin<Stime[i] \&\& SubscribedBS[j,i]=True \&\& ... \}
  \{ L[1]:=\text{case}
  \mid l=j : L[1]+1.0
  \mid _ : L[1];
  \}
- **unsafe**(i) \{ L[i]>1.5 \} \implies \exists_{i} ...

Symbolic representation of array-based systems: *quantified first-order logic formulae*. Verification of array-based systems: *decision procedure* based on backward reachability. Termination of the decision procedure:

- the current set of reachable states has a non-empty intersection with the set of initial states (safety check) \implies system is unsafe
- the current set has reached a fix-point (fix-point check) \implies system is safe
Array-based Systems

Formalizing an array-based system means specifying:
▶ the set of initial states
▶ the ordering of the actions (by means of a transition relation)
▶ the set of unsafe states (the negation of the formula we want to check)

Examples (Cubicle syntax):
▶ init (i x) { T = 0.0 && B[i,x]=I && ... } \iff \forall i,x...
▶ transition spout.emit(i j) \iff \exists i,j...
requires {Tsmin<Stime[i] && SubscribedBS[j,i]=True && ...}
{L[1]:=case
| l=j : L[1]+1.0
| _ : L[1];
}
▶ unsafe(i) { L[i]>1.5 } \iff \exists i...

Symbolic representation of array-based systems: quantified first-order logic formulae.
Verification of array-based systems: decision procedure based on backward reachability.
Termination of the decision procedure:
▶ the current set of reachable states has a non-empty intersection with the set of initial states (safety check) \implies system is unsafe
▶ the current set has reached a fix-point (fix-point check) \implies system is safe
Array-based Systems

Formalizing an array-based system means specifying:
- the set of initial states
- the ordering of the actions (by means of a transition relation)
- the set of unsafe states (the negation of the formula we want to check)

Examples (Cubicle syntax):

- init (i x) \{ T = 0.0 \&\& B[i,x]=I \&\& ... \} \leadsto \forall_{i,x} ...

- transition spout.emit(i j) \leadsto \exists_{i,j} ...

  requires \{Tsmin<Stime[i] \&\& SubscribedBS[j,i]=True \&\& ...\}
  \{
    L[1]:=case
    | l=j : L[1]+1.0
    | _ : L[1];
  \}

- unsafe(i) \{ L[i]>1.5 \} \leadsto \exists_{i} ...

Symbolic representation of array-based systems: quantified first-order logic formulae. Verification of array-based systems: decision procedure based on backward reachability. Termination of the decision procedure:

- the current set of reachable states has a non-empty intersection with the set of initial states (safety check) \Rightarrow system is unsafe
- the current set has reached a fix-point (fix-point check) \Rightarrow system is safe
**Array-based Systems**

Formalizing an array-based system means specifying:
- the set of initial states
- the ordering of the actions (by means of a transition relation)
- the set of unsafe states (the negation of the formula we want to check)

**Examples (Cubicle syntax):**
- `init (i x) { T = 0.0 && B[i,x]=I && ... } \leadsto \forall i,x ...

- `transition spout.emit(i j) \leadsto \exists i,j ...

  requires \{ Tsmin<Stime[i] && SubscribedBS[j,i]=True && ... \}
  \{ L[l] := case
    | l=j : L[l]+1.0
    | _ : L[l];
  \}

- `unsafe(i) { L[i]>1.5 } \leadsto \exists i ...

**Symbolic representation of array-based systems:** *quantified first-order logic formulae.*

**Verification of array-based systems:** *decision procedure* based on *backward reachability.*

**Termination** of the decision procedure:
- the current set of reachable states has a non-empty intersection with the set of initial states (*safety check*) \( \Rightarrow \) system is unsafe
- the current set has reached a fix-point (*fix-point check*) \( \Rightarrow \) system is safe
Array-based Systems

Formalizing an array-based system means specifying:

▶ the set of initial states
▶ the ordering of the actions (by means of a transition relation)
▶ the set of unsafe states (the negation of the formula we want to check)

Examples (Cubicle syntax):

▶ init (i x) { T = 0.0 && B[i,x]=I && ... } ⇝ ∀ i,x ...

▶ transition spout_emit(i j) ⇝ ∃ i,j ...

requires {Tsmin<Stime[i] && SubscribedBS[j,i]=True && ...} 
{ 
  L[1]:=case
  | l=j : L[1]+1.0
  | _ : L[1];
}

▶ unsafe(i) { L[i]>1.5 } ⇝ ∃ i ...

Symbolic representation of array-based systems: quantified first-order logic formulae. Verification of array-based systems: decision procedure based on backward reachability. Termination of the decision procedure:

▶ the current set of reachable states has a non-empty intersection with the set of initial states (safety check) ⇒ system is unsafe
▶ the current set has reached a fix-point (fix-point check) ⇒ system is safe
Modeling assumptions

- Focus on the behavior of the queues of the bolts
- How time parameters of the topology affect the accumulation of tuples in the queues of the bolts
  - Time frequency the spouts send information to the subscribed bolts, i.e., *minimum time between two consecutive spout emits*
  - Tuples processing time for each bolt, i.e., *the time required by bolts to process a tuple (execution rate)*
- Spouts are considered sources of information; their queues are not represented
- Each bolt has one receiving queue and no sending queue
- Two approaches for abstracting the queues of the bolts:
  - Each bolt has one receiving queue for each of its parallel instances (multiple queues) ($L[i, x]$)
  - One single receiving queue is shared among all its parallel instances (shared queues) ($L[i]$)
- Usage of discrete counters for queues size changes
Modeling assumptions

- Focus on the behavior of the queues of the bolts
- How time parameters of the topology affect the accumulation of tuples in the queues of the bolts
  - Time frequency the spouts send information to the subscribed bolts, i.e. minimum time between two consecutive spout emits
  - Tuples processing time for each bolt, i.e. the time required by bolts to process a tuple (execution rate)
- Spouts are considered sources of information; their queues are not represented
- Each bolt has one receiving queue and no sending queue
- Two approaches for abstracting the queues of the bolts:
  - Each bolt has one receiving queue for each of its parallel instances (multiple queues) \( L[i, x] \)
  - One single receiving queue is shared among all its parallel instances (shared queues) \( L[i] \)
- Usage of discrete counters for queues size changes
Modeling assumptions

- Focus on the behavior of the queues of the bolts
- How time parameters of the topology affect the accumulation of tuples in the queues of the bolts
  - Time frequency the spouts send information to the subscribed bolts, i.e. *minimum time between two consecutive spout emits*
  - Tuples processing time for each bolt, i.e. *the time required by bolts to process a tuple (execution rate)*
- Spouts are considered sources of information; their queues are not represented
- Each bolt has one receiving queue and no sending queue
- Two approaches for abstracting the queues of the bolts:
  - each bolt has one receiving queue for each of its parallel instances (multiple queues) \((L[i,x])\)
  - one single receiving queue is shared among all its parallel instances (shared queues) \((L[i])\)
- Usage of discrete counters for queues size changes
Modeling assumptions

- Focus on the behavior of the queues of the bolts
- How time parameters of the topology affect the accumulation of tuples in the queues of the bolts
  - Time frequency the spouts send information to the subscribed bolts, i.e. *minimum time between two consecutive spout emits*
  - Tuples processing time for each bolt, i.e. *the time required by bolts to process a tuple (execution rate)*
- Spouts are considered sources of information; their queues are not represented
- Each bolt has one receiving queue and no sending queue
- Two approaches for abstracting the queues of the bolts:
  - Each bolt has one receiving queue for each of its parallel instances (multiple queues) ($L[i, x]$)
  - One single receiving queue is shared among all its parallel instances (shared queues) ($L[i]$)
- Usage of discrete counters for queues size changes
Modeling assumptions

- Focus on the behavior of the queues of the bolts
- How time parameters of the topology affect the accumulation of tuples in the queues of the bolts
  - Time frequency the spouts send information to the subscribed bolts, i.e. *minimum time between two consecutive spout emits*
  - Tuples processing time for each bolt, i.e. *the time required by bolts to process a tuple (execution rate)*
- Spouts are considered sources of information; their queues are not represented
- Each bolt has one receiving queue and no sending queue
- Two approaches for abstracting the queues of the bolts:
  - each bolt has one receiving queue for each of its parallel instances (multiple queues) \((L[i, x])\)
  - one single receiving queue is shared among all its parallel instances (shared queues) \((L[i])\)
- Usage of discrete counters for queues size changes
Modeling assumptions

- Focus on the behavior of the queues of the bolts
- How time parameters of the topology affect the accumulation of tuples in the queues of the bolts
  - Time frequency the spouts send information to the subscribed bolts, i.e. *minimum time between two consecutive spout emits*
  - Tuples processing time for each bolt, i.e. *the time required by bolts to process a tuple (execution rate)*
- Spouts are considered sources of information; their queues are not represented
- Each bolt has one receiving queue and no sending queue
- Two approaches for abstracting the queues of the bolts:
  - Each bolt has one receiving queue for each of its parallel instances (multiple queues) ($L[i,x]$)
  - One single receiving queue is shared among all its parallel instances (shared queues) ($L[i]$)
- Usage of discrete counters for queues size changes
Modeling assumptions

- Focus on the behavior of the queues of the bolts
- How time parameters of the topology affect the accumulation of tuples in the queues of the bolts
  - Time frequency the spouts send information to the subscribed bolts, i.e. *minimum time between two consecutive spout emits*
  - Tuples processing time for each bolt, i.e. *the time required by bolts to process a tuple (execution rate)*
- Spouts are considered sources of information; their queues are not represented
- Each bolt has one receiving queue and no sending queue
- Two approaches for abstracting the queues of the bolts:
  - each bolt has one receiving queue for each of its parallel instances (multiple queues) ($L[i, x]$)
  - one single receiving queue is shared among all its parallel instances (shared queues) ($L[i]$)
- Usage of discrete counters for queues size changes
Modeling assumptions

- Focus on the behavior of the queues of the bolts
- How time parameters of the topology affect the accumulation of tuples in the queues of the bolts
  - Time frequency the spouts send information to the subscribed bolts, i.e. *minimum time between two consecutive spout emits*
  - Tuples processing time for each bolt, i.e. *the time required by bolts to process a tuple (execution rate)*
- Spouts are considered sources of information; their queues are not represented
- Each bolt has one receiving queue and no sending queue
- Two approaches for abstracting the queues of the bolts:
  - each bolt has one receiving queue for each of its parallel instances (multiple queues) \((L[i, x])\)
  - one single receiving queue is shared among all its parallel instances (shared queues) \((L[i])\)
- Usage of discrete counters for queues size changes
Modeling assumptions

- Focus on the behavior of the queues of the bolts
- How time parameters of the topology affect the accumulation of tuples in the queues of the bolts
  - Time frequency the spouts send information to the subscribed bolts, i.e. *minimum time between two consecutive spout emits*
  - Tuples processing time for each bolt, i.e. *the time required by bolts to process a tuple (execution rate)*
- Spouts are considered sources of information; their queues are not represented
- Each bolt has one receiving queue and no sending queue
- Two approaches for abstracting the queues of the bolts:
  - each bolt has one receiving queue for each of its parallel instances (multiple queues) \((L[i, x])\)
  - one single receiving queue is shared among all its parallel instances (shared queues) \((L[i])\)
- Usage of discrete counters for queues size changes
Modeling assumptions

- Focus on the behavior of the queues of the bolts
- How time parameters of the topology affect the accumulation of tuples in the queues of the bolts
  - Time frequency the spouts send information to the subscribed bolts, i.e. *minimum time between two consecutive spout emits*
  - Tuples processing time for each bolt, i.e. *the time required by bolts to process a tuple (execution rate)*
- Spouts are considered sources of information; their queues are not represented
- Each bolt has one receiving queue and no sending queue
- Two approaches for abstracting the queues of the bolts:
  - Each bolt has one receiving queue for each of its parallel instances (multiple queues) \((L[i, x])\)
  - One single receiving queue is shared among all its parallel instances (shared queues) \((L[i])\)
- Usage of discrete counters for queues size changes
Formalization and Verification

The formalization captures the topology behavior (subscription relation, current state, modeling assumptions) through transitions:

- **discrete** transitions change the state of the topology components or updating the size of the queues of the bolts but they do not modify the value of the global time $T$

- **continuous** transition changes the value of the global time $T$ and, possibly, the states of some bolts when their processing has been terminated during the last $\delta$ time units

$$\exists 0 < \delta \land CanTimeElapse = \text{true} \land$$

$$\begin{align*}
T' &= T + \delta \\
\forall j, z \left( P'[j, z] &= \text{if } (0 \leq P[j, z] - \delta) \text{then } P[j, z] - \delta \text{ else } 0 \right) \\
B'[j, z] &= \ldots \\
\text{CanTimeElapse}' &= \text{false}
\end{align*}$$

Examples of transitions and their effect:

- **spout emit**($i, j$): $L[j]$ increases ($\text{SubscribedBS}[j, i]$); emit time of the spout ($\text{Stime}$) is reset

- **bolt emit**($i, j$): the state of $B[i]$ is changed into idle and $L[j]$ is incremented by 1

- **bolt take**($j, y$): $L[j]$ is decreased by 1 and the percentage of tuple processing of the thread receiving the tuple ($P[j, y]$) is set to 1

**Formalization and verification** was performed in the same framework: MCMT (http://users.mat.unimi.it/users/ghilardi/mcmt/), respectively Cubicle (http://cubicle.lri.fr/).
Formalization and Verification

The formalization captures the topology behavior (subscription relation, current state, modeling assumptions) through transitions:

- **discrete** transitions change the state of the topology components or updating the size of the queues of the bolts but they do not modify the value of the global time $T$

- **continuous** transition changes the value of the global time $T$ and, possibly, the states of some bolts when their processing has been terminated during the last $\delta$ time units

\[
\exists \delta > 0 \land \text{CanTimeElapse} = \text{true} \land \\
\begin{aligned}
\frac{\delta}{T'} &= T + \delta \\
\forall j (P'[j, z]) &= \text{if } (0 \leq P[j, z] - \delta) \text{then } P[j, z] - \delta \text{ else } 0 \\
\forall j (B'[j, z]) &= \text{...} \\
\text{CanTimeElapse}' &= \text{false}
\end{aligned}
\]

Examples of transitions and their effect:

- **spout\_emit(i, j)**: $L[j]$ increases ($\text{SubscribedBS}[j, i]$); emit time of the spout ($\text{Stime}$) is reset

- **bolt\_emit(i, j)**: the state of $B[i]$ is changed into idle and $L[j]$ is incremented by 1

- **bolt\_take(j, y)**: $L[j]$ is decreased by 1 and the percentage of tuple processing of the thread receiving the tuple ($P[j, y]$) is set to 1

Formalization and verification was performed in the same framework: MCMT (http://users.mat.unimi.it/users/ghilardi/mcmt/), respectively Cubicle (http://cubicle.lri.fr/).
Formalization and Verification

The formalization captures the topology behavior (subscription relation, current state, modeling assumptions) through transitions:

- **discrete** transitions change the state of the topology components or updating the size of the queues of the bolts but they do not modify the value of the global time \( T \)

- **continuous** transition changes the value of the global time \( T \) and, possibly, the states of some bolts when their processing has been terminated during the last \( \delta \) time units

\[
\exists \delta > 0 \land CanTimeElapse = true \land \\
\forall \left( \begin{array}{l}
T' = T + \delta \\
P'[j, z] = \text{if } (0 \leq P[j, z] - \delta) \text{then } P[j, z] - \delta \text{ else } 0 \\
B'[j, z] \ldots \\
CanTimeElapse' = false
\end{array} \right)
\]

Examples of transitions and their effect:

- **spout\_emit(i, j)**: \( L[j] \) increases (\( SubscribedBS[j, i] \)); emit time of the spout (\( Stime \)) is reset

- **bolt\_emit(i, j)**: the state of \( B[i] \) is changed into \textit{idle} and \( L[j] \) is incremented by 1

- **bolt\_take(j, y)**: \( L[j] \) is decreased by 1 and the percentage of tuple processing of the thread receiving the tuple \( (P[j, y]) \) is set to 1

Formalization and verification was performed in the same framework: MCMT (http://users.mat.unimi.it/users/ghilardi/mcmt/), respectively Cubicle (http://cubicle.lri.fr/).
Formalization and Verification

The formalization captures the topology behavior (subscription relation, current state, modeling assumptions) through transitions:

- **discrete** transitions change the state of the topology components or updating the size of the queues of the bolts but they do not modify the value of the global time \( T \)
- **continuous** transition changes the value of the global time \( T \) and, possibly, the states of some bolts when their processing has been terminated during the last \( \delta \) time units

\[
\exists \frac{\delta}{\delta} 0 < \delta \wedge \text{CanTimeElapse} = \text{true} \wedge \\
\forall j, z \begin{pmatrix} \left( T' = T + \delta \\ P'[j, z] = \text{if } (0 \leq P[j, z] - \delta) \text{then } P[j, z] - \delta \text{ else } 0 \\ B'[j, z] \\
\text{CanTimeElapse'} = \text{false} \end{pmatrix}
\]

Examples of transitions and their effect:

- **spout\_{emit}(i, j)**: \( L[j] \) increases (\( \text{SubscribedBS}[j, i] \)); emit time of the spout (\( \text{Stime} \)) is reset
- **bolt\_{emit}(i, j)**: the state of \( B[i] \) is changed into \text{idle} and \( L[j] \) is incremented by 1
- **bolt\_{take}(j, y)**: \( L[j] \) is decreased by 1 and the percentage of tuple processing of the thread receiving the tuple (\( P[j, y] \)) is set to 1

Formalization and verification was performed in the same framework: MCMT (http://users.mat.unimi.it/users/ghilardi/mcmt/), respectively Cubicle (http://cubicle.lri.fr/).
Formalization and Verification

The formalization captures the topology behavior (subscription relation, current state, modeling assumptions) through transitions:

- **discrete** transitions change the state of the topology components or updating the size of the queues of the bolts but they do not modify the value of the global time $T$
- **continuous** transition changes the value of the global time $T$ and, possibly, the states of some bolts when their processing has been terminated during the last $\delta$ time units

$$\exists \delta > 0 \land \text{CanTimeElapse} = \text{true} \land$$

$$\forall (T', P'[j, z], B'[j, z], \ldots) = \begin{cases} T + \delta & \text{if } (0 \leq P[j, z] - \delta) \text{then } P[j, z] - \delta \text{ else } 0 \\ \text{CanTimeElapse}' = \text{false} & \end{cases}$$

Examples of transitions and their effect:

- **spout Emit** $(i, j)$: $L[j]$ increases ($\text{SubscribedBS}[j, i]$); emit time of the spout ($\text{Stime}$) is reset
- **bolt Emit** $(i, j)$: the state of $B[i]$ is changed into $\text{idle}$ and $L[j]$ is incremented by 1
- **bolt Take** $(j, y)$: $L[j]$ is decreased by 1 and the percentage of tuple processing of the thread receiving the tuple ($P[j, y]$) is set to 1

Formalization and verification were performed in the same framework: MCMT (http://users.mat.unimi.it/users/ghilardi/mcmt/), respectively Cubicle (http://cubicle.lri.fr/).
Formalization and Verification

The **formalization** captures the topology behavior (subscription relation, current state, modeling assumptions) through transitions:

- **discrete** transitions change the state of the topology components or updating the size of the queues of the bolts but they do not modify the value of the global time $T$

- **continuous** transition changes the value of the global time $T$ and, possibly, the states of some bolts when their processing has been terminated during the last $\delta$ time units

\[
\exists \delta > 0 \land \text{CanTimeElapse} = \text{true} \land \\
\forall j, z \left( \\
\begin{align*}
T' &= T + \delta \\
P'[j, z] &= \text{if } (0 \leq P[j, z] - \delta) \text{then } P[j, z] - \delta \text{ else } 0 \\
B'[j, z] &= \ldots \\
\text{CanTimeElapse}' &= \text{false}
\end{align*}
\right)
\]

Examples of transitions and their effect:

- **spout**$_{\text{emit}}(i, j)$: $L[j]$ increases ($\text{SubscribedBS}[j, i]$); emit time of the spout ($Stime$) is reset

- **bolt**$_{\text{emit}}(i, j)$: the state of $B[i]$ is changed into $\text{idle}$ and $L[j]$ is incremented by 1

- **bolt**$_{\text{take}}(j, y)$: $L[j]$ is decreased by 1 and the percentage of tuple processing of the thread receiving the tuple ($P[j, y]$) is set to 1

Formalization and verification was performed in the same framework: MCMT (http://users.mat.unimi.it/users/ghilardi/mcmt/), respectively Cubicle (http://cubicle.lri.fr/).
Formalization and Verification

The formalization captures the topology behavior (subscription relation, current state, modeling assumptions) through transitions:

- **discrete** transitions change the state of the topology components or updating the size of the queues of the bolts but they do not modify the value of the global time $T$
- **continuous** transition changes the value of the global time $T$ and, possibly, the states of some bolts when their processing has been terminated during the last $\delta$ time units

$$\exists \delta > 0 \land CanTimeElapse = true \land$$

$$\forall j, z \left(\begin{array}{c}
T' = T + \delta \\
p'[j, z] = \text{if } (0 \leq p[j, z] - \delta) \text{then } p[j, z] - \delta \text{ else } 0 \\
b'[j, z] = \ldots \\
CanTimeElapse' = false
\end{array}\right)$$

Examples of transitions and their effect:

- **spout** emit $(i, j)$: $L[j]$ increases ($SubscribedBS[j, i]$); emit time of the spout ($Stime$) is reset
- **bolt** emit $(i, j)$: the state of $B[i]$ is changed into $\text{idle}$ and $L[j]$ is incremented by 1
- **bolt** take $(j, y)$: $L[j]$ is decreased by 1 and the percentage of tuple processing of the thread receiving the tuple ($P[j, y]$) is set to 1

Formalization and verification was performed in the same framework: MCMT (http://users.mat.unimi.it/users/ghilardi/mcmt/), respectively Cubicle (http://cubicle.lri.fr/).
Formalization and Verification

The formalization captures the topology behavior (subscription relation, current state, modeling assumptions) through transitions:

- **discrete** transitions change the state of the topology components or updating the size of the queues of the bolts but they do not modify the value of the global time $T$
- **continuous** transition changes the value of the global time $T$ and, possibly, the states of some bolts when their processing has been terminated during the last $\delta$ time units

\[
\exists \delta > 0 \land \text{CanTimeElapse} = \text{true} \land \\
\left( T' = T + \delta, P'[j, z] = \begin{cases} P[j, z] - \delta & \text{if } 0 \leq P[j, z] - \delta \\ 0 & \text{else} \end{cases}, B'[j, z] = \ldots, \text{CanTimeElapse}' = \text{false} \right)
\]

Examples of transitions and their effect:

- **spout\_emit\(_i,j\)**: $L[j]$ increases ($SubscribedBS[j, i]$); emit time of the spout ($Stime$) is reset
- **bolt\_emit\(_i,j\)**: the state of $B[i]$ is changed into idle and $L[j]$ is incremented by 1
- **bolt\_take\(_j,y\)**: $L[j]$ is decreased by 1 and the percentage of tuple processing of the thread receiving the tuple ($P[j, y]$) is set to 1

Formalization and verification was performed in the same framework: MCMT (http://users.mat.unimi.it/users/ghilardi/mcmt/), respectively Cubicle (http://cubicle.lri.fr/).
Challenges

- **Nondeterministic updates**

\[
\exists x, y, i, j \quad \text{statechange} = \text{true} \land \\
\left( \begin{array}{l}
\text{statechange}' = \text{false} \\
\ldots \\
B'[l, z] = \\
\forall l, k, z \quad \begin{cases}
\text{true} & \text{if } (z = y \land l = j \land B[j, y] = E) \\
\text{else} & \\
\end{cases}
\end{array} \right)
\]

- **Reducing the dimension of the search space**

  - spout states were left out; only the time elapsing to enable spout emit is considered
  - bolt queues have only one dimension (shared queue)

\[
\exists i, j, x \quad T_{\text{min}} < Stime[j] \land \text{SubscribedBS}[j, i] = \text{true} \land \\
\left( \begin{array}{l}
\text{true} \quad \begin{cases}
L'[l] = \text{if } (l = j) \text{ then } L[l] + 1 \text{ else } L[l] \\
\text{then } \\
\end{cases} \\
\forall L' \quad \begin{cases}
\text{true} \quad \begin{cases}
\text{else } & \\
\end{cases} \\
\end{cases}
\end{array} \right)
\]

- **Incorrect firing of transitions:** the implemented backward reachability algorithm lacks the so-called *urgent transitions*.

**Our case:** simulation of urgent transitions via flags; bolt emit and take are urgent wrt spout emits.

- **Number of transitions** limited by:

  - the emit state of a bolt is enforced if a bolt is ready to emit
  - state take omitted
  - restrict the reachability analysis only to one bolt (bolt 1) of the system
Challenges

▶ Nondeterministic updates

\[ \exists x, y, i, j \text{ statechange} = \text{true} \land \]
\[ \text{statechange}' = \text{false} \]
\[ \ldots \]
\[ \forall l, k, z \left( B'[l, z] = \begin{cases} \text{if } (z = y \land l = j \land B[j, y] = E) \text{ then } (I \text{ or } K) \text{ else } B[l, z] \\ \text{elseif } \ldots \end{cases} \right) \]
\[ \text{CanTimeElapse}' = \text{true} \]

▶ Reducing the dimension of the search space

▶ spout states were left out; only the time elapsing to enable spout emit is considered
▶ bolt queues have only one dimension (shared queue)

\[ \exists i, j, x \text{ Ts}_{\text{min}} < \text{Stime}[i] \land \text{SubscribedBS}[j, i] = \text{true} \land \ldots \]
\[ \left( L'[l] = \begin{cases} \text{if } (l = j) \text{ then } L[l] + 1 \text{ else } L[l] \\ \text{elseif } \ldots \end{cases} \right) \]
\[ \forall l \left( \text{Stime}'[l] = \begin{cases} \text{if } (l = i) \text{ then } 0 \text{ else } \text{Stime}[l] \end{cases} \right) \]

▶ Incorrect firing of transitions: the implemented backward reachability algorithm lacks the so-called urgent transitions.
Our case: simulation of urgent transitions via flags; bolt emit and take are urgent wrt spout emits.

▶ Number of transitions limited by:
▶ the emit state of a bolt is enforced if a bolt is ready to emit
▶ state take omitted
▶ restrict the reachability analysis only to one bolt (bolt 1) of the system
Challenges

- **Nondeterministic updates**

  \[\exists_{x,y,i,j} \text{statechange} = \text{true} \land \]
  \[
  \begin{cases}
  \text{statechange}' = \text{false} \\
  \ldots \\
  B'[l, z] = \begin{cases}
    \text{if } (z=y \land l=j \land B[j, y]=E) & \text{then } (I \text{ or } K) \text{ else } B[l, z] \\
    \text{elseif } \ldots \\
    \text{CanTimeElapse}' = \text{true}
  \end{cases}
  \end{cases}
  \]

- **Reducing the dimension of the search space**
  - spout states were left out; only the time elapsing to enable spout emit is considered
  - bolt queues have only one dimension (shared queue)

  \[\exists_{i,j,x} T_{\text{min}}<\text{Stime}[i] \land \text{SubscribedBS}[j, i]=\text{true} \land \ldots \]

  \[
  \begin{cases}
  L'[l] = \begin{cases}
    \text{if } (l=j) & \text{then } L[l]+1 \text{ else } L[l] \\
    \ldots \\
    \text{elseif } \ldots \\
    \text{Stime}'[l] = \begin{cases}
      \text{if } (l=i) & \text{then } 0 \text{ else } \text{Stime}[l] \\
      \ldots
    \end{cases}
  \end{cases}
  \end{cases}
  \]

- **Incorrect firing of transitions**: the implemented backward reachability algorithm lacks the so-called *urgent transitions*.

  Our case: simulation of urgent transitions via flags; bolt emit and take are urgent wrt spout emits.

- **Number of transitions** limited by:
  - the emit state of a bolt is enforced if a bolt is ready to emit
  - state take omitted
  - restrict the reachability analysis only to one bolt (bolt 1) of the system
Challenges

▶ Nondeterministic updates

\[ \exists x, y, i, j \quad \text{statechange} = \text{true} \wedge \]
\[ \begin{cases} 
\text{statechange}' = \text{false} \\
\ldots \\
B'[l, z] = \begin{cases} 
\text{if } (z = y \wedge l = j \wedge B[j, y] = \text{E}) & \text{then } (I \text{ or } K) \\
\text{else } B[l, z] & \text{elseif } \ldots 
\end{cases} \\
\text{CanTimeElapse}' = \text{true} 
\end{cases} \]

▶ Reducing the dimension of the search space

▶ spout states were left out; only the time elapsing to enable spout emit is considered
▶ bolt queues have only one dimension (shared queue)

\[ \exists i, j, x \quad Ts_{\text{min}} < Stime[i] \wedge \text{SubscribedBS}[j, i] = \text{true} \wedge \ldots \\
\begin{cases} 
L'[l] = \begin{cases} 
\text{if } (l = j) & \text{then } L[l] + 1 \\
\text{else } L[l] & \text{elseif } \ldots 
\end{cases} \\
Stime'[l] = \begin{cases} 
\text{if } (l = i) & \text{then } 0 \\
\text{else } Stime[l] & \text{elseif } \ldots 
\end{cases} 
\end{cases} \]

▶ Incorrect firing of transitions: the implemented backward reachability algorithm lacks the so-called urgent transitions.
Our case: simulation of urgent transitions via flags; bolt emit and take are urgent wrt spout emits.

▶ Number of transitions limited by:
▶ the emit state of a bolt is enforced if a bolt is ready to emit
▶ state take omitted
▶ restrict the reachability analysis only to one bolt (bolt 1) of the system
Challenges

- **Nondeterministic updates**

\[ \exists x, y, i, j \quad \text{statechange} = \text{true} \land \]
\[
\left( \text{statechange}' = \text{false} \right) \implies \]
\[
\left( \forall l, k, z \quad B'[l, z] = \begin{cases} 
  \text{if } (z = y \land l = j \land B[j, y] = E) & \text{then } (I \lor K) \text{ else } B[l, z] \\
  \text{elseif } \ldots
\end{cases} \right) \]
\[
\text{CanTimeElapse}' = \text{true} \]

- **Reducing the dimension of the search space**
  - spout states were left out; only the time elapsing to enable spout emit is considered
  - bolt queues have only one dimension (shared queue)

\[ \exists i, j, x \quad T_{\text{min}} < \text{Stime}[i] \land \text{SubscribedBS}[j, i] = \text{true} \land \ldots \]
\[
\left( L'[l] = \begin{cases} 
  \text{if } (l = j) & \text{then } L[l] + 1 \text{ else } L[l] \\
  \text{elseif } \ldots
\end{cases} \right) \]
\[
\forall l, k, z \quad \left( \text{Stime}'[l] = \begin{cases} 
  \text{if } (l = i) & \text{then } 0 \text{ else } \text{Stime}[l] \\
  \text{elseif } \ldots
\end{cases} \right) \]

- **Incorrect firing of transitions**: the implemented backward reachability algorithm lacks the so-called *urgent transitions*. 
**Our case**: simulation of urgent transitions via flags; bolt emit and take are urgent wrt spout emits.

- **Number of transitions limited by**:
  - the emit state of a bolt is enforced if a bolt is ready to emit
  - state take omitted
  - restrict the reachability analysis only to one bolt (bolt 1) of the system
Challenges

- Nondeterministic updates

\[ \exists \ x,y,i,j \quad \text{statechange} = \text{true} \land \]
\[ \begin{align*}
\text{statechange}' &= \text{false} \\
\ldots \\
B'[l,z] &= \begin{cases}
\text{if } (z=y \land l=j \land B[j,y]=E) \\
\text{then } (I \lor K) \text{ else } B[l,z] \\
\text{elseif } \ldots \\
\text{CanTimeElapse}' &= \text{true}
\end{cases}
\end{align*} \]

- Reducing the dimension of the search space
  - spout states were left out; only the time elapsing to enable spout emit is considered
  - bolt queues have only one dimension (shared queue)

\[ \exists \ i,j,x \quad Ts_{\min}<Stime[i] \land SubscribedBS[j, i]=\text{true} \land \ldots \]
\[ \begin{align*}
L'[l] &= \begin{cases}
\text{if } (l=j) \text{ then } L[l]+1 \text{ else } L[l]
\end{cases} \\
\forall \ l \quad Stime'[l] &= \begin{cases}
\text{if } (l=i) \text{ then } 0 \text{ else } Stime[l]
\end{cases} \\
\ldots
\end{align*} \]

- Incorrect firing of transitions: the implemented backward reachability algorithm lacks the so-called \textit{urgent transitions}.

\textbf{Our case:} simulation of urgent transitions via flags; bolt emit and take are urgent \textit{wrt} spout emits.

- Number of transitions limited by:
  - the emit state of a bolt is enforced if a bolt is ready to emit
  - state take omitted
  - restrict the reachability analysis only to one bolt (bolt 1) of the system
Challenges

- **Nondeterministic updates**

\[
\exists \ x, y, i, j \quad \begin{cases}
\text{statechange} = \text{true} \land \\
\text{statechange}' = \text{false} \\
\ldots \\
B'[l, z] = \begin{cases}
\text{if} \ (z = y \land l = j \land B[j, y] = E) \text{ then } (I \text{ or } K) \text{ else } B[l, z] \\
\text{elseif} \ldots \\
\text{CanTimeElapse}' = \text{true}
\end{cases}
\end{cases}
\]

- **Reducing the dimension of the search space**

  - spout states were left out; only the time elapsing to enable spout emit is considered
  - bolt queues have only one dimension (shared queue)

\[
\exists \ i, j, x \quad \begin{cases}
Ts_{\text{min}} < Stime[i] \land SubscribedBS[j, i] = \text{true} \land \\
L'[l] = \begin{cases}
\text{if} \ (l = j) \text{ then } L[l] + 1 \text{ else } L[l] \\
\end{cases} \\
Stime'[l] = \begin{cases}
\text{if} \ (l = i) \text{ then } 0 \text{ else } Stime[l] \\
\ldots
\end{cases}
\end{cases}
\]

- **Incorrect firing of transitions**: the implemented backward reachability algorithm lacks the so-called *urgent transitions*.
  
  **Our case**: simulation of urgent transitions via flags; bolt emit and take are urgent wrt spout emits.

- **Number of transitions** limited by:
  - the emit state of a bolt is enforced if a bolt is ready to emit
  - state take omitted
  - restrict the reachability analysis only to one bolt (bolt 1) of the system
Challenges

▶ Nondeterministic updates

\[
\exists x, y, i, j, \exists statechange = true \land
\begin{align*}
    statechange' &= false \\
    \forall l, k, z \left( B'[l, z] \right) &= \begin{cases} 
    \text{if} \ (z = y \land l = j \land B[j, y] = E) & \text{then} (I \lor K) \text{ else } B[l, z] \\
    \text{elseif} & \ldots \\
    CanTimeElapse' &= true
\end{cases}
\end{align*}
\]

▶ Reducing the dimension of the search space

▶ spout states were left out; only the time elapsing to enable spout emit is considered
▶ bolt queues have only one dimension (shared queue)

\[
\exists T_{\min} < Stime[i] \land SubscribedBS[j, i] = true \land \ldots
\]

\[
\begin{align*}
    L'[l] &= \begin{cases} 
    \text{if} \ (l = j) & \text{then } L[l] + 1 \text{ else } L[l] \\
    \ldots
\end{cases} \\
    Stime'[l] &= \begin{cases} 
    \text{if} \ (l = i) & \text{then } 0 \text{ else } Stime[l] \\
    \ldots
\end{cases}
\end{align*}
\]

▶ Incorrect firing of transitions: the implemented backward reachability algorithm lacks the so-called urgent transitions.

Our case: simulation of urgent transitions via flags; bolt emit and take are urgent wrt spout emits.

▶ Number of transitions limited by:

▶ the emit state of a bolt is enforced if a bolt is ready to emit
▶ state take omitted
▶ restrict the reachability analysis only to one bolt (bolt 1) of the system
Challenges

- **Nondeterministic updates**

\[
\exists \ x, y, i, j \text{ such that}\]
\[
\exists statechange = \text{true} \land
\]
\[
\left(\begin{array}{c}
statechange' = \text{false} \\
B'[l, z] = \text{if } (z=y \land l=j \land B[j, y]=E) \text{ then } (I \text{ or } K) \text{ else } B[l, z] \\
\text{CanTimeElapse'} = \text{true}
\end{array}\right)
\]

- **Reducing the dimension of the search space**
  - spout states were left out; only the time elapsing to enable spout emit is considered
  - bolt queues have only one dimension (shared queue)

\[
\exists \ i, j, x \text{ such that}\]
\[
T_{\text{min}} < Stime[i] \land \text{SubscribedBS}[j, i]=\text{true} \land \ldots
\]
\[
\forall l \left(L'[l] = \text{if } (l=j) \text{ then } L[l]+1 \text{ else } L[l]\right)
\]
\[
\forall l \left(Stime'[l] = \text{if } (l=i) \text{ then } 0 \text{ else } Stime[l]\right)
\]

- **Incorrect firing of transitions**: the implemented backward reachability algorithm lacks the so-called *urgent transitions*.

**Our case**: simulation of urgent transitions via flags; bolt emit and take are urgent wrt spout emits.

- **Number of transitions** limited by:
  - the emit state of a bolt is enforced if a bolt is ready to emit
  - state take omitted
  - restrict the reachability analysis only to one bolt (bolt 1) of the system
Challenges

- **Nondeterministic updates**

\[
\exists x, y, i, j \quad \text{statechange} = \text{true} \land \\
\left(\text{statechange}' = \text{false} \land \ldots \begin{align*}
B'[l, z] &= \begin{cases}
\text{if (z=y \land l=j \land B[j, y]=E)} & \text{then (I or K) else } B[l, z] \\
\text{elseif} & \ldots 
\end{cases} \\
\text{CanTimeElapse}' &= \text{true}
\right)
\]

- **Reducing the dimension of the search space**
  - spout states were left out; only the time elapsing to enable spout emit is considered
  - bolt queues have only one dimension (shared queue)

\[
\exists i, j, x \quad Ts_{min} < Stime[i] \land SubscribedBS[j, i]=\text{true} \land \ldots \\
\begin{align*}
L'[l] &= \begin{cases}
\text{if (l=j)} & \text{then } L[l]+1 \text{ else } L[l] \\
\text{elseif} & \ldots 
\end{cases} \\
Stime'[l] &= \begin{cases}
\text{if (l=i)} & \text{then } 0 \text{ else } Stime[l] \\
\text{elseif} & \ldots 
\end{cases}
\end{align*}
\]

- **Incorrect firing of transitions**: the implemented backward reachability algorithm lacks the so-called *urgent transitions*.  
  **Our case**: simulation of urgent transitions via flags; bolt emit and take are urgent wrt spout emits.

- **Number of transitions** limited by:
  - the emit state of a bolt is enforced if a bolt is ready to emit
  - state take omitted
  - restrict the reachability analysis only to one bolt (bolt 1) of the system
Experimental results

First attempt:

- $L[1] \geq 3$ and $Tsmin < 1$ – expected result: UNSAFE
  
  Trace: \textit{Init} $\rightarrow$ \textit{time\_elapsed} $\rightarrow$ \textit{setDoTake}$_{\text{False}}$ $\rightarrow$ \textit{setDoEmit}$_{\text{False}}$ $\rightarrow$ \textit{spout\_emit}$ \rightarrow$ \textit{time\_elapsed} $\rightarrow$ \textit{setDoTake}$_{\text{True}}$ $\rightarrow$ \textit{setDoEmit}$_{\text{False}}$ $\rightarrow$ \textit{bolt1\_take}$ \rightarrow$ \textit{setDoTake}$_{\text{False}}$ $\rightarrow$ \textit{setDoEmit}$_{\text{False}}$ $\rightarrow$ \textit{spout\_emit}$ \rightarrow$ \textit{time\_elapsed} $\rightarrow$ \textit{setDoTake}$_{\text{False}}$ $\rightarrow$ \textit{setDoEmit}$_{\text{False}}$ $\rightarrow$ \textit{spout\_emit}$ \rightarrow$ $L[1] \geq 2$

- $L[1] \geq 3$ and $Tsmin \geq 1$ – expected result: SAFE

Result: the verification problems lead to memory exhaustion.

Second attempt:

- $L[1] \geq 2$ and $Tsmin < 1$ – obtained result: UNSAFE

- $L[1] \geq 2$ and $Tsmin \geq 1$ – expected result: SAFE
Experimental results

First attempt:

- $L[1] \geq 3$ and $Ts\min < 1$ – expected result: UNSAFE
  
  Trace: $Init \rightarrow time\_elapse \rightarrow setDoTake_{False} \rightarrow setDoEmit_{False} \rightarrow spout_{emit} \rightarrow time\_elapse \rightarrow setDoTake_{True} \rightarrow setDoEmit_{False} \rightarrow bolt1\_take \rightarrow setDoTake_{False} \rightarrow setDoEmit_{False} \rightarrow spout_{emit} \rightarrow time\_elapse \rightarrow setDoTake_{False} \rightarrow setDoEmit_{False} \rightarrow spout_{emit} \rightarrow L[1] \geq 2$

- $L[1] \geq 3$ and $Ts\min \geq 1$ – expected result: SAFE

Result: the verification problems lead to memory exhaustion.

Second attempt:

- $L[1] \geq 2$ and $Ts\min < 1$ – obtained result: UNSAFE
- $L[1] \geq 2$ and $Ts\min \geq 1$ – expected result: SAFE
Experimental results

First attempt:

- $L[1] \geq 3$ and $T_{smin} < 1$ – expected result: UNSAFE
  
  Trace: $Init \rightarrow time\_elapse \rightarrow setDoTake_{False} \rightarrow setDoEmit_{False} \rightarrow spout_{emit} \rightarrow time\_elapse \rightarrow setDoTake_{True} \rightarrow setDoEmit_{False} \rightarrow bolt1_{take} \rightarrow setDoTake_{False} \rightarrow setDoEmit_{False} \rightarrow spout_{emit} \rightarrow time\_elapse \rightarrow setDoTake_{False} \rightarrow spout_{emit} \rightarrow l[1] \geq 2$

- $L[1] \geq 3$ and $T_{smin} \geq 1$ – expected result: SAFE

Result: the verification problems lead to memory exhaustion.

Second attempt:

- $L[1] \geq 2$ and $T_{smin} < 1$ – obtained result: UNSAFE

- $L[1] \geq 2$ and $T_{smin} \geq 1$ – expected result: SAFE
Experimental results

First attempt:

- $L[1] \geq 3$ and $Tsmin < 1$ – expected result: UNSAFE

  Trace:  
  Init $\rightarrow$ time\_elapse $\rightarrow$ setDoTake\_False $\rightarrow$ setDoEmit\_False $\rightarrow$ spout\_emit $\rightarrow$ time\_elapse $\rightarrow$ setDoTake\_True $\rightarrow$ setDoEmit\_False $\rightarrow$ bolt1\_take $\rightarrow$ setDoTake\_False $\rightarrow$ setDoEmit\_False $\rightarrow$ spout\_emit $\rightarrow$ time\_elapse $\rightarrow$ setDoTake\_False $\rightarrow$ setDoEmit\_False $\rightarrow$ spout\_emit $\rightarrow$ $L[1] \geq 2$

- $L[1] \geq 3$ and $Tsmin \geq 1$ – expected result: SAFE

Result: the verification problems lead to memory exhaustion.

Second attempt:

- $L[1] \geq 2$ and $Tsmin < 1$ – obtained result: UNSAFE

- $L[1] \geq 2$ and $Tsmin \geq 1$ – expected result: SAFE
Experimental results

First attempt:

- \( L[1] \geq 3 \) and \( Tsmin < 1 \) – expected result: UNSAFE

  Trace: \( Init \rightarrow time\_elapse \rightarrow setDoTake_{False} \rightarrow setDoEmit_{False} \rightarrow spout_{emit} \rightarrow time\_elapse \rightarrow setDoTake_{True} \rightarrow setDoEmit_{False} \rightarrow bolt1_{take} \rightarrow setDoTake_{False} \rightarrow setDoEmit_{False} \rightarrow spout_{emit} \rightarrow time\_elapse \rightarrow setDoTake_{False} \rightarrow setDoEmit_{False} \rightarrow spout_{emit} \rightarrow L[1] \geq 2 \)

- \( L[1] \geq 3 \) and \( Tsmin \geq 1 \) – expected result: SAFE

Result: the verification problems lead to memory exhaustion.

Second attempt:

- \( L[1] \geq 2 \) and \( Tsmin < 1 \) – obtained result: UNSAFE

- \( L[1] \geq 2 \) and \( Tsmin \geq 1 \) – expected result: SAFE
Experimental results

First attempt:

- $L[1] \geq 3$ and $Tsmin < 1$ – expected result: UNSAFE

  Trace: $Init \rightarrow time\_elapse \rightarrow setDoTake_{False} \rightarrow setDoEmit_{False} \rightarrow spout_{emit} \rightarrow time\_elapse \rightarrow setDoTake_{True} \rightarrow setDoEmit_{False} \rightarrow bolt1_{take} \rightarrow setDoTake_{False} \rightarrow setDoEmit_{False} \rightarrow spout_{emit} \rightarrow time\_elapse \rightarrow setDoTake_{False} \rightarrow setDoEmit_{False} \rightarrow spout_{emit} \rightarrow L[1] \geq 2$

- $L[1] \geq 3$ and $Tsmin \geq 1$ – expected result: SAFE

Result: the verification problems lead to memory exhaustion.

Second attempt:

- $L[1] \geq 2$ and $Tsmin < 1$ – obtained result: UNSAFE

- $L[1] \geq 2$ and $Tsmin \geq 1$ – expected result: SAFE
Experimental results

First attempt:
- $L[1] \geq 3$ and $Tsmin < 1$ – expected result: UNSAFE

Trace:
- $\text{Init} \rightarrow \text{time\_elapse} \rightarrow \text{setDoTake}_\text{False} \rightarrow \text{setDoEmit}_\text{False} \rightarrow \text{spout}_\text{emit} \rightarrow \text{time\_elapse} \rightarrow \text{setDoTake}_\text{True} \rightarrow \text{setDoEmit}_\text{False} \rightarrow \text{bolt1}_\text{take} \rightarrow \text{setDoTake}_\text{False} \rightarrow \text{setDoEmit}_\text{False} \rightarrow \text{spout}_\text{emit} \rightarrow \text{time\_elapse} \rightarrow \text{setDoTake}_\text{False} \rightarrow \text{setDoEmit}_\text{False} \rightarrow \text{spout}_\text{emit} \rightarrow L[1] \geq 2$

- $L[1] \geq 3$ and $Tsmin \geq 1$ – expected result: SAFE

Result: the verification problems lead to memory exhaustion.

Second attempt:
- $L[1] \geq 2$ and $Tsmin < 1$ – obtained result: UNSAFE
- $L[1] \geq 2$ and $Tsmin \geq 1$ – expected result: SAFE
Experimental results

First attempt:

- $L[1] \geq 3$ and $Tsmin < 1$ – expected result: UNSAFE

  Trace: $Init \rightarrow time\_elapse \rightarrow setDoTake_{\text{False}} \rightarrow setDoEmit_{\text{False}} \rightarrow spout_{\text{emit}} \rightarrow time\_elapse \rightarrow setDoTake_{\text{True}} \rightarrow setDoEmit_{\text{False}} \rightarrow bolt_{1\text{take}} \rightarrow setDoTake_{\text{False}} \rightarrow setDoEmit_{\text{False}} \rightarrow spout_{\text{emit}} \rightarrow time\_elapse \rightarrow setDoTake_{\text{False}} \rightarrow setDoEmit_{\text{False}} \rightarrow spout_{\text{emit}} \rightarrow L[1] \geq 2$

- $L[1] \geq 3$ and $Tsmin \geq 1$ – expected result: SAFE

Result: the verification problems lead to memory exhaustion.

Second attempt:

- $L[1] \geq 2$ and $Tsmin < 1$ – obtained result: UNSAFE

- $L[1] \geq 2$ and $Tsmin \geq 1$ – expected result: SAFE
Current and Future Work

- Refinements of the presented model, linear topologies limiting the analysis to well-founded transition systems
- New model to capturing relevant properties of distributed systems, e.g. tuple order is compatible with tuple time
Current and Future Work

- Refinements of the presented model, linear topologies → limiting the analysis to well-founded transition systems
- New model to capturing relevant properties of distributed systems, e.g. tuple order is compatible with tuple time