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

Marcello M. Bersani[1] Mădălina Erașcu[2] Francesco Marconi[1]
Silvio Ghilardi[3] Matteo Rossi[1]

[1]DEIB, Politecnico di Milano, Milano, Italy

[2]West University of Timişoara and Institute eAustria,
Timişoara, Romania

[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

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)

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`) 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.

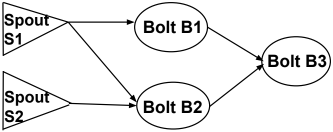
Key concepts:

- 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:

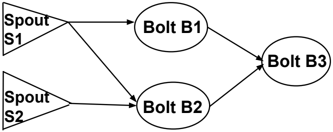
- parametric in the number of nodes and processes
- the number of nodes is known at design-time, hence fixed

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:

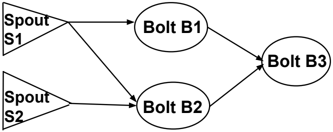
- parametric in the number of nodes and processes
- the number of nodes is known at design-time, hence fixed

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:

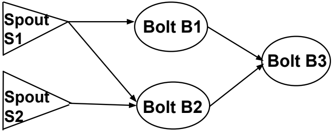
- parametric in the number of nodes and processes
- the number of nodes is known at design-time, hence fixed

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:

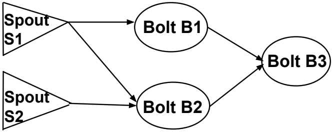
- parametric in the number of nodes and processes
- the number of nodes is known at design-time, hence fixed

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:

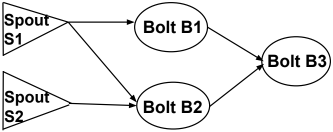
- parametric in the number of nodes and processes
- the number of nodes is known at design-time, hence fixed

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:

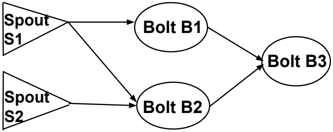
- parametric in the number of nodes and processes
- the number of nodes is known at design-time, hence fixed

Infinite-state model checking!

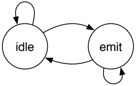
Suitable abstraction: **array-based systems.** (Ghilardi et al.)

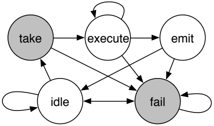
Safety verification of Storm topologies: given queue(s) bound(s) defined by the designer, "all bolt queues have a limited occupation level".

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 && ...  }` $\rightsquigarrow \underset{i,x}{\forall} \ldots$
- `transition spout_emit(i j)` $\rightsquigarrow \underset{i,j}{\exists} \ldots$

  ```
  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 }` $\rightsquigarrow \underset{i}{\exists} \ldots$

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):

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

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

  $\rightsquigarrow \underset{i,j}{\exists} \ldots$
- `unsafe(i) { L[i]>1.5 }` $\rightsquigarrow \underset{i}{\exists} \ldots$

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):

- `init (i x) { T = 0.0 && B[i,x]=I && ... }` $\rightsquigarrow \forall_{i,x} \ldots$
- `transition spout_emit(i j)` $\rightsquigarrow \exists_{i,j} \ldots$

  ```
  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 }` $\rightsquigarrow \exists_{i} \ldots$

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):

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

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

$\rightsquigarrow \underset{i,j}{\exists} \ldots$

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

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):

- `init (i x) { T = 0.0 && B[i,x]=I && ... }` $\rightsquigarrow \underset{i,x}{\forall} \ldots$
-
  ```
  transition spout_emit(i j)                    ⇝ ∃ ...
                                                  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 }` $\rightsquigarrow \underset{i}{\exists} \ldots$

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):

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

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):

- `init (i x) { T = 0.0 && B[i,x]=I && ...  }` $\leadsto \underset{i,x}{\forall} \ldots$
- `transition spout_emit(i j)` $\leadsto \underset{i,j}{\exists} \ldots$

  ```
  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 }` $\leadsto \underset{i}{\exists} \ldots$

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):

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

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

$\rightsquigarrow \underset{i,j}{\exists} \ldots$ (for `transition spout_emit(i j)`)

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

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):

- `init (i x) { T = 0.0 && B[i,x]=I && ... }` $\rightsquigarrow \underset{i,x}{\forall} \dots$
- `transition spout_emit(i j)` $\rightsquigarrow \underset{i,j}{\exists} \dots$

  ```
  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 }` $\rightsquigarrow \underset{i}{\exists} \dots$

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 && ...  }` $\rightsquigarrow \underset{i,x}{\forall} \ldots$
- `transition spout_emit(i j)` $\rightsquigarrow \underset{i,j}{\exists} \ldots$

  ```
  transition spout_emit(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 }` $\rightsquigarrow \underset{i}{\exists} \ldots$

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):

- `init (i x) { T = 0.0 && B[i,x]=I && ...  }` $\rightsquigarrow \underset{i,x}{\forall} \dots$
-
  ```
  transition spout_emit(i j)
  ```
  $\rightsquigarrow \underset{i,j}{\exists} \dots$
  ```
  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 }` $\rightsquigarrow \underset{i}{\exists} \dots$

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 && ...  }` $\rightsquigarrow \underset{i,x}{\forall} \dots$
-
  ```
  transition spout_emit(i j)
  requires {Tsmin<Stime[i] && SubscribedBS[j,i]=True && ...}
   {
          L[l]:=case
          | l=j :  L[l]+1.0
          | _ :  L[l];
   }
  ```
  $\rightsquigarrow \underset{i,j}{\exists} \dots$
- `unsafe(i) { L[i]>1.5 }` $\rightsquigarrow \underset{i}{\exists} \dots$

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 && ...  }` $\rightsquigarrow \underset{i,x}{\forall} \ldots$
-

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

$\rightsquigarrow \underset{i,j}{\exists} \ldots$

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

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):

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

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

$\rightsquigarrow \underset{i,j}{\exists} \dots$

- `unsafe(i) { L[i]>1.5 }` $\rightsquigarrow \underset{i}{\exists} \dots$

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 δ time units

$$\underset{\delta}{\exists}\, 0 < \delta \wedge \mathit{CanTimeElapse} = \texttt{true} \wedge$$
$$\underset{j,z}{\forall} \begin{pmatrix} T' & = T + \delta \\ P'[j,z] & = \textbf{if } (0 \leq P[j,z] - \delta) \textbf{then } P[j,z] - \delta \textbf{ else } 0 \\ B'[j,z] & \ldots \\ \mathit{CanTimeElapse}' & = \texttt{false} \end{pmatrix}$$

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/`).

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 δ time units

$$\underset{\delta}{\exists}\, 0 < \delta \wedge \mathit{CanTimeElapse} = \texttt{true} \wedge \underset{j,z}{\forall} \begin{pmatrix} T' & = T + \delta \\ P'[j,z] & = \textbf{if } (0 \leq P[j,z] - \delta) \textbf{then } P[j,z] - \delta \textbf{ else } 0 \\ B'[j,z] & \ldots \\ \mathit{CanTimeElapse}' & = \texttt{false} \end{pmatrix}$$

Examples of transitions and their effect:

- $\mathit{spout}_{\mathit{emit}}(i,j)$: $L[j]$ increases ($\mathit{SubscribedBS}[j,i]$); emit time of the spout (Stime) is reset
- $\mathit{bolt}_{\mathit{emit}}(i,j)$: the state of $B[i]$ is changed into `idle` and $L[j]$ is incremented by 1
- $\mathit{bolt}_{\mathit{take}}(j,y)$: $L[j]$ 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 δ time units

$$\underset{\delta}{\exists}\, 0 < \delta \wedge \mathit{CanTimeElapse} = \texttt{true} \wedge \\ \underset{j,z}{\forall} \begin{pmatrix} T' & = T + \delta \\ P'[j,z] & = \textbf{if } (0 \leq P[j,z] - \delta) \textbf{then } P[j,z] - \delta \textbf{ else } 0 \\ B'[j,z] & \dots \\ \mathit{CanTimeElapse}' & = \texttt{false} \end{pmatrix}$$

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/`).

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 δ time units

$$\exists_{\delta}\, 0 < \delta \wedge \mathit{CanTimeElapse} = \texttt{true} \wedge \\ \forall_{j,z} \begin{pmatrix} T' & = T + \delta \\ P'[j,z] & = \textbf{if } (0 \leq P[j,z] - \delta) \textbf{then } P[j,z] - \delta \textbf{ else } 0 \\ B'[j,z] & \ldots \\ \mathit{CanTimeElapse}' & = \texttt{false} \end{pmatrix}$$

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/`).

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 δ time units

$$\exists_{\delta}\, 0 < \delta \wedge \mathit{CanTimeElapse} = \texttt{true} \wedge \forall_{j,z} \begin{pmatrix} T' & = T + \delta \\ P'[j,z] & = \textbf{if } (0 \leq P[j,z] - \delta) \textbf{then } P[j,z] - \delta \textbf{ else } 0 \\ B'[j,z] & \ldots \\ \mathit{CanTimeElapse}' & = \texttt{false} \end{pmatrix}$$

Examples of transitions and their effect:

- $\mathit{spout}_{\mathit{emit}}(i,j)$: $L[j]$ increases ($\mathit{SubscribedBS}[j,i]$); emit time of the spout (Stime) is reset
- $\mathit{bolt}_{\mathit{emit}}(i,j)$: the state of $B[i]$ is changed into `idle` and $L[j]$ is incremented by 1
- $\mathit{bolt}_{\mathit{take}}(j,y)$: $L[j]$ 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 δ time units

$$\underset{\delta}{\exists}\, 0 < \delta \wedge \mathit{CanTimeElapse} = \texttt{true} \wedge \\ \underset{j,z}{\forall} \begin{pmatrix} T' & = T + \delta \\ P'[j,z] & = \textbf{if } (0 \leq P[j,z] - \delta)\textbf{then } P[j,z] - \delta \textbf{ else } 0 \\ B'[j,z] & \ldots \\ \mathit{CanTimeElapse}' & = \texttt{false} \end{pmatrix}$$

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/`).

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 δ time units

$$\underset{\delta}{\exists}\, 0 < \delta \wedge \mathit{CanTimeElapse} = \mathtt{true} \wedge$$
$$\underset{j,z}{\forall} \begin{pmatrix} T' & = T + \delta \\ P'[j,z] & = \textbf{if } (0 \leq P[j,z] - \delta) \textbf{then } P[j,z] - \delta \textbf{ else } 0 \\ B'[j,z] & \ldots \\ \mathit{CanTimeElapse}' & = \mathtt{false} \end{pmatrix}$$

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/`).

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 δ time units

$$\exists_{\delta} 0 < \delta \wedge \mathit{CanTimeElapse} = \mathtt{true} \wedge$$
$$\forall_{j,z} \begin{pmatrix} T' & = T + \delta \\ P'[j,z] & = \textbf{if } (0 \leq P[j,z] - \delta) \textbf{then } P[j,z] - \delta \textbf{ else } 0 \\ B'[j,z] & \ldots \\ \mathit{CanTimeElapse}' & = \mathtt{false} \end{pmatrix}$$

Examples of transitions and their effect:

- $\mathit{spout}_{\mathit{emit}}(i,j)$: $L[j]$ increases ($\mathit{SubscribedBS}[j,i]$); emit time of the spout (Stime) is reset
- $\mathit{bolt}_{\mathit{emit}}(i,j)$: the state of $B[i]$ is changed into `idle` and $L[j]$ is incremented by 1
- $\mathit{bolt}_{\mathit{take}}(j,y)$: $L[j]$ 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

$$\underset{x,y,i,j}{\exists}\ \mathit{statechange} = \texttt{true}\ \wedge$$
$$\underset{l,k,z}{\forall}\begin{pmatrix} \mathit{statechange}' = & \texttt{false} \\ \ldots & \\ B'[l,z] = & \textbf{if}\ (z = y \wedge l = j \wedge B[j,y] = \texttt{E}) \\ & \textbf{then}\ (\texttt{I}\ \textbf{or}\ \texttt{K})\ \textbf{else}\ B[l,z] \\ & \textbf{elseif}\ \ldots \\ \mathit{CanTimeElapse}' = & \texttt{true} \end{pmatrix}$$

- 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)

$$\underset{i,j,x}{\exists}\ Ts_{min} < \mathit{Stime}[i] \wedge \mathit{SubscribedBS}[j,i] = \texttt{true} \wedge \ldots$$
$$\underset{l}{\forall}\begin{pmatrix} L'[l] & = \textbf{if}\ (l = j)\ \textbf{then}\ L[l]+1\ \textbf{else}\ L[l] \\ \mathit{Stime}'[l] & = \textbf{if}\ (l = i)\ \textbf{then}\ 0\ \textbf{else}\ \mathit{Stime}[l] \\ \ldots & \end{pmatrix}$$

- 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

$$\exists_{x,y,i,j}\ \mathit{statechange} = \mathtt{true}\ \wedge$$
$$\forall_{l,k,z} \begin{pmatrix} \mathit{statechange}' = & \mathtt{false} \\ \ldots & \\ B'[l,z] = & \textbf{if}\ (z = y \wedge l = j \wedge B[j,y] = \mathtt{E}) \\ & \textbf{then}\ (\mathtt{I}\ \textbf{or}\ \mathtt{K})\ \textbf{else}\ B[l,z] \\ & \textbf{elseif}\ \ldots \\ \mathit{CanTimeElapse}' = & \mathtt{true} \end{pmatrix}$$

- 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)

$$\exists_{i,j,x}\ Ts_{min} < Stime[i] \wedge SubscribedBS[j,i] = \mathtt{true} \wedge \ldots$$
$$\forall_{l} \begin{pmatrix} L'[l] & = \textbf{if}\ (l = j)\ \textbf{then}\ L[l]+1\ \textbf{else}\ L[l] \\ Stime'[l] & = \textbf{if}\ (l = i)\ \textbf{then}\ 0\ \textbf{else}\ Stime[l] \\ \ldots & \end{pmatrix}$$

- 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

$$\exists_{x,y,i,j}\, \mathit{statechange} = \mathtt{true} \wedge \forall_{l,k,z} \begin{pmatrix} \mathit{statechange}' = & \mathtt{false} \\ \ldots & \\ B'[l,z] = & \textbf{if } (z = y \wedge l = j \wedge B[j,y] = \mathtt{E}) \\ & \textbf{then } (\mathtt{I} \textbf{ or } \mathtt{K}) \textbf{ else } B[l,z] \\ & \textbf{elseif } \ldots \\ \mathit{CanTimeElapse}' = & \mathtt{true} \end{pmatrix}$$

- 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)

$$\exists_{i,j,x}\, Ts_{min} < \mathit{Stime}[i] \wedge \mathit{SubscribedBS}[j,i] = \mathtt{true} \wedge \ldots \\ \forall_{l} \begin{pmatrix} L'[l] & = \textbf{if } (l = j) \textbf{ then } L[l]+1 \textbf{ else } L[l] \\ \mathit{Stime}'[l] & = \textbf{if } (l = i) \textbf{ then } 0 \textbf{ else } \mathit{Stime}[l] \\ \ldots & \end{pmatrix}$$

- 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

$$\underset{x,y,i,j}{\exists}\, \mathit{statechange} = \texttt{true} \wedge \underset{l,k,z}{\forall} \begin{pmatrix} \mathit{statechange}' = & \texttt{false} \\ \ldots & \\ B'[l,z] = & \textbf{if } (z = y \wedge l = j \wedge B[j,y] = \texttt{E}) \\ & \textbf{then } (\texttt{I} \textbf{ or } \texttt{K}) \textbf{ else } B[l,z] \\ & \textbf{elseif } \ldots \\ \mathit{CanTimeElapse}' = & \texttt{true} \end{pmatrix}$$

- 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)

$$\underset{i,j,x}{\exists}\, Ts_{min} < Stime[i] \wedge SubscribedBS[j,i] = \texttt{true} \wedge \ldots$$

$$\underset{l}{\forall} \begin{pmatrix} L'[l] & = \textbf{if } (l = j) \textbf{ then } L[l]+1 \textbf{ else } L[l] \\ Stime'[l] & = \textbf{if } (l = i) \textbf{ then } 0 \textbf{ else } Stime[l] \\ \ldots & \end{pmatrix}$$

- 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

$$\exists_{x,y,i,j}\, \mathit{statechange} = \mathtt{true} \wedge$$
$$\forall_{l,k,z} \begin{pmatrix} \mathit{statechange}' = & \mathtt{false} \\ \ldots & \\ B'[l,z] = & \textbf{if } (z = y \wedge l = j \wedge B[j,y] = \mathtt{E}) \\ & \textbf{then } (\mathtt{I} \textbf{ or } \mathtt{K}) \textbf{ else } B[l,z] \\ & \textbf{elseif } \ldots \\ \mathit{CanTimeElapse}' = & \mathtt{true} \end{pmatrix}$$

- 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)

$$\exists_{i,j,x}\, \mathit{Ts}_{min} < \mathit{Stime}[i] \wedge \mathit{SubscribedBS}[j,i] = \mathtt{true} \wedge \ldots$$
$$\forall_{l} \begin{pmatrix} L'[l] \quad = \textbf{if } (l = j) \textbf{ then } L[l] + 1 \textbf{ else } L[l] \\ \mathit{Stime}'[l] = \textbf{if } (l = i) \textbf{ then } 0 \textbf{ else } \mathit{Stime}[l] \\ \ldots \end{pmatrix}$$

- 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

$$\underset{x,y,i,j}{\exists}\ statechange = \mathtt{true} \wedge \underset{l,k,z}{\forall} \begin{pmatrix} statechange' = & \mathtt{false} \\ \ldots & \\ B'[l,z] = & \textbf{if } (z=y \wedge l=j \wedge B[j,y]=\mathtt{E}) \\ & \textbf{then } (\mathtt{I} \textbf{ or } \mathtt{K}) \textbf{ else } B[l,z] \\ & \textbf{elseif } \ldots \\ CanTimeElapse' = & \mathtt{true} \end{pmatrix}$$

- 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)

$$\underset{i,j,x}{\exists}\ Ts_{min} < Stime[i] \wedge SubscribedBS[j,i] = \mathtt{true} \wedge \ldots \underset{l}{\forall} \begin{pmatrix} L'[l] & = \textbf{if } (l=j) \textbf{ then } L[l]+1 \textbf{ else } L[l] \\ Stime'[l] & = \textbf{if } (l=i) \textbf{ then } 0 \textbf{ else } Stime[l] \\ \ldots & \end{pmatrix}$$

- 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

$$\exists_{x,y,i,j}\ \mathit{statechange} = \texttt{true} \wedge \forall_{l,k,z} \begin{pmatrix} \mathit{statechange}' = & \texttt{false} \\ \ldots & \\ B'[l,z] = & \textbf{if }\ (z = y \wedge l = j \wedge B[j,y] = \texttt{E}) \\ & \textbf{then }\ (\texttt{I}\ \textbf{or}\ \texttt{K})\ \textbf{ else }\ B[l,z] \\ & \textbf{elseif }\ \ldots \\ \mathit{CanTimeElapse}' = & \texttt{true} \end{pmatrix}$$

- 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)

$$\exists_{i,j,x}\ \mathit{Ts}_{min} < \mathit{Stime}[i] \wedge \mathit{SubscribedBS}[j,i] = \texttt{true} \wedge \ldots \forall_{l} \begin{pmatrix} L'[l] & = \textbf{if }\ (l = j)\ \textbf{ then }\ L[l] + 1\ \textbf{ else }\ L[l] \\ \mathit{Stime}'[l] & = \textbf{if }\ (l = i)\ \textbf{ then }\ 0\ \textbf{ else }\ \mathit{Stime}[l] \\ \ldots & \end{pmatrix}$$

- 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

$$\underset{x,y,i,j}{\exists} \mathit{statechange} = \mathtt{true} \wedge$$
$$\underset{l,k,z}{\forall} \begin{pmatrix} \mathit{statechange}' = & \mathtt{false} \\ \ldots & \\ B'[l,z] = & \textbf{if } (z=y \wedge l=j \wedge B[j,y]=\mathtt{E}) \\ & \textbf{then } (\mathtt{I} \textbf{ or } \mathtt{K}) \textbf{ else } B[l,z] \\ & \textbf{elseif } \ldots \\ \mathit{CanTimeElapse}' = & \mathtt{true} \end{pmatrix}$$

- 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)

$$\underset{i,j,x}{\exists} Ts_{min} < Stime[i] \wedge SubscribedBS[j,i] = \mathtt{true} \wedge \ldots$$
$$\underset{l}{\forall} \begin{pmatrix} L'[l] = \textbf{if } (l=j) \textbf{ then } L[l]+1 \textbf{ else } L[l] \\ Stime'[l] = \textbf{if } (l=i) \textbf{ then } 0 \textbf{ else } Stime[l] \\ \ldots \end{pmatrix}$$

- 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

$$\exists_{x,y,i,j}\ statechange = \texttt{true} \land$$
$$\forall_{l,k,z} \begin{pmatrix} statechange' = & \texttt{false} \\ \dots & \\ B'[l,z] = & \textbf{if}\ (z=y \land l=j \land B[j,y]=\texttt{E}) \\ & \textbf{then}\ (\texttt{I}\ \textbf{or}\ \texttt{K})\ \textbf{else}\ B[l,z] \\ & \textbf{elseif}\ \dots \\ CanTimeElapse' = & \texttt{true} \end{pmatrix}$$

- 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)

$$\exists_{i,j,x}\ Ts_{min} < Stime[i] \land SubscribedBS[j,i] = \texttt{true} \land \dots$$
$$\forall_{l} \begin{pmatrix} L'[l] & = \textbf{if}\ (l=j)\ \textbf{then}\ L[l]+1\ \textbf{else}\ L[l] \\ Stime'[l] & = \textbf{if}\ (l=i)\ \textbf{then}\ 0\ \textbf{else}\ Stime[l] \\ \dots & \end{pmatrix}$$

- 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

$$\underset{x,y,i,j}{\exists}\ statechange = \texttt{true} \wedge$$
$$\underset{l,k,z}{\forall} \begin{pmatrix} statechange' = & \texttt{false} \\ \ldots & \\ B'[l,z] = & \textbf{if}\ (z = y \wedge l = j \wedge B[j,y] = \texttt{E}) \\ & \textbf{then}\ (\texttt{I}\ \textbf{or}\ \texttt{K})\ \textbf{else}\ B[l,z] \\ & \textbf{elseif}\ \ldots \\ CanTimeElapse' = & \texttt{true} \end{pmatrix}$$

- 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)

$$\underset{i,j,x}{\exists}\ Ts_{min} < Stime[i] \wedge SubscribedBS[j,i] = \texttt{true} \wedge \ldots$$
$$\underset{l}{\forall} \begin{pmatrix} L'[l] & = \textbf{if}\ (l = j)\ \textbf{then}\ L[l]+1\ \textbf{else}\ L[l] \\ Stime'[l] & = \textbf{if}\ (l = i)\ \textbf{then}\ 0\ \textbf{else}\ Stime[l] \\ \ldots & \end{pmatrix}$$

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

- $L[1] \geq 3$ 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: $Init \rightarrow time\_elapse \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \mathbf{spout_{emit}} \rightarrow$
  $time\_elapse \rightarrow setDoTake_{\texttt{True}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow$
  $\mathbf{bolt1_{take}} \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \mathbf{spout_{emit}} \rightarrow$
  $time\_elapse \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \mathbf{spout_{emit}} \rightarrow L[1] \geq 2$

- $L[1] \geq 3$ 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: $Init \rightarrow time\_elapse \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \textbf{spout}_{\textbf{emit}} \rightarrow$
  $time\_elapse \rightarrow setDoTake_{\texttt{True}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow$
  $\textbf{bolt1}_{\textbf{take}} \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \textbf{spout}_{\textbf{emit}} \rightarrow$
  $time\_elapse \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \textbf{spout}_{\textbf{emit}} \rightarrow L[1] \geq 2$

- $L[1] \geq 3$ 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: $Init \rightarrow time\_elapse \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \textbf{spout}_{\textbf{emit}} \rightarrow time\_elapse \rightarrow setDoTake_{\texttt{True}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \textbf{bolt1}_{\textbf{take}} \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \textbf{spout}_{\textbf{emit}} \rightarrow time\_elapse \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \textbf{spout}_{\textbf{emit}} \rightarrow L[1] \geq 2$

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

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

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

Result: the verification problems lead to memory exhaustion.

Second attempt:

- $L[1] \geq 2$ and $\mathit{Tsmin} < 1$ – obtained result: UNSAFE
- $L[1] \geq 2$ and $\mathit{Tsmin} \geq 1$ – expected result: SAFE

Experimental results

First attempt:

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

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

- $L[1] \geq 3$ 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: $Init \rightarrow time\_elapse \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \mathbf{spout_{emit}} \rightarrow$
  $time\_elapse \rightarrow setDoTake_{\texttt{True}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow$
  $\mathbf{bolt1_{take}} \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \mathbf{spout_{emit}} \rightarrow$
  $time\_elapse \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \mathbf{spout_{emit}} \rightarrow L[1] \geq 2$

- $L[1] \geq 3$ 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: $Init \rightarrow time\_elapse \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \textbf{spout}_{\textbf{emit}} \rightarrow time\_elapse \rightarrow setDoTake_{\texttt{True}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \textbf{bolt1}_{\textbf{take}} \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \textbf{spout}_{\textbf{emit}} \rightarrow time\_elapse \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \textbf{spout}_{\textbf{emit}} \rightarrow L[1] \geq 2$

- $L[1] \geq 3$ 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 $\rightsquigarrow$ 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 $\rightsquigarrow$ 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