Kernel: functional notation for system description

Wojciech Fraczak

Dépt d'informatique, Université du Québec en Outaouais, Gatineau PQ, Canada

Introduction

- *Kernel* is a notation for describing computer systems
- *Kernel* is both a programing language which can be compiled into a highly optimized machine code, and a specification language for modeling and verification purposes
- Main features unique to *Kernel* are:
  - the uniform and minimalistic syntax;
  - precise mathematical semantics in terms of relations;
  - execution model (virtual machine) in terms of FIFO system.

Introduction

- *Kernel* is a notation for describing computer systems
- *Kernel* is both a programing language which can be compiled into a highly optimized machine code, and a specification language for modeling and verification purposes
- Main features unique to *Kernel* are:
  - the uniform and minimalistic syntax;
  - precise mathematical semantics in terms of relations;
  - execution model (virtual machine) in terms of FIFO system.

Introduction

- *Kernel* is a notation for describing computer systems
- *Kernel* is both a programing language which can be compiled into a highly optimized machine code, and a specification language for modeling and verification purposes
- Main features unique to *Kernel* are:
  - the uniform and minimalistic syntax;
  - precise mathematical semantics in terms of relations;
  - execution model (virtual machine) in terms of FIFO system.

History and motivation

1. My Solidum/IDT Canada experience: PDL/PTL (Packet Description/Transformation Language)
   - for programming one-state push-down automaton implemented in hardware
2. Edgewater: RTEdge — a real-time programming language with integrated verifier for "meeting deadlines"

- In both cases the proprietary languages were needed even though hundreds of programming languages had already existed...
- *Kernel* aims to become a common notation easily extensible ... to meet PTL or RTEdge requirements, for example.

History and motivation

1. My Solidum/IDT Canada experience: PDL/PTL (Packet Description/Transformation Language)
   - for programming one-state push-down automaton implemented in hardware
2. Edgewater: RTEdge — a real-time programming language with integrated verifier for “meeting deadlines”

- In both cases the proprietary languages were needed even though hundreds of programming languages had already existed...
- *Kernel* aims to become a common notation easily extensible ... to meet PTL or RTEdge requirements, for example.

History and motivation

1. My Solidum/IDT Canada experience: PDL/PTL (Packet Description/Transformation Language)
   - for programming one-state push-down automaton implemented in hardware
2. Edgewater: RTEdge — a real-time programming language with integrated verifier for "meeting deadlines"

- In both cases the proprietary languages were needed even though hundreds of programming languages had already existed...
- *Kernel* aims to become a common notation easily extensible ... to meet PTL or RTEdge requirements, for example.

History and motivation

1. My Solidum/IDT Canada experience: PDL/PTL (Packet Description/Transformation Language)
   - for programming one-state push-down automaton implemented in hardware
2. Edgewater: RTEdge — a real-time programming language with integrated verifier for "meeting deadlines"

- In both cases the proprietary languages were needed even though hundreds of programming languages had already existed...
- *Kernel* aims to become a common notation easily extensible ... to meet PTL or RTEdge requirements, for example.

What *Kernel* is not...

- *Kernel* is not a full featured general purpose programming language for IT professionals ...

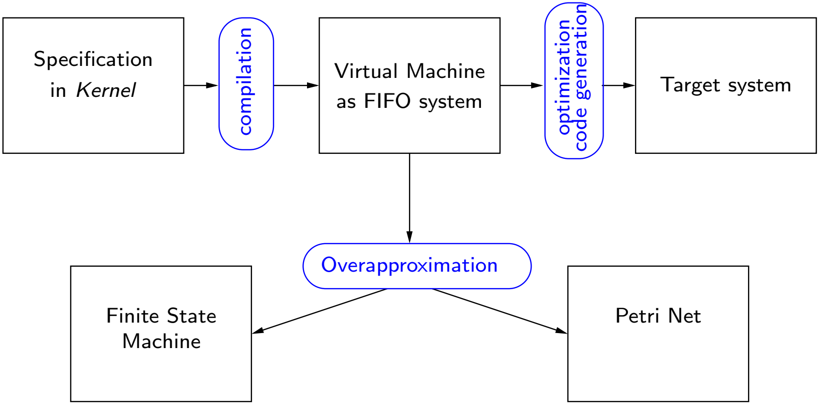
  Even though one could use *Kernel* to develop a Web site or a flashy graphical user interface, the notation was not designed for it ...
- You may be interested in *Kernel* if you want your program to be **provably**:
  - fast,
  - resource efficient, and/or
  - correct.

What *Kernel* is not...

- *Kernel* is not a full featured general purpose programming language for IT professionals ...

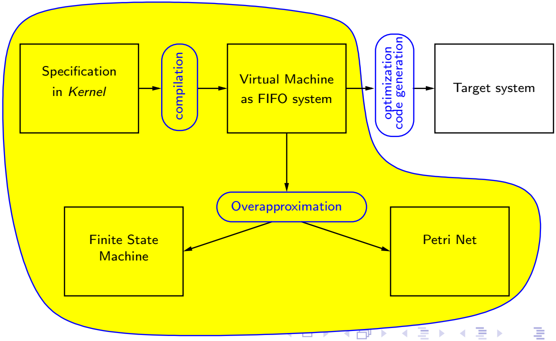
  Even though one could use *Kernel* to develop a Web site or a flashy graphical user interface, the notation was not designed for it ...
- You may be interested in *Kernel* if you want your program to be **provably**:
  - fast,
  - resource efficient, and/or
  - correct.

Overview of the approach



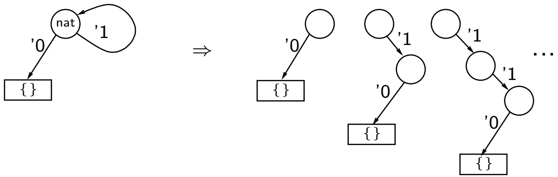
Outline of the talk

1. Short introduction to *Kernel* notation
2. How do we compile a *Kernel* program into FIFO system
3. Verification by *over-approximations*



Flat (printable) types, values, and patterns

```
<type> nat <is> ['0 {}, '1 nat ];
```



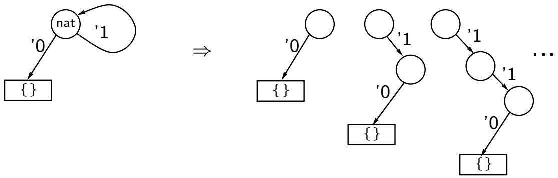
Values: '0, '1'0, '1'1'0, '1'1'1'0, ..., are *trees*!

A *Kernel* type is:

- a set of values (finite tree automata = and-or graphs),
- a set of constructors (field names), and
- a set of *patterns* (initial parts of values).

Flat (printable) types, values, and patterns

```
<type> nat <is> ['0 {}, '1 nat ];
```



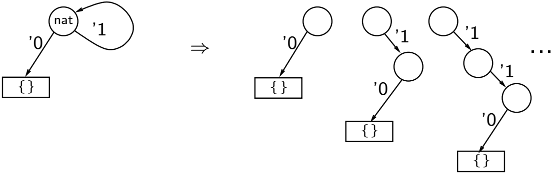
Values: `'0`, `'1'0`, `'1'1'0`, `'1'1'1'0`, ..., are *trees*!

A *Kernel* type is:

- a set of values (finite tree automata = and-or graphs),
- a set of constructors (field names), and
- a set of *patterns* (initial parts of values).

Flat (printable) types, values, and patterns

```
<type> nat <is> ['0 {}, '1 nat ];
```



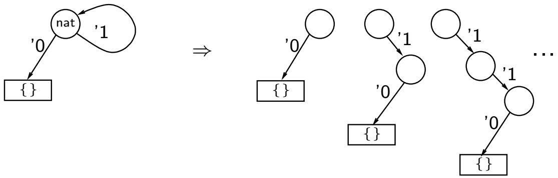
Values: `'0`, `'1'0`, `'1'1'0`, `'1'1'1'0`, ..., are *trees*!

A *Kernel* type is:

- a set of values (finite tree automata = and-or graphs),
- a set of constructors (field names), and
- a set of *patterns* (initial parts of values).

Flat (printable) types, values, and patterns

```
<type> nat <is> ['0 {}, '1 nat ];
```

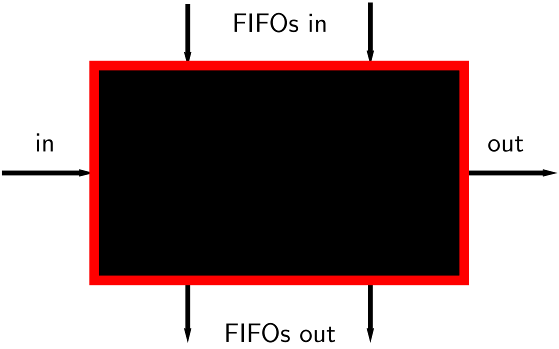


Values: `'0`, `'1'0`, `'1'1'0`, `'1'1'1'0`, ..., are *trees*!

A *Kernel* type is:

- a set of values (finite tree automata = and-or graphs),
- a set of constructors (field names), and
- a set of *patterns* (initial parts of values).

Kernel components are *relations*



Every non-recursive *Kernel* component can be translated into one (possibly very big) state machine.

Kernel syntax by example

```
<type> nat <is> ['0, '1 nat ];        -- (recursive) variant type
<type> pair <is> {'x nat, 'y nat};    -- record type

(pair -> pair) swap = (pair -> {'x $.y, 'y $.x});

({} -> nat) get_nat;                  -- FIFO

({} -> pair) get_pair = {'x get_nat, 'y get_nat} <by> CONCAT;

(pair -> nat) x-y =
   [ ({'x ..., 'y '0 } -> $.x),
     ({'x '1 ..., 'y ... } -> {'x $.x.1, 'y $.y.1}) :: x-y
   ] <by> FIRST_MATCH;

(pair -> nat) y-x = swap :: x-y;

({} -> nat) main =
 [ {'y get_nat, 'x '1 @y} :: x-y,
   get_nat,
   get_pair :: x-y
   <longest> /* <shortest>, <unknown> */ get_nat
 ];
```

Kernel syntax by example

```
<type> nat <is> ['0, '1 nat ];        -- (recursive) variant type
<type> pair <is> {'x nat, 'y nat};    -- record type

(pair -> pair) swap = (pair -> {'x $.y, 'y $.x});

({} -> nat) get_nat;                  -- FIFO

({} -> pair) get_pair = {'x get_nat, 'y get_nat} <by> CONCAT;

(pair -> nat) x-y =
   [ ({'x ..., 'y '0 } -> $.x),
     ({'x '1 ..., 'y ... } -> {'x $.x.1, 'y $.y.1}) :: x-y
   ] <by> FIRST_MATCH;

(pair -> nat) y-x = swap :: x-y;

({} -> nat) main =
 [ {'y get_nat, 'x '1 @y} :: x-y,
   get_nat,
   get_pair :: x-y
   <longest> /* <shortest>, <unknown> */ get_nat
 ];
```

Kernel syntax by example

```
<type> nat <is> ['0, '1 nat ];         -- (recursive) variant type
<type> pair <is> {'x nat, 'y nat};     -- record type

(pair -> pair) swap = (pair -> {'x $.y, 'y $.x});

({} -> nat) get_nat;                   -- FIFO

({} -> pair) get_pair = {'x get_nat, 'y get_nat} <by> CONCAT;

(pair -> nat) x-y =
   [ ({'x ..., 'y '0 } -> $.x),
     ({'x '1 ..., 'y ... } -> {'x $.x.1, 'y $.y.1}) :: x-y
   ] <by> FIRST_MATCH;

(pair -> nat) y-x = swap :: x-y;

({} -> nat) main =
 [ {'y get_nat, 'x '1 @y} :: x-y,
   get_nat,
   get_pair :: x-y
   <longest> /* <shortest>, <unknown> */ get_nat
 ];
```

Kernel syntax by example

```
<type> nat <is> ['0, '1 nat ];         -- (recursive) variant type
<type> pair <is> {'x nat, 'y nat};   -- record type

(pair -> pair) swap = (pair -> {'x $.y, 'y $.x});

({} -> nat) get_nat;                   -- FIFO

({} -> pair) get_pair = {'x get_nat, 'y get_nat} <by> CONCAT;

(pair -> nat) x-y =
   [ ({'x ..., 'y '0 } -> $.x),
     ({'x '1 ..., 'y ... } -> {'x $.x.1, 'y $.y.1}) :: x-y
   ] <by> FIRST_MATCH;

(pair -> nat) y-x = swap :: x-y;

({} -> nat) main =
 [ {'y get_nat, 'x '1 @y} :: x-y,
   get_nat,
   get_pair :: x-y
   <longest> /* <shortest>, <unknown> */ get_nat
 ];
```

Kernel syntax by example

```
<type> nat <is> [’0, ’1 nat ];        -- (recursive) variant type
<type> pair <is> {’x nat, ’y nat};    -- record type

(pair -> pair) swap = (pair -> {’x $.y, ’y $.x});

({} -> nat) get_nat;                  -- FIFO

({} -> pair) get_pair = {’x get_nat, ’y get_nat} <by> CONCAT;

(pair -> nat) x-y =
   [ ({’x ..., ’y ’0 } -> $.x),
     ({’x ’1 ..., ’y ... } -> {’x $.x.1, ’y $.y.1}) :: x-y
   ] <by> FIRST_MATCH;

(pair -> nat) y-x = swap :: x-y;

({} -> nat) main =
 [ {’y get_nat, ’x ’1 @y} :: x-y,
   get_nat,
   get_pair :: x-y
   <longest> /* <shortest>, <unknown> */ get_nat
 ];
```

Kernel syntax by example

```
<type> nat <is> ['0, '1 nat ];          -- (recursive) variant type
<type> pair <is> {'x nat, 'y nat};     -- record type

(pair -> pair) swap = (pair -> {'x $.y, 'y $.x});

({} -> nat) get_nat;                   -- FIFO

({} -> pair) get_pair = {'x get_nat, 'y get_nat} <by> CONCAT;

(pair -> nat) x-y =
   [ ({'x ..., 'y '0 } -> $.x),
     ({'x '1 ..., 'y ... } -> {'x $.x.1, 'y $.y.1}) :: x-y
   ] <by> FIRST_MATCH;

(pair -> nat) y-x = swap :: x-y;

({} -> nat) main =
 [ {'y get_nat, 'x '1 @y} :: x-y,
   get_nat,
   get_pair :: x-y
   <longest> /* <shortest>, <unknown> */ get_nat
 ];
```

Kernel syntax by example

```
<type> nat <is> ['0, '1 nat ];         -- (recursive) variant type
<type> pair <is> {'x nat, 'y nat};    -- record type

(pair -> pair) swap = (pair -> {'x $.y, 'y $.x});

({} -> nat) get_nat;                   -- FIFO

({} -> pair) get_pair = {'x get_nat, 'y get_nat} <by> CONCAT;

(pair -> nat) x-y =
   [ ({'x ..., 'y '0 } -> $.x),
     ({'x '1 ..., 'y ... } -> {'x $.x.1, 'y $.y.1}) :: x-y
   ] <by> FIRST_MATCH;

(pair -> nat) y-x = swap :: x-y;

({} -> nat) main =
 [ {'y get_nat, 'x '1 @y} :: x-y,
   get_nat,
   get_pair :: x-y
   <longest> /* <shortest>, <unknown> */ get_nat
 ];
```

Kernel syntax by example

```
<type> nat <is> ['0, '1 nat ];        -- (recursive) variant type
<type> pair <is> {'x nat, 'y nat};    -- record type

(pair -> pair) swap = (pair -> {'x $.y, 'y $.x});

({} -> nat) get_nat;                   -- FIFO

({} -> pair) get_pair = {'x get_nat, 'y get_nat} <by> CONCAT;

(pair -> nat) x-y =
   [ ({'x ..., 'y '0 } -> $.x),
     ({'x '1 ..., 'y ... } -> {'x $.x.1, 'y $.y.1}) :: x-y
   ] <by> FIRST_MATCH;

(pair -> nat) y-x = swap :: x-y;

({} -> nat) main =
 [ {'y get_nat, 'x '1 @y} :: x-y,
   get_nat,
   get_pair :: x-y
   <longest> /* <shortest>, <unknown> */ get_nat
 ];
```

Kernel syntax by example

```
<type> nat <is> ['0, '1 nat ];        -- (recursive) variant type
<type> pair <is> {'x nat, 'y nat};    -- record type

(pair -> pair) swap = (pair -> {'x $.y, 'y $.x});

({} -> nat) get_nat;                  -- FIFO

({} -> pair) get_pair = {'x get_nat, 'y get_nat} <by> CONCAT;

(pair -> nat) x-y =
   [ ({'x ..., 'y '0 } -> $.x),
     ({'x '1 ..., 'y ... } -> {'x $.x.1, 'y $.y.1}) :: x-y
   ] <by> FIRST_MATCH;

(pair -> nat) y-x = swap :: x-y;

({} -> nat) main =
 [ {'y get_nat, 'x '1 @y} :: x-y,
   get_nat,
   get_pair :: x-y
   <longest> /* <shortest>, <unknown> */ get_nat
 ];
```

Building state machines

Every construction in *Kernel* is an operation of *relations*.

- **Pattern matching**
  - Every pattern corresponds to a regular tree language encoded by a "simple language" (one state push-down automaton).
  - Regular tree languages are closed by union, intersection, and complement.
- **Product**, **Union**, and **Composition** are variations of composition of relations - realized by corresponding state products
- **Projection** is relabeling
- ...

Building state machines

Every construction in *Kernel* is an operation of *relations*.

- **Pattern matching**
  - Every pattern corresponds to a regular tree language encoded by a "simple language" (one state push-down automaton).
  - Regular tree languages are closed by union, intersection, and complement.
- **Product**, **Union**, and **Composition** are variations of composition of relations - realized by corresponding state products
- **Projection** is relabeling
- ...

Building state machines

Every construction in *Kernel* is an operation of *relations*.

- **Pattern matching**
  - Every pattern corresponds to a regular tree language encoded by a "simple language" (one state push-down automaton).
  - Regular tree languages are closed by union, intersection, and complement.
- **Product**, **Union**, and **Composition** are variations of composition of relations - realized by corresponding state products
- **Projection** is relabeling
- ...

Building state machines

Every construction in *Kernel* is an operation of *relations*.

- **Pattern matching**
  - Every pattern corresponds to a regular tree language encoded by a "simple language" (one state push-down automaton).
  - Regular tree languages are closed by union, intersection, and complement.
- **Product**, **Union**, and **Composition** are variations of composition of relations - realized by corresponding state products
- **Projection** is relabeling
- ...

Building state machines

Every construction in *Kernel* is an operation of *relations*.

- **Pattern matching**
  - Every pattern corresponds to a regular tree language encoded by a "simple language" (one state push-down automaton).
  - Regular tree languages are closed by union, intersection, and complement.
- **Product**, **Union**, and **Composition** are variations of composition of relations - realized by corresponding state products
- **Projection** is relabeling
- ...

An example...

- Pattern `{'x ..., 'y '0}` of type `pair`:

$$(x/x)(1/1)^*(0/0)(y/y)(0/0)$$

- Projection `$.x` over argument of type `pair`:

$$(x/)(1/1)^*(0/0)(y/)(1/)^*(0/)$$

- Abstraction:

$$\underbrace{\Big(\underbrace{\texttt{\{'x ..., 'y '0\}}}_{(x/x)(1/1)^*(0/0)(y/y)(0/0)} \rightarrow \underbrace{\texttt{\$.x}}_{(x/)(1/1)^*(0/0)(y/)(1/)^*(0/)} \Big)}_{(x/)(1/1)^*(0/0)(y/)(0/)}$$

An example...

- Pattern `{'x ..., 'y '0}` of type `pair`:

$$(x/x)(1/1)^*(0/0)(y/y)(0/0)$$

- Projection `$.x` over argument of type `pair`:

$$(x/)(1/1)^*(0/0)(y/)(1/)^*(0/)$$

- Abstraction:

$$\underbrace{\left(\underbrace{\texttt{\{'x ..., 'y '0\}}}_{(x/x)(1/1)^*(0/0)(y/y)(0/0)} \rightarrow \underbrace{\texttt{\$.x}}_{(x/)(1/1)^*(0/0)(y/)(1/)^*(0/)} \right)}_{(x/)(1/1)^*(0/0)(y/)(0/)}$$

An example...

- Pattern `{'x ..., 'y '0}` of type `pair`:

$$(x/x)(1/1)^*(0/0)(y/y)(0/0)$$

- Projection `$.x` over argument of type `pair`:

$$(x/)(1/1)^*(0/0)(y/)(1/)^*(0/)$$

- Abstraction:

$$\underbrace{(\ \underbrace{\{\texttt{'x}\ \texttt{...},\ \texttt{'y}\ \texttt{'0}\}}_{(x/x)(1/1)^*(0/0)(y/y)(0/0)} \rightarrow \underbrace{\texttt{\$.x}}_{(x/)(1/1)^*(0/0)(y/)(1/)^*(0/)}\ )}_{(x/)(1/1)^*(0/0)(y/)(0/)}$$

FIFO system

- A FIFO system is a network of finite state machines connected by point-to-point FIFO channels.
- Since FIFOs are unbounded, even a single state machine with a couple of FIFOs is Turing Machine equivalent.
- Advantages of FIFO virtual machine representation are:
  - simple and easy to implement
  - admits distributed implementation
  - friendly to formal verification via over-approximation

FIFO system

- A FIFO system is a network of finite state machines connected by point-to-point FIFO channels.
- Since FIFOs are unbounded, even a single state machine with a couple of FIFOs is Turing Machine equivalent.
- Advantages of FIFO virtual machine representation are:
  - simple and easy to implement
  - admits distributed implementation
  - friendly to formal verification via over-approximation

FIFO system

- A FIFO system is a network of finite state machines connected by point-to-point FIFO channels.
- Since FIFOs are unbounded, even a single state machine with a couple of FIFOs is Turing Machine equivalent.
- Advantages of FIFO virtual machine representation are:
  - simple and easy to implement
  - admits distributed implementation
  - friendly to formal verification via over-approximation

FIFO system

- A FIFO system is a network of finite state machines connected by point-to-point FIFO channels.
- Since FIFOs are unbounded, even a single state machine with a couple of FIFOs is Turing Machine equivalent.
- Advantages of FIFO virtual machine representation are:
  - simple and easy to implement
  - admits distributed implementation
  - friendly to formal verification via over-approximation

FIFO system

- A FIFO system is a network of finite state machines connected by point-to-point FIFO channels.
- Since FIFOs are unbounded, even a single state machine with a couple of FIFOs is Turing Machine equivalent.
- Advantages of FIFO virtual machine representation are:
  - simple and easy to implement
  - admits distributed implementation
  - friendly to formal verification via over-approximation

Connecting state machines into FIFO system

- Finiteness of state representation of relations is not always preserved, e.g., because of recursion.
- We overcome the state explosion problem by constructing a FIFO system instead of a single transducer.
- The challenge is in constructing a “useful” FIFO system

many different techniques may be considered to build a FIFO system from a *Kernel* specification... this remains our main research topic...

Connecting state machines into FIFO system

- Finiteness of state representation of relations is not always preserved, e.g., because of recursion.
- We overcome the state explosion problem by constructing a FIFO system instead of a single transducer.
- The challenge is in constructing a "useful" FIFO system

many different techniques may be considered to build a FIFO system from a *Kernel* specification... this remains our main research topic...

Connecting state machines into FIFO system

- Finiteness of state representation of relations is not always preserved, e.g., because of recursion.
- We overcome the state explosion problem by constructing a FIFO system instead of a single transducer.
- The challenge is in constructing a “useful” FIFO system

many different techniques may be considered to build a FIFO system from a *Kernel* specification... this remains our main research topic...

Connecting state machines into FIFO system

- Finiteness of state representation of relations is not always preserved, e.g., because of recursion.
- We overcome the state explosion problem by constructing a FIFO system instead of a single transducer.
- The challenge is in constructing a "useful" FIFO system

many different techniques may be considered to build a FIFO system from a *Kernel* specification... this remains our main research topic...

Over-approximation

- In theory, a finite state FIFO system (i.e., with bounded size channels) can be verified using SPIN/Promela tool set ... — however, checking if FIFO system is "finite state" is undecidable!
- Over-approximation:
  - over-approximation of S consists in adding "executions" to the system creating its "over-approximation" S', hopefully simpler to analyze
  - to prove that every execution of S has a property ϕ, we prove that every execution of S' has the property ϕ
  - E.g., the over-approximation can be used in proving that a WCRT is bounded by a "deadline"

Over-approximation

- In theory, a finite state FIFO system (i.e., with bounded size channels) can be verified using SPIN/Promela tool set ... — however, checking if FIFO system is “finite state” is undecidable!
- Over-approximation:
  - over-approximation of S consists in adding “executions” to the system creating its “over-approximation” S', hopefully simpler to analyze
  - to prove that every execution of S has a property ϕ, we prove that every execution of S' has the property ϕ
  - E.g., the over-approximation can be used in proving that a WCRT is bounded by a “deadline”

Over-approximation

- In theory, a finite state FIFO system (i.e., with bounded size channels) can be verified using SPIN/Promela tool set ... — however, checking if FIFO system is "finite state" is undecidable!
- Over-approximation:
  - over-approximation of S consists in adding "executions" to the system creating its "over-approximation" S', hopefully simpler to analyze
  - to prove that every execution of S has a property ϕ, we prove that every execution of S' has the property ϕ
  - E.g., the over-approximation can be used in proving that a WCRT is bounded by a "deadline"

Over-approximation

- In theory, a finite state FIFO system (i.e., with bounded size channels) can be verified using SPIN/Promela tool set ... — however, checking if FIFO system is "finite state" is undecidable!
- Over-approximation:
  - over-approximation of S consists in adding "executions" to the system creating its "over-approximation" S', hopefully simpler to analyze
  - to prove that every execution of S has a property ϕ, we prove that every execution of S' has the property ϕ
  - E.g., the over-approximation can be used in proving that a WCRT is bounded by a "deadline"

Over-approximation

- In theory, a finite state FIFO system (i.e., with bounded size channels) can be verified using SPIN/Promela tool set ... — however, checking if FIFO system is "finite state" is undecidable!
- Over-approximation:
  - over-approximation of S consists in adding "executions" to the system creating its "over-approximation" S', hopefully simpler to analyze
  - to prove that every execution of S has a property ϕ, we prove that every execution of S' has the property ϕ
  - E.g., the over-approximation can be used in proving that a WCRT is bounded by a "deadline"

Over-approximation techniques

- We have considered two over-approximations of a FIFO system S:
  - Transformation of S into a finite state machine:
    It can be done by a successive elimination of FIFO channels.
    Question: How a FIFO channel can be eliminated?
  - Transformation of S into a Petri Net.
    If the FIFO channels are replaced by "any-order" containers then without loosing any possible execution of the initial FIFO system, we can replace every channel by a finite set of Places.

Over-approximation techniques

- We have considered two over-approximations of a FIFO system S:
  - Transformation of S into a finite state machine:
    It can be done by a successive elimination of FIFO channels.
    Question: How a FIFO channel can be eliminated?
  - Transformation of S into a Petri Net.
    If the FIFO channels are replaced by "any-order" containers then without loosing any possible execution of the initial FIFO system, we can replace every channel by a finite set of Places.

Over-approximation techniques

- We have considered two over-approximations of a FIFO system S:
  - Transformation of S into a finite state machine:
    It can be done by a successive elimination of FIFO channels.
    Question: How a FIFO channel can be eliminated?
  - Transformation of S into a Petri Net.
    If the FIFO channels are replaced by "any-order" containers then without loosing any possible execution of the initial FIFO system, we can replace every channel by a finite set of Places.

Over-approximation techniques

- We have considered two over-approximations of a FIFO system S:
  - Transformation of S into a finite state machine:
    It can be done by a successive elimination of FIFO channels.
    Question: How a FIFO channel can be eliminated?
  - Transformation of S into a Petri Net.
    If the FIFO channels are replaced by "any-order" containers then without loosing any possible execution of the initial FIFO system, we can replace every channel by a finite set of Places.

Over-approximation techniques

- We have considered two over-approximations of a FIFO system S:
  - Transformation of S into a finite state machine:
    It can be done by a successive elimination of FIFO channels.
    Question: How a FIFO channel can be eliminated?
  - Transformation of S into a Petri Net.
    If the FIFO channels are replaced by "any-order" containers then without loosing any possible execution of the initial FIFO system, we can replace every channel by a finite set of Places.

Over-approximation techniques

- We have considered two over-approximations of a FIFO system S:
  - Transformation of S into a finite state machine:
    It can be done by a successive elimination of FIFO channels.
    Question: How a FIFO channel can be eliminated?
  - Transformation of S into a Petri Net.
    If the FIFO channels are replaced by "any-order" containers then without loosing any possible execution of the initial FIFO system, we can replace every channel by a finite set of Places.

Conclusions

- We propose a simple but powerful notation for describing computing systems
- Semantics of the notation is expressed in terms of multi-dimensional "relations" which can be effectively implemented as a FIFO system
- We mentioned two techniques which can transform FIFO system by over-approximation into:
  - a finite state machine
  - a Petri-Net
- This is work in progress ... especially meaningful FIFO system construction

Conclusions

- We propose a simple but powerful notation for describing computing systems
- Semantics of the notation is expressed in terms of multi-dimensional “relations” which can be effectively implemented as a FIFO system
- We mentioned two techniques which can transform FIFO system by over-approximation into:
  - a finite state machine
  - a Petri-Net
- This is work in progress ... especially meaningful FIFO system construction

Conclusions

- We propose a simple but powerful notation for describing computing systems
- Semantics of the notation is expressed in terms of multi-dimensional "relations" which can be effectively implemented as a FIFO system
- We mentioned two techniques which can transform FIFO system by over-approximation into:
  - a finite state machine
  - a Petri-Net
- This is work in progress … especially meaningful FIFO system construction

Conclusions

- We propose a simple but powerful notation for describing computing systems
- Semantics of the notation is expressed in terms of multi-dimensional "relations" which can be effectively implemented as a FIFO system
- We mentioned two techniques which can transform FIFO system by over-approximation into:
  - a finite state machine
  - a Petri-Net
- This is work in progress ... especially meaningful FIFO system construction

Thank you

Thank you!

Thank you

Thank you!

the end