Deductive verification of distributed systems with PVS theorem prover—Part 1

CS141a: Distributed Systems Laboratory

sayan mitra

mitras@caltech.edu

CALIFORNIA INSTITUTE OF TECHNOLOGY

January 2007

verifying distributed systems

- proving that a distributed system behaves correctly is difficult

verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency

verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures

verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures
  - real-time constraints

verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures
  - real-time constraints
- simulations and tests can be used to find bugs but cannot guarantee correctness

verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures
  - real-time constraints
- simulations and tests can be used to find bugs but cannot guarantee correctness
- model based distributed system development

verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures
  - real-time constraints
- simulations and tests can be used to find bugs but cannot guarantee correctness
- model based distributed system development
  - describe distributed system and its environment as a mathematical object, e.g., a state machine or an automaton

verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures
  - real-time constraints
- simulations and tests can be used to find bugs but cannot guarantee correctness
- model based distributed system development
  - describe distributed system and its environment as a mathematical object, e.g., a state machine or an automaton
  - write the correctness requirements of the system as formulas in some logic, e.g., propositional logic, temporal logic

verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures
  - real-time constraints
- simulations and tests can be used to find bugs but cannot guarantee correctness
- model based distributed system development
  - describe distributed system and its environment as a mathematical object, e.g., a state machine or an automaton
  - write the correctness requirements of the system as formulas in some logic, e.g., propositional logic, temporal logic
  - prove that the automaton satisfies the requirements

verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures
  - real-time constraints
- simulations and tests can be used to find bugs but cannot guarantee correctness
- model based distributed system development
  - describe distributed system and its environment as a mathematical object, e.g., a state machine or an automaton
  - write the correctness requirements of the system as formulas in some logic, e.g., propositional logic, temporal logic
  - prove that the automaton satisfies the requirements
  - generate verified executable code from automaton through successive refinements

overview of tutorial

- quick introduction to PVS—a theorem prover for high-order logic
  - PVS specification language
  - prover commands
- specifying distributed algorithms in PVS
- proving properties of algorithms using PVS

propositional logic

$$P := \mathit{true} \mid \mathit{false} \mid \neg P_1 \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2$$

propositional logic

$$P := \mathit{true} \mid \mathit{false} \mid \neg P_1 \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2$$

sentences are built from finitely many atomic propositions $\{P_i\}$

propositional logic

$$P := \mathit{true} \mid \mathit{false} \mid \neg P_1 \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2$$

sentences are built from finitely many atomic propositions $\{P_i\}$

validity and satisfiability of any propositional sentence can be checked by construcing the truth table

propositional logic

$$P := \mathit{true} \mid \mathit{false} \mid \neg P_1 \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2$$

sentences are built from finitely many atomic propositions $\{P_i\}$

validity and satisfiability of any propositional sentence can be checked by construcing the truth table

propositional logic is decidable

propositional logic

$$P := \mathit{true} \mid \mathit{false} \mid \neg P_1 \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2$$

sentences are built from finitely many atomic propositions $\{P_i\}$

validity and satisfiability of any propositional sentence can be checked by construcing the truth table

propositional logic is decidable

many interesting problems can be expressed in propositional logic, e.g., circuit design, hardware verification

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\, x \in \mathbb{R},\ \exists\, n \in \mathbb{N}, n > x$

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\, x \in \mathbb{R},\ \exists\, n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\, x \in \mathbb{R},\ \exists\, n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
  - cannot quantify over functions and predicates

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\, x \in \mathbb{R},\ \exists\, n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
  - cannot quantify over functions and predicates
- only certain fragments of FOL are decidable

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\, x \in \mathbb{R},\ \exists\, n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
  - cannot quantify over functions and predicates
- only certain fragments of FOL are decidable
  - E.g., monadic formulas: no function symbols, only unary predicates

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\ x \in \mathbb{R},\ \exists\ n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
  - cannot quantify over functions and predicates
- only certain fragments of FOL are decidable
  - E.g., monadic formulas: no function symbols, only unary predicates
- higher order logic (HOL):

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\, x \in \mathbb{R},\ \exists\, n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
  - cannot quantify over functions and predicates
- only certain fragments of FOL are decidable
  - E.g., monadic formulas: no function symbols, only unary predicates
- higher order logic (HOL):
  - more expressive $\Rightarrow$ allows natural description of systems

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\ x \in \mathbb{R},\ \exists\ n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
  - cannot quantify over functions and predicates
- only certain fragments of FOL are decidable
  - E.g., monadic formulas: no function symbols, only unary predicates
- higher order logic (HOL):
  - more expressive $\Rightarrow$ allows natural description of systems
  - harder to decide $\Rightarrow$ fully automatic verification not possible

PVS

- Prototype Verification System (Version 4.1)
  `http://pvs.csl.sri.com/`

PVS

- Prototype Verification System (Version 4.1)
  `http://pvs.csl.sri.com/`
- a specification language, a theorem prover, and much more ...

PVS

- Prototype Verification System (Version 4.1)
  `http://pvs.csl.sri.com/`
- a specification language, a theorem prover, and much more ...
- the PVS specification language is based on HOL; typed lambda calculus

PVS

- Prototype Verification System (Version 4.1)
  `http://pvs.csl.sri.com/`
- a specification language, a theorem prover, and much more ...
- the PVS specification language is based on HOL; typed lambda calculus
- the PVS prover is an interactive theorem prover with built-in semi-decision procedures

PVS

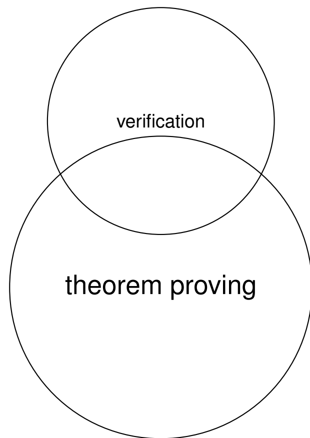
- Prototype Verification System (Version 4.1)
  `http://pvs.csl.sri.com/`
- a specification language, a theorem prover, and much more ...
- the PVS specification language is based on HOL; typed lambda calculus
- the PVS prover is an interactive theorem prover with built-in semi-decision procedures
- relatively easy to plug in new proof strategies and decision procedures

PVS

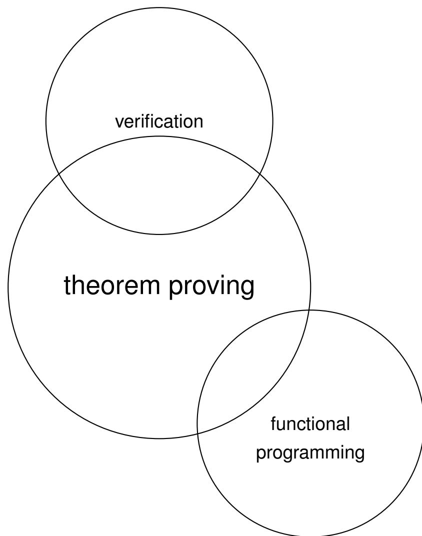
- Prototype Verification System (Version 4.1)
  `http://pvs.csl.sri.com/`
- a specification language, a theorem prover, and much more ...
- the PVS specification language is based on HOL; typed lambda calculus
- the PVS prover is an interactive theorem prover with built-in semi-decision procedures
- relatively easy to plug in new proof strategies and decision procedures
- written in LISP, version 4.1 is open source

theorem proving and other areas of CS

theorem proving and other areas of CS



theorem proving and other areas of CS



theorem proving and other areas of CS

verification

theorem proving

logic

functional programming

theorem proving and other areas of CS

verification

theorem proving

PVS

logic

functional
programming

theorem proving and other areas of CS

verification

proof
methods

theorem proving

PVS

decision
procedures

LISP

logic

functional
programming

example 1: a theory of stack of integers

Stack: **theory begin**

Stack: **type** = [**#** *length*: **nat**, *seq*: [*below*[*length*] -> **nat**] **#**]

example 1: a theory of stack of integers

Stack: **theory begin**

Stack: **type** = [# *length*: **nat**, *seq*: [*below*[*length*] -> **nat**] #]

NonEmptyStack?(*c*:*Stack*): **bool** = *c*'*length* /= **0**

NonEmptyStack: **type** = (*NonEmptyStack*?)

example 1: a theory of stack of integers

Stack: **theory begin**

Stack: **type** = [# *length*: **nat**, *seq*: [*below*[*length*] -> **nat**] #]

NonEmptyStack?(*c*:*Stack*): **bool** = *c*‘*length* /= **0**

NonEmptyStack: **type** = (*NonEmptyStack*?)

length(*c*:*Stack*):**nat** = *c*‘*length*

top(*c*:*NonEmptyStack*):**nat** = *q*‘*seq*(*length*(*c*)-**1**)

example 1: a theory of stack of integers

```
Stack: theory begin

Stack: type = [# length: nat, seq: [below[length] -> nat ] # ]

NonEmptyStack?(c:Stack): bool = c`length /=  0

NonEmptyStack: type = (NonEmptyStack?)

length(c:Stack):nat = c`length

top(c:NonEmptyStack):nat = q`seq(length(c)-1)

push(c:stack, a:nat):NonEmptyStack =
(# length :=  c`length + 1,
seq :=  seq(c) with [(c`length) :=  a] #)

pop(c:NonEmptyStack):[Stack,nat ]

end Stack
```

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- all functions are total

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- all functions are total
- type/function definitions can be concrete, e.g., *top*, or uninterpreted, e.g., *pop*

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- all functions are total
- type/function definitions can be concrete, e.g., *top*, or uninterpreted, e.g., *pop*
- a predicate B on type T automatically defines a subtype (B) of T, e.g., $(NonEmptyStack?)$ is a subtype of *Stack*

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- all functions are total
- type/function definitions can be concrete, e.g., *top*, or uninterpreted, e.g., *pop*
- a predicate *B* on type *T* automatically defines a subtype (*B*) of *T*, e.g., (*NonEmptyStack?*) is a subtype of *Stack*
- all assignments and definitions must be type-correct

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- all functions are total
- type/function definitions can be concrete, e.g., *top*, or uninterpreted, e.g., *pop*
- a predicate B on type T automatically defines a subtype (B) of T, e.g., $(NonEmptyStack?)$ is a subtype of *Stack*
- all assignments and definitions must be type-correct
- typechecking is in general undecidable; PVS generates proof obligations or type correctness conditions (TCCs). E.g., application of $pop(c)$ generates the TCC $NonEmptyStack\mathbf{?}(c)$

some properties of stacks

```
Stack: theory begin

...
c: var Stack
a: var nat

nonempty: lemma forall (c,a): NonEmptyStack?(push(c,a))

idem : lemma forall (c, a): pop(push(c , a))`1 = c

pushpop: lemma forall (c, a): pop(push(c,a))`2 = a

end Stack
```

a polymorphic stack

```
Stack[T:type+]: theory begin

Stack: type = [# length: nat, seq: [below[length] -> T] #]
...
c: var Stack
a: var T

nonempty: lemma forall (c,a): NonEmptyStack?(push(c,a))

idem : lemma forall (c, a): pop(push(c , a))`1 = c

pushpop: lemma forall (c, a): pop(push(c,a))`2 = a

end Stack
```

inductive definitions and recursive functions

$even(n\mathbf{:}\mathbf{nat})\mathbf{:}$ **inductive** **bool** $= n = \mathbf{0}$ **or** $n > \mathbf{1}$ **and** $even(n\text{-}\mathbf{2})$

inductive definitions and recursive functions

even(*n*:**nat**): **inductive bool** $=$ *n* $=$ **0** **or** *n* $>$ **1** **and** *even*(*n*-**2**)

fact(*n*:**nat**): **recursive nat** $=$ **if** *n* $=$ **0** **then** **1** **else** *n* $*$ *fact*(*n*-**1**) **endif**
measure lambda (*n*:**nat**):*n*

inductive definitions and recursive functions

even(*n*:**nat**): **inductive bool** $=$ *n* $=$ **0** **or** *n* $>$ **1** **and** *even*(*n*-**2**)

fact(*n*:**nat**): **recursive nat** $=$ **if** *n* $=$ **0** **then** **1** **else** *n* $*$ *fact*(*n*-**1**) **endif**
measure lambda (*n*:**nat**):*n*

- inductive definitions cannot be used as rewrite rules

inductive definitions and recursive functions

even(*n*:**nat**): **inductive bool** $=$ *n* $=$ **0** **or** *n* $>$ **1** **and** *even*(*n*-**2**)

fact(*n*:**nat**): **recursive nat** $=$ **if** *n* $=$ **0** **then** **1** **else** *n* $*$ *fact*(*n*-**1**) **endif**
measure lambda (*n*:**nat**):*n*

- inductive definitions cannot be used as rewrite rules
- mutual recursion not allowed

inductive definitions and recursive functions

even(*n*:**nat**): **inductive bool** $=$ *n* $=$ **0** **or** *n* $>$ **1** **and** *even*(*n*-**2**)

fact(*n*:**nat**): **recursive nat** $=$ **if** *n* $=$ **0** **then** **1** **else** *n* $*$ *fact*(*n*-**1**) **endif**
measure lambda (*n*:**nat**):*n*

- inductive definitions cannot be used as rewrite rules
- mutual recursion not allowed
- domain of the measure function is the same domain as the recursive function being defined and its range must be a well-founded set with a order relation

polymorphic theory of automata

simplemachine[
states, *actions*: **type**,
enabled: [*actions*,*states* -> **bool**],
trans: [*actions*,*states* -> *states*],
start: [*states* -> **bool**]
]: **theory**

polymorphic theory of automata

```
simplemachine[
states, actions: type,
enabled: [actions,states -> bool],
trans: [actions,states -> states],
start: [states -> bool]
]: theory

reachable_hidden(s,n): recursive bool =
if n = 0 then start(s)
 else (exists a, s1 : reachable_hidden(s1,n -1) and
enabled(a,s1) and s = trans(a,s1))
 endif
```

polymorphic theory of automata

```
simplemachine[
states, actions: type,
enabled: [actions,states -> bool],
trans: [actions,states -> states],
start: [states -> bool]
]: theory

reachable_hidden(s,n): recursive bool =
if n = 0 then start(s)
 else (exists a, s1 : reachable_hidden(s1,n -1) and
enabled(a,s1) and s = trans(a,s1))
 endif

measure (lambda s,n: n)
```

polymorphic theory of automata

```
simplemachine[
states, actions: type,
enabled: [actions,states -> bool],
trans: [actions,states -> states],
start: [states -> bool]
]: theory

reachable_hidden(s,n): recursive bool =
if n = 0 then start(s)
 else (exists a, s1 : reachable_hidden(s1,n -1) and
enabled(a,s1) and s = trans(a,s1))
 endif

measure (lambda s,n: n)

reachable(s): bool = exists n : reachable_hidden(s,n)
```

polymorphic theory of automata

base(*Inv*) **:** **bool** $=$ **forall** *s***:** *start*(*s*)
implies *Inv*(*s*)

inductstep(*Inv*) **:** **bool** $=$ **forall** *s***,** *a***:** *reachable*(*s*) **and** *Inv*(*s*) **and**
enabled(*a***,***s*) **implies** *Inv*(*trans*(*a***,***s*))

polymorphic theory of automata

base(*Inv*) **:** **bool** $=$ **forall** *s***:** *start*(*s*)
implies *Inv*(*s*)

inductstep(*Inv*) **:** **bool** $=$ **forall** *s***,** *a***:** *reachable*(*s*) **and** *Inv*(*s*) **and**
enabled(*a***,***s*) **implies** *Inv*(*trans*(*a***,***s*))

inductthm(*Inv*)**:** **bool** $=$ *base*(*Inv*) **and** *inductstep*(*Inv*)
implies (**forall** *s* **:** *reachable*(*s*) **implies** *Inv*(*s*))

example: specifying an automaton

an automaton is specified by the following components:

- *states*:**type+**

example: specifying an automaton

an automaton is specified by the following components:

- *states*:**type**+
- *actions*:**type**

example: specifying an automaton

an automaton is specified by the following components:

- *states*:**type+**
- *actions*:**type**
- *enabled*:[*states*, *actions* -> **bool**]

example: specifying an automaton

an automaton is specified by the following components:

- *states*:**type+**
- *actions*:**type**
- *enabled*:[*states*, *actions* -> **bool**]
- *trans*:[*states*, *actions* -> *states*]

example: specifying an automaton

an automaton is specified by the following components:

- *states*:**type+**
- *actions*:**type**
- *enabled*:[*states*, *actions* -> **bool**]
- *trans*:[*states*, *actions* -> *states*]

does this force transitions to be deterministic?

example: specifying an automaton

an automaton is specified by the following components:

- *states*:**type+**
- *actions*:**type**
- *enabled*:[*states*, *actions* -> **bool**]
- *trans*:[*states*, *actions* -> *states*]

does this force transitions to be deterministic?

no! push internal nondeterministic choices to (external) choice over actions

many more types of types

- enumerations $\mathit{color}\mathbf{:}\ \mathbf{type} = [\mathit{red}, \mathit{orange}, \mathit{green}]$

many more types of types

- enumerations $\mathit{color}\textbf{:}\ \textbf{type} = [\mathit{red}, \mathit{orange}, \mathit{green}]$
- tuple $\mathit{states}\textbf{:}\ \textbf{type} = [\textbf{nat}, \textbf{real}, \mathit{color}]$

many more types of types

- enumerations *color*: **type** = [*red*, *orange*, *green*]
- tuple *states*: **type** = [**nat**, **real**, *color*]
- record *states2*: **type** = [**#** *counter*:**nat**, *timer*:**real**, *light*:*color* **#**]

many more types of types

- enumerations *color*: **type** = [*red*, *orange*, *green*]
- tuple *states*: **type** = [**nat**, **real**, *color*]
- record *states2*: **type** = [**#** *counter*:**nat**, *timer*:**real**, *light*:*color* **#**]
- functions

  Values: **type** = [*I* -> **nat**]

  Values: **type** = **function** [*I* -> **nat**]

  Values: **type** = **array** [*I* -> **nat**]

many more types of types

- enumerations *color*: **type** = [*red*, *orange*, *green*]
- tuple *states*: **type** = [**nat**, **real**, *color*]
- record *states2*: **type** = [**#** *counter*:**nat**, *timer*:**real**, *light*:*color* **#**]
- functions

  Values: **type** = [*I* -> **nat**]

  Values: **type** = **function** [*I* -> **nat**]

  Values: **type** = **array** [*I* -> **nat**]
- dependent types

  Queue: [**#** *length*: **nat**, *seq*:[{*n*:**nat** |*n* < *length*} -> *t*] **#**]

many more types of types

- enumerations *color*: **type** = [*red*, *orange*, *green*]
- tuple *states*: **type** = [**nat**, **real**, *color*]
- record *states2*: **type** = [**#** *counter*:**nat**, *timer*:**real**, *light*:*color* **#**]
- functions

  Values: **type** = [*I* -> **nat**]

  Values: **type** = **function** [*I* -> **nat**]

  Values: **type** = **array** [*I* -> **nat**]
- dependent types

  Queue: [**#** *length*: **nat**, *seq*:[{*n*:**nat** | $n <$ *length*} -> *t*] **#**]

ID:**type** = {**1,2,3,4**}
location:**type** = [*x*:**real**, *y*:**real**]

states: [**#** *pos*:[*ID* -> *location*], *clock*:[*ID* -> **posreal**], *failed*:[*ID* -> **bool**] **#**]

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

```
actions: datatype
fail(i:ID):fail?
time_elapse(t:posreal):time_elapse?
send(i:ID,m:location):send?
receive(i:ID,m:location):receive?
end actions
```

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

```
actions: datatype
fail(i:ID):fail?
time_elapse(t:posreal):time_elapse?
send(i:ID,m:location):send?
receive(i:ID,m:location):receive?
end actions
```

- defines a new type called actions

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

actions: **datatype**
fail(*i*:*ID*):*fail*?
time_elapse(*t*:**posreal**):*time_elapse*?
send(*i*:*ID*,*m*:*location*):*send*?
receive(*i*:*ID*,*m*:*location*):*receive*?
end *actions*

- defines a new type called actions
- *a_f3*: *actions* = *fail*(**3**) is a constant of type action

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

actions: **datatype**
fail(*i*:*ID*):*fail*?
time_elapse(*t*:**posreal**):*time_elapse*?
send(*i*:*ID*,*m*:*location*):*send*?
receive(*i*:*ID*,*m*:*location*):*receive*?
end *actions*

- defines a new type called actions
- *a_f3*: *actions* = *fail*(**3**) is a constant of type action
  - *fail*?(*a_f3*) returns true

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

actions: **datatype**
fail(*i*:*ID*):*fail*?
time_elapse(*t*:**posreal**):*time_elapse*?
send(*i*:*ID*,*m*:*location*):*send*?
receive(*i*:*ID*,*m*:*location*):*receive*?
end *actions*

- defines a new type called actions
- a_f3: $actions = fail(\mathbf{3})$ is a constant of type action
  - *fail*?(*a_f3*) returns true
  - *time_elapse*?(*a_f3*) returns false

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

actions: **datatype**
fail(*i*:*ID*):*fail*?
time_elapse(*t*:**posreal**):*time_elapse*?
send(*i*:*ID*,*m*:*location*):*send*?
receive(*i*:*ID*,*m*:*location*):*receive*?
end *actions*

- defines a new type called actions
- *a_f3*: *actions* = *fail*(**3**) is a constant of type action
  - *fail*?(*a_f3*) returns true
  - *time_elapse*?(*a_f3*) returns false
  - *i*(*a_f3*) returns 3

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

actions: **datatype**
fail(*i*:*ID*):*fail***?**
time_elapse(*t*:**posreal**):*time_elapse***?**
send(*i*:*ID*,*m*:*location*):*send***?**
receive(*i*:*ID*,*m*:*location*):*receive***?**
end *actions*

- defines a new type called actions
- *a_f3*: *actions* = *fail*(**3**) is a constant of type action
  - *fail***?**(*a_f3*) returns true
  - *time_elapse***?**(*a_f3*) returns false
  - *i*(*a_f3*) returns 3
  - what is *i*(*time_elapse*(**10**)) ?

defining enabling conditions and transitions

```
enabled(a:actions, s:states):bool =
cases a of
fail(i):
not failed(s)(i)
```

defining enabling conditions and transitions

```
enabled(a:actions, s:states):bool =
cases a of
fail(i):
not failed(s)(i)

send(i,m):
pos(s)(i) = m
...
endcases
```

defining enabling conditions and transitions

$enabled(a\!:\!actions,\ s\!:\!states)\!:\!$**bool** $=$
cases *a* **of**
$fail(i)$**:**
not $failed(s)(i)$

$send(i,m)$**:**
$pos(s)(i) = m$
...
endcases

$trans(a\!:\!actions,\ s\!:\!states)\!:\!states =$
cases *a* **of**
$time_elapse(t)$**:**
s **with** $[clock := clock(s) + t]$

defining enabling conditions and transitions

```
enabled(a:actions, s:states):bool =
cases a of
fail(i):
not failed(s)(i)

send(i,m):
pos(s)(i) = m
...
endcases

trans(a:actions, s:states):states =
 cases a of
time_elapse(t):
s with [clock := clock(s) + t]

fail(i):
s with [failed := failed(s) with [(i) := true]
...
 endcases
```

references

1. PVS system guide `http://pvs.csl.sri.com/doc/pvs-system-guide.pdf`
   Read chapter 2 for basic instructions about the user interface
2. PVS language `http://pvs.csl.sri.com/doc/pvs-language-reference.pdf`
3. PVS prover guide `http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf`