# Using First-order Theorem Provers in Verification

Ruzica Piskac

Digital Enterprise Research Institute, Innsbruck, Austria
ruzica.piskac@deri.org

Lecture hold at EPFL, May 2007

# Motivation

How to obtain software reliability?

- Testing.
  Gives no guarantees for future runs.

# Motivation

How to obtain software reliability?

- Testing.
  Gives no guarantees for future runs.
- Result checking.
  One still needs to trust the checker.

# Motivation

How to obtain software reliability?

- Testing.
  Gives no guarantees for future runs.
- Result checking.
  One still needs to trust the checker.
- Formal verification.
  "Königsweg", though hard, tedious, and itself error prone.

# Motivation

How to obtain software reliability?

- Testing.
  Gives no guarantees for future runs.
- Result checking.
  One still needs to trust the checker.
- Formal verification.
  "Königsweg", though hard, tedious, and itself error prone.

➜ Apply formal verification on result checkers.

# Motivation

How to obtain software reliability?

- Testing.
  Gives no guarantees for future runs.
- Result checking.
  One still needs to trust the checker.
- Formal verification.
  "Königsweg", though hard, tedious, and itself error prone.

➜ Apply formal verification on result checkers.
➜ Do formal verification automatically.

# Motivation

How to obtain software reliability?

- Testing.
  Gives no guarantees for future runs.
- Result checking.
  One still needs to trust the checker.
- Formal verification.
  "Königsweg", though hard, tedious, and itself error prone.

➜ Apply formal verification on result checkers.
➜ Do formal verification automatically.

**Our result**

We formally verified a checker for priority queues in the first-order theorem prover Saturate.

# Priority Queues

## Definition

`p_queue<E,I>` is a standard data structure, where $E$ is a linearly ordered set (priorities), and $I$ is the type of data (items) stored in the queue. It supports the following operations:

- create
- insert$(e, i)$
- delete$(e, i)$
- find_min
- del_min

## Note

The results returned by find_min and del_min have to be checked

# Checking Priority Queues

## On-line Checker

Whenever `del_min` returns an element, then go through the priority queue and check that there is no smaller element.

# Checking Priority Queues

## On-line Checker

Whenever `del_min` returns an element, then go through the priority queue and check that there is no smaller element.

## Problem

This is too costly. PQ finds minimal element in time $O(\log(n))$. Going through all elements takes time $O(n)$.

# Checking Priority Queues

## On-line Checker

Whenever `del_min` returns an element, then go through the priority queue and check that there is no smaller element.

## Problem

This is too costly. PQ finds minimal element in time $O(\log(n))$. Going through all elements takes time $O(n)$.

## Solution

Postpone detection of errors. An error is detected, not when a non-minimal element is returned, but at the moment when a smaller element is returned.

# Checking Priority Queues

- during the execution of a correct priority queue, one collects knowledge about elements in it.
- if `del_min` returns some element $e$ then one knows that all other elements $e'$ satisfy $e' \geq e$.

# Checking Priority Queues

- during the execution of a correct priority queue, one collects knowledge about elements in it.
- if `del_min` returns some element $e$ then one knows that all other elements $e'$ satisfy $e' \geq e$.
- to every element $e$ we associate its lower bound.
- the lower bound of an element is the maximal priority reported by all `del_min` operations after the insertion of the element.

# Checking Priority Queues

- during the execution of a correct priority queue, one collects knowledge about elements in it.
- if `del_min` returns some element $e$ then one knows that all other elements $e'$ satisfy $e' \geq e$.
- to every element $e$ we associate its lower bound.
- the lower bound of an element is the maximal priority reported by all `del_min` operations after the insertion of the element.

➜ priority queue checker maintains system of lower bounds

# Priority Queue Checker

## Example

PRIORITY QUEUE OPERATIONS:

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

LOWER BOUNDS SYSTEM:

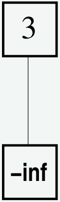
# Priority Queue Checker

## Example

PRIORITY QUEUE OPERATIONS:

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

LOWER BOUNDS SYSTEM:



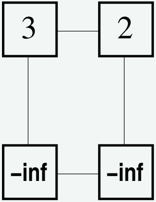
# Priority Queue Checker

## Example

PRIORITY QUEUE OPERATIONS:

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

LOWER BOUNDS SYSTEM:



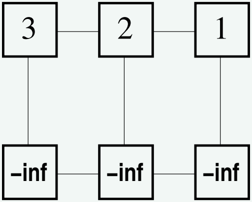
# Priority Queue Checker

## Example

**PRIORITY QUEUE OPERATIONS:**

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

**LOWER BOUNDS SYSTEM:**



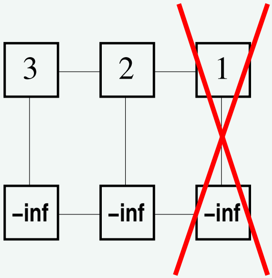
# Priority Queue Checker

## Example

PRIORITY QUEUE OPERATIONS:

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

LOWER BOUNDS SYSTEM:



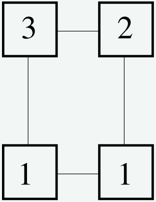
# Priority Queue Checker

## Example

PRIORITY QUEUE OPERATIONS:

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

LOWER BOUNDS SYSTEM:



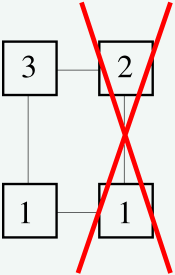
# Priority Queue Checker

## Example

PRIORITY QUEUE OPERATIONS:

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

LOWER BOUNDS SYSTEM:



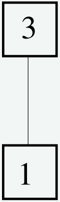
# Priority Queue Checker

## Example

PRIORITY QUEUE OPERATIONS:

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

LOWER BOUNDS SYSTEM:



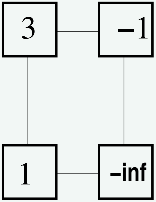
# Priority Queue Checker

## Example

PRIORITY QUEUE OPERATIONS:

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

LOWER BOUNDS SYSTEM:



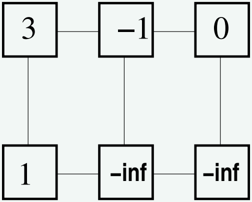
# Priority Queue Checker

## Example

PRIORITY QUEUE OPERATIONS:

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

LOWER BOUNDS SYSTEM:



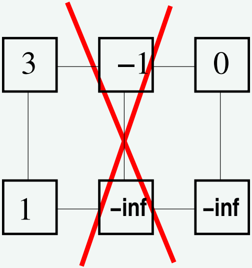
# Priority Queue Checker

## Example

PRIORITY QUEUE OPERATIONS:

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

LOWER BOUNDS SYSTEM:



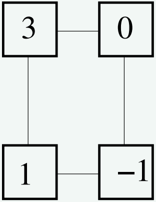
# Priority Queue Checker

## Example

PRIORITY QUEUE OPERATIONS:

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

LOWER BOUNDS SYSTEM:



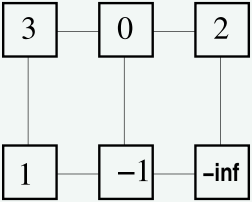
# Priority Queue Checker

## Example

PRIORITY QUEUE OPERATIONS:

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

LOWER BOUNDS SYSTEM:



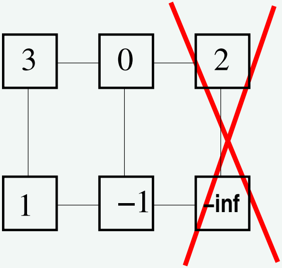
# Priority Queue Checker

## Example

**Priority queue operations:**

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

**Lower bounds system:**



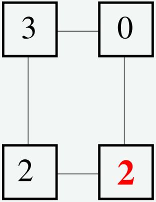
# Priority Queue Checker

## Example

PRIORITY QUEUE OPERATIONS:

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

LOWER BOUNDS SYSTEM:



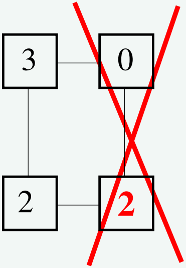
# Priority Queue Checker

## Example

PRIORITY QUEUE OPERATIONS:

```
p_queue Q;
Q.insert(3);
Q.insert(2);
Q.insert(1);
int p = Q.del_min();
Q.del_item(2);
Q.insert(-1);
Q.insert(0);
int p = Q.del_min();
Q.insert(2);
int p = Q.del_min();
Q.del_item(0);
```

LOWER BOUNDS SYSTEM:



# Verification Process

## Ideal workflow

1. express correctness theorem as first-order formula $F$
2. run first-order theorem prover on $F$
3. prover terminates with “$F$ holds”

# Verification Process

## Ideal workflow

1. express correctness theorem as first-order formula $F$
2. run first-order theorem prover on $F$
3. prover terminates with "$F$ holds"

## Problems in real life

- expressing correctness is non-trivial
  - what is a priority queue?
  - what is the checker doing?
- theorem prover cannot handle induction
  ➜ generate induction hypotheses by hand
- theorem prover runs out of memory
  ➜ guide theorem prover by manually inserting lemmas

# Priority Queues in FOL

## Inductive definition of priority queues

Let $E$ be a linearly ordered set. The set of priority queues $PQ$ is the smallest set satisfying:

- $\texttt{empty} \in PQ$
- if $q \in PQ$ and $e \in E$ then $\texttt{insert}(q, e) \in PQ$

# Priority Queues in FOL

## Inductive definition of priority queues

Let $E$ be a linearly ordered set. The set of priority queues $PQ$ is the smallest set satisfying:

- $\texttt{empty} \in PQ$
- if $q \in PQ$ and $e \in E$ then $\texttt{insert}(q, e) \in PQ$

## Specification of delete

$$\texttt{delete}(\texttt{empty}, e) = \texttt{empty}$$
$$\texttt{delete}(\texttt{insert}(q, e), e) = q$$
$$e_1 \neq e_2 \rightarrow$$
$$\quad \texttt{delete}(\texttt{insert}(q, e_1), e_2) = \texttt{insert}(\texttt{delete}(q, e_2), e_1)$$

# Lower Bounds System in FOL

## Definition of lower bound systems

A lower bounds system is a sequence of priority pairs:
LBS $= (E \times E)^*$

# Lower Bounds System in FOL

## Definition of lower bound systems

A lower bounds system is a sequence of priority pairs:
$\text{LBS} = (E \times E)^*$

## Specification of `adjust`

$\texttt{adjust}(\epsilon, e) = \epsilon$
$e_3 < e_1 \rightarrow \texttt{adjust}(l.(e_2, e_3), e_1) = \texttt{adjust}(l, e_1).(e_2, e_3)$
$e_1 \leq e_3 \rightarrow \texttt{adjust}(l.(e_2, e_3), e_1) = \texttt{adjust}(l, e_1).(e_2, e_3)$

# Defining the Goal

**"Theorem"**

If a sequence of commands is alert free and complete then it is also correct.

# Defining the Goal

**"Theorem"**

If a sequence of commands is alert free and complete then it is also correct.

- a sequence is alert free if every time the element is accessed, it is greater or equal to its lower bound.
- a sequence of commands is complete if after its execution the queue is empty.
- a sequence of commands is correct if every `del_min` operation returns indeed the minimal element.

# Alert free Sequences

## Sequence of Operations

Sequence of operations = word over the alphabet

$$\Sigma = \{\text{ins}(e), \text{del}(e), \text{dmin} \mid e \in E\}$$

# Alert free Sequences

## Sequence of Operations

Sequence of operations = word over the alphabet

$$\Sigma = \{\text{ins}(e), \text{del}(e), \text{dmin} \mid e \in E\}$$

## Alert free sequence

$\texttt{alert\_free}(\epsilon)$
$\texttt{alert\_free}(s.\text{ins}(e)) \Leftrightarrow \texttt{alert\_free}(s)$
$\texttt{alert\_free}(s.\text{del}(e)) \Leftrightarrow$
$\quad \texttt{alert\_free}(s) \wedge \texttt{lower\_bound}(e, \texttt{ex1}(s, \epsilon)) \leq e$
$\texttt{alert\_free}(s.\text{dmin}) \Leftrightarrow$
$\quad \texttt{alert\_free}(s) \wedge \texttt{lower\_bound}(e^*, \texttt{ex1}(s, \epsilon)) \leq e^*$
where $e^* = \texttt{elem}(\texttt{ex}(s.\text{dmin}, \mathrm{empty}))$

# Correctness Theorem

**Theorem**

*Let* $s \in \Sigma^*$ *then*
$\texttt{alert\_free}(s) \land \texttt{complete}(s) \Rightarrow \texttt{correct}(s)$.

# Correctness Theorem

**Theorem**

*Let* $s \in \Sigma^*$ *then*
$\mathtt{alert\_free}(s) \land \mathtt{complete}(s) \Rightarrow \mathtt{correct}(s)$.

**Proof**

The theorem is not inductive, so we cannot prove it. We need to formulate a more general (but inductive) theorem.

# Correctness Theorem

**Theorem**

*Let* $s \in \Sigma^*$ *then*

$$\texttt{alert\_free}(s) \wedge$$
$$\forall e \Big( \texttt{contains}(\texttt{queue}(\texttt{ex}(s, \texttt{empty})), e) \Rightarrow$$
$$\texttt{lower\_bound}(e, \texttt{ex1}(s, \epsilon)) \leq e \Big)$$
$$\Rightarrow \texttt{correct}(s).$$

# Correctness Theorem

## Theorem

*Let* $s \in \Sigma^*$ *then*

$$\texttt{alert\_free}(s) \wedge$$
$$\forall e \Big( \texttt{contains}(\texttt{queue}(\texttt{ex}(s, \texttt{empty})), e) \Rightarrow$$
$$\texttt{lower\_bound}(e, \texttt{ex1}(s, \epsilon)) \leq e \Big)$$
$$\Rightarrow \texttt{correct}(s).$$

# Correctness Theorem

**Theorem**

*Let* $s \in \Sigma^*$ *then*

$$\texttt{alert\_free}(s) \wedge$$
$$\forall e \Big( \texttt{contains}(\texttt{queue}(\texttt{ex}(s, \texttt{empty})), e) \Rightarrow$$
$$\texttt{lower\_bound}(e, \texttt{ex1}(s, \epsilon)) \leq e \Big)$$
$$\Rightarrow \texttt{correct}(s).$$

**Proof**

By induction on the length of word $s$.

# Further Contributions

## SEFM 2005

- We developed a specification that closely follows the concrete implementation and data structure in LEDA.
- We developed a framework that allows partial functions and non-deterministic behaviour
- Our formalization is a big benchmark for theorem provers: we had to insert and prove 88 lemmas

# Further Contributions

## SEFM 2005

- We developed a specification that closely follows the concrete implementation and data structure in LEDA.
- We developed a framework that allows partial functions and non-deterministic behaviour
- Our formalization is a big benchmark for theorem provers: we had to insert and prove 88 lemmas

## TPTP 2006

- The problem is being added to TPTP library
- Currently it is the only benchmark for lemma handling