Verification of an Off-Line Checker for Priority Queues

Hans de Nivelle, Ruzica Piskac
Max Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
{nivelle, rpiskac}@mpi-inf.mpg.de

Abstract

We formally verify the result checker for priority queues that is implemented in LEDA. We have developed a method, based on the notion of implementation, which links abstract specifications to concrete implementations. The method allows non-determinism in the abstract specifications that the concrete implementations have to fill in. We have formally verified that, if the checker has not reported an error up to a certain moment, then the structure it checks has behaved like a priority queue up to that moment. For the verification, we have used the first-order theorem prover Saturate.

1. Introduction

Everyone who has ever programmed knows that reliable software is hard to obtain. One of the approaches for obtaining reliability is *formal verification*. Verifying a program means giving the formal mathematical proof of its correctness. Verification is theoretically the best approach, because it guarantees that the program will behave correctly on every given input. However, in practice, verification is difficult. Verifying a program is much harder than writing a program, and it requires highly specialized skills. Small changes in the program may require a completely new correctness proof.

Another approach is *program result checking*, which was proposed by Blum [3]. Given some problem f, let P be a program that computes a solution for f. Let x and y represent the input and the output of P, respectively. Then a *program result checker* for program P is a new program C which can decide whether $y = f(x)$.

The result checker is run after the original program. The checker confirms correctness of the program's output or reports an error. It does not verify whether the program is correct, it only verifies that the output was correct on a given input.

Often it is easier to check whether $y = f(x)$ than to calculate y from x. For example, it is usually easier to check that x is a root of an equation $f(x) = 0$, than to calculate x from the equation.

In order to make checking easier, one may require that the program should not return only the output y, but also a correctness proof (so-called *witness*). The main requirement of the introduced witness w is that it should make it easy to check whether $y = f(x)$. "Easy" means that the resource requirements of checking should be lower than the requirements of the program itself. "Easy" also means that the checker should be simple enough so that its correctness is obvious or that it can be formally verified with present technology. Moreover, if $y \neq f(x)$ then there should exist no witness w such that the triple (x, y, w) would pass checking.

The checking of data structures can be seen as a generalized form of program checking. The user of a data structure calls functions which update the data structure and functions that return values depending on the state of the data structure. Result checking has to done by doing result checking on the functions that return some output, relative to the updates of the data structure that have taken place before.

However, contrary to the input to an algorithm, the sequence of updates to a data structure is not bounded in size. Therefore, the inputs to the functions that check the results of the data structure are also not bounded. If one wants to use a witness, then for program correctness checking, the witness has to be another data structure S' that is updated in parallel with the updates on the original data structure S. It should be constructed in such a way that it is easy to check whether the values returned by functions of S are correct, using S'.

Checking data structures is distinctive from checking programs because a wrong output of a data structure does not necessarily have to be detected immediately. It is also acceptable to detect it at a later moment after some more commands have been executed. Depending on the moment

at which errors are detected, one can distinguish two types of checkers: *on-line* and *off-line checkers*. A checker is called on-line when an incorrect value is detected at the moment when it is returned. A checker is called off-line, when an incorrect value will be detected eventually, but possibly at a later moment.

Often, off-line checkers are more time-efficient than on-line checkers. An example of this is the off-line checker for priority queues that we will define in Section 2.

In this paper we will formally verify the correctness of an off-line result checker for priority queues that is implemented in LEDA. Although ideally the checker should be so simple that its correctness is obvious, the checker in LEDA is quite sophisticated, and it might be useful to verify its correctness. In [6], the authors write that 'formal verification of simple checkers might be a realistic goal in the near future'.

In Section 2, we briefly introduce LEDA, and explain the intuition behind the priority queue checker. In Section 3, we introduce a mathematical model that formally expresses when an implementation implements a certain specification. After that, we extend the notion, so that it can express when a given implementation *conditionally implements* a given specification. Using this notion, we can state that a datastructure conditionally implements a priority queue on the condition that it is run in combination with the checker, and that the checker accepts its outputs. In Section 4, we give a formal proof of this statement.

2. An Off-line Priority Queue Checker

LEDA [8] is a C^{++}class library for efficient data types and algorithms. The priority queue checker implemented in LEDA [6] uses the encapsulation model for checking data structures. Sometimes in the literature this model is also called "the client-checker-server model" [1, 6]. In such a model, the priority queue checker is a program layer that acts as a connection between the user and the priority queue. The checker monitors the behaviour of the priority queue and if there was no error, it stays silent. In case of an error, the checker reports the error to the user. This means that on the user level, if the priority queue operates correctly, there is no difference between checked and unchecked priority queues. Since the checker implemented in LEDA is not an on-line checker, but an off-line one, a potential error will not be reported immediately but eventually.

The key idea of the checker is the fact that whenever we retrieve the minimal element all the remaining elements in the priority queue must have priority at least as large as the reported minimum, when the implementation is correct. Therefore, to each element of the priority queue we can assign a *lower bound*, which has to be smaller than the element itself, if the implementation is not faulty. The lower bound of the element changes through the time of the program execution.

Definition 1. *A lower bound of an element at time t is a maximal priority reported by all* `del_min` *and* `find_min` *operations performed between the moment when the element was inserted into the priority queue and the time t.*

Initially, when the element is inserted into the priority queue, its lower bound is set to $\perp$. Whenever the user executes the `del_min` or `find_min` command, the checker updates the existing lower bounds by increasing all lower bounds which were smaller than the reported minimal to the value of the reported minimum. Therefore, during the time of program execution, the lower bound of the element can only become greater.

The error detection is such that every time some element of the priority queue is accessed, the checker compares the element to its lower bound. If the element is greater or equal to its lower bound, no error is reported and the checker remains silent. But, if the element is strictly smaller than its lower bound, this indicates that an error occurred during the code execution and the checker alarms the user.

The resulting checker is an off-line checker: An error is reported not immediately when a non-minimal element is returned, but later when a smaller element is found. It was shown in [6], that there exists no on-line checker whose running time is less than the running time of the priority queue itself.

In the following section, we develop an abstract model for the priority queue and its checkers. We introduce a general condition when a concrete program implements an abstract data structure. Using this condition, we show that if the checked priority queue fails to implement the abstract queue, then an error will be eventually reported. With the exception of the generation of induction hypotheses, the proof has been formally verified using the Saturate system [2, 11]. Saturate is a first order theorem prover based on saturation and redundancy. The theoretical basis of Saturate includes the superposition calculus and the chaining calculus. Its main focus is on the efficient treatment of transitive relations by term rewrite techniques and on the restriction of the search space by applying sophisticated techniques for detecting redundancy of clauses.

3. A Mathematical Model for Priority Queue Checkers

In this section, we give a mathematical model for priority queues, lower bound systems, and priority queue checkers. The specification technique is very general, and it can be applied to other abstract data types as well. Its distinguishing feature is that it handles non-determinism in a functional context. Most existing methods for handling speci-

fication of non-determinism use transition relations [5, 9], because non-determinism is more naturally handled by relations than by functions. However, we preferred to stay as much as possible within the functional framework, because it allows to use the rewriting paradigm, and to have executable specifications as much as possible. Our method is closely related to the notion of strong simulation in [4]. The main difference is that our method allows *non-determinism* in the specification and *partial operations*.

We assume that an object of a data type has to be introduced through a constructor, and that during its existence, it can be modified only by its methods. As a consequence, a data type can be considered as an *inductively defined set*. For example, a priority queue is defined by a constructor that constructs an empty priority queue, and its methods that insert new elements, delete elements, or find the minimal element. Contrary to fundamental inductive sets in mathematics, data types are usually not *freely generated*. A set is freely generated if every element can be obtained in only one way through its constructors. Examples of freely generated inductive sets are the natural numbers or lists. It is not possible to construct the same list in two different ways using nil and cons. On the other hand, if we insert an element in a priority queue, and then remove the element again, one obtains the same priority queue back. (At least in the mathematical model, this needs not hold in the implementation) Depending on the data type, it can be the case that the data type is freely generated by a subset of its functions, but this need not always be the case.

First we give an informal definition of priority queues. After that we introduce some notation, with which we can introduce our implementation model and formal specifications of priority queues and priority queue checkers.

A priority queue assumes a quasio-ordered set:

Definition 2. *An QOSB (Quasi Ordered Set with Bottom Element) is a triple $(P, \leq, \bot)$, satisfying the following conditions:*

- $\bot \in P$.
- *For all* $p \in P$, $\bot \leq p$.
- *Relation* $\leq$ *is transitive on* P : *For all* $p_1, p_2, p_3 \in P$, *if* $p_1 \leq p_2$ *and* $p_2 \leq p_3$, *then also* $p_1 \leq p_3$.
- *Relation* $\leq$ *is total on* P : *For each pair* $p_1, p_2 \in P$, *either* $p_1 \leq p_2$ *or* $p_2 \leq p_1$.
- *The relation* $\leq$ *is reflexive on* P : *For each* $p \in P$, *we have* $p \leq p$.

We define the relation $<$ *from* $p_1 < p_2$ *iff* $p_1 \leq p_2$ *and* $p_2 \not\leq p_1$. *We define* $p_1 \equiv p_2$ *iff* $p_1 \leq p_2$ *and* $p_2 \leq p_1$.

The bottom element is not used by the priority queue, but it is needed for the system of lower bounds. Assuming a bottom element causes no loss of generality, since one can always extend a data type by a new element, and make this element minimal. We do not require that bottom elements are unique. The difference between a quasi-order and a usual order is that for usual orders $\equiv$ has to be the same relation as $=$.

Definition 3. *Let P be an QOSB. A* priority queue *over P is a container that is characterized by the following operations:*

- create_{PQ} *creates the empty priority queue.*
- $\mathsf{insert}(p)$ *inserts the element p into the queue.*
- $\mathsf{contains}(p)$ *returns true if the priority queue contains p.*
- $\mathsf{empty}()$ *returns true if the priority queue is empty.*
- $\mathsf{find\_min}()$ *returns a minimal element, if the queue is not empty.*
- $\mathsf{remove}(p)$ *removes p from the priority queue, if p is contained in the queue.*
- $\mathsf{remove\_min}()$ *finds and removes a minimal element, if the queue is not empty.*

Our definition of priority queue follows [10] in the fact that it requires only a single ordered set. An alternative would be to define the priority queue over a quasi-ordered set I and a data type D. The first type would be used for finding minimal elements, and the second type would be used for storing additional data in the priority queue. This approach was taken in LEDA, see [8, 14]. The priority queue using two types is not more general, because it can be simulated by the priority queue with single type, if one replaces the ordered set P by $I \times D$, and extends $\leq$ to $I \times D$ as follows: $(i_1, d_1) \leq (i_2, d_2)$ iff $i_1 \leq i_2$.

Before we can give a formal specification of priority queues, we introduce some additional notation. In object oriented languages, it is standard to use the notation $T\ p.\mathsf{method}()$ for declaring a method of data type P which returns a value of type T. Typically, $\mathsf{method}()$ does two things at the same time: it returns an object of type T and it can also modify p in the process. Therefore, $\mathsf{method}()$ defines two functions at the same time. The first function returns an object of type T, and the second function returns an object of type P. We will keep the notation $p.\mathsf{method}()$ for the first function, and introduce the notation $p/\mathsf{method}()$ for the second function. We assume that $/$ is left associative, so that one can write $p/\mathsf{method}_1()/\mathsf{method}_2()/\mathsf{method}_3()$ instead of $((p/\mathsf{method}_1())/\mathsf{method}_2())/\mathsf{method}_3()$. Different from the C^{++}-notation, we will use the notation

create_P for constructors, in which P is the type being constructed. In case there is no confusion about the type, we will mostly omit the P.

When axiomatizing abstract data types, we will distinguish between abstract specification of the data type and concrete implementations. The concrete implementations will be connected to the specifications by implementation functions. This method is general enough to prove the correctness of the standard implementation of priority queues, but also to show that a checked priority queue which does not report any errors, in fact is a priority queue. The distinction between abstract and concrete implementation is necessary because the same abstract object may have different concrete representations. Although different concrete representations at first cannot be distinguished by the abstract representation, they may diverge when methods are applied on them. As a consequence, we have to model a certain amount of non-determinism in the abstract specification that has to be filled in by the implementations. As an example, consider the heap implementation of priority queue. We want to assume in the abstract specification that two priority queues are identical if they contain the same set of elements. Two identical priority queues may be represented by different heaps, which can have different (but equivalent) elements on top. If one calls $\mathsf{find\_min}$, the two heaps will return different minimal elements. One could change the specification s.t. $\mathsf{find\_min}$ is totally specified, but in our view, non-determinism on the abstract level is an essential feature of object-oriented specification. Any attempt to force the specification of the priority queue to choose between equivalent, but distinct objects, would be artificial.

In our implementation model, we will model non-determinism in the specification by allowing the methods in the specification to have more arguments than the corresponding methods in the implementation. For priority queues, $\mathsf{find\_min}$ in the specification will have signature $\mathsf{find\_min}_p$, where p is an additional element that specifies which minimal element will be returned. $\mathsf{find\_min}_p$ will have the precondition that p is in the priority queue, and minimal. An implementation of priority queues will have to provide a function $\mathsf{find\_min}$ that behaves like $\mathsf{find\_min}_p$ for some p satisfying the precondition.

Figure 1 gives an axiomatization for priority queues. We omitted the identities for methods that obviously do not modify the priority queue. For example, we did not state $pq/\mathsf{empty} = pq$.

Some methods have preconditions. For example, $\mathsf{remove}(p)$ has precondition that p occurs in the queue. The preconditions should not be confused with case distinctions, which are marked by 'if'. In axioms and verification conditions we will usually not mention preconditions, but they have to be verified as well. Whenever one speaks about a method application $pq.f(p)$, one has to verify its precondition $\Pi(pq, p)$. For a method f, the functions $pq.f(p)$ and $pq/f(p)$ always have the same precondition. We will mention the precondition only once in specifications. It is allowed to use other methods in preconditions, but these methods then should not have preconditions by themselves.

Figure 1. Axioms for priority queues

Method $\mathsf{contains}(p)$ has no preconditions.
$\mathsf{create}.\mathsf{contains}(p) = \mathsf{false}$,
$pq/\mathsf{insert}(p_1).\mathsf{contains}(p_1) = \mathsf{true}$,
$pq/\mathsf{insert}(p_1).\mathsf{contains}(p_2) = pq.\mathsf{contains}(p_2)$ if $p_1 \neq p_2$,

Method empty has no preconditions.
$\mathsf{create}.\mathsf{empty} = \mathsf{true}$,
$pq/\mathsf{insert}(p).\mathsf{empty} = \mathsf{false}$,

Method $\mathsf{remove}(p)$ has precondition $\mathsf{contains}(p)$.
$pq/\mathsf{insert}(p)/\mathsf{remove}(p) = pq$,
$pq/\mathsf{insert}(p_1)/\mathsf{remove}(p_2) = pq/\mathsf{remove}(p_2)/\mathsf{insert}(p_1)$
if $p_1 \neq p_2$,

Methods $\mathsf{find\_min}_p$ and $\mathsf{remove\_min}_p$ have precondition $\mathsf{contains}(p) = \mathsf{true}$ and $\forall p' \;\; pq.\mathsf{contains}(p') \rightarrow p \leq p'$.
$pq.\mathsf{find\_min}_p = p$,
$pq.\mathsf{remove\_min}_p = p$,
$pq/\mathsf{remove\_min}_p = pq/\mathsf{remove}(p)$

Extensionality:
$pq/\mathsf{insert}(p_1)/\mathsf{insert}(p_2) = pq/\mathsf{insert}(p_2)/\mathsf{insert}(p_1)$.

The following example from standard mathematics where the integers are implemented by pairs of natural numbers, illustrates the use of embedding functions.

Example 4. *It is possible to introduce the integers $\mathbb{Z}$ from the natural numbers $\mathbb{N}$, by representing them as pairs (n_1, n_2), with $n_1, n_2 \in \mathbb{N}$, see [7]. The embedding from $\mathbb{N}$ to $\mathbb{Z}$ would be defined by $I(n_1, n_2) = n_1 - n_2$. One can define $+$ and $-$ on $\mathbb{Z}$ from $(n_1, n_2) + (m_1, m_2) = (n_1 + m_1, n_2 + m_2)$, $(n_1, n_2) - (m_1, m_2) = (n_1 + m_2, n_2 + m_1)$.*

In order to show that this embedding is correct, one would have to show that, for $z_1, z_2 \in \mathbb{Z}$, $I(z_1 + z_2) = I(z_1) + I(z_2)$, and $I(z_1 - z_2) = I(z_1) - I(z_2)$.

We will give a general definition of when a concrete type P implements an abstract type $P^{\#}$ with implementation function I. The definition follows Example 4. Assuming that $p.f(y)$ and y are not of type P, one needs to prove that for every $p \in P$,

$$I(p).f(y) = p.f(y) \text{ and } I(p)/f(y) = I(p/f(y)).$$

We use implicit overloading of methods. Mathematically seen, the f in $p.f(y)$ and $I(p).f(y)$ are different functions, but we reuse the f for both of them, in order to avoid introducing too many names.

If a parameter is of type P, then it has to be replaced by a parameter of type $P^\#$, and the conditions become

$$I(p).f(I(y)) = p.f(y) \text{ and } I(p)/f(I(y)) = I(\ p/f(y)\ ).$$

As for non-determinism, consider the case where the abstract type $P^\#$ has a non-deterministic method $f_x(y)$ with implicit parameter x. Assume that y is not of type $P^\#$. The implementation P has to provide a method $f(y)$, which for each input y finds an x, and behaves like $f_x(y)$. The conditions become

$$\exists x{:}X\ \ I(p).f_x(y) = p.f(y) \text{ and}$$

$$\exists x{:}X\ \ I(p)/f_x(y) = I(\ p/f(y)\ ).$$

We have assumed that x has type X, which is distinct from $P^\#$ and y has type Y which is also distinct from $P^\#$.

We are now ready for the general definition. It covers the cases we discussed above, and it allows partial methods.

Definition 5. *Let $P^\#$ be some abstract data type. Let f be a method of $P^\#$. We will use subscript notation for the implicit parameters of f. We assume that the implicit parameters are not of type $P^\#$. Furthermore, we assume that among the non-implicit parameters, the parameters of type $P^\#$ precede the parameters of other types. Hence we can write $f_{\overline{z}}(\overline{x};\overline{y})$ for an occurrence of f. We assume that the $\overline{x}$ have type $P^\#$, and that the $\overline{y}$ have types $\overline{Y}$, with each type in $\overline{Y}$ distinct from $P^\#$. Similarly, we assume that the $\overline{z}$ have types $\overline{Z}$, with each type in $\overline{Z}$ distinct from $P^\#$.*
Then a concrete type P implements *an abstract type $P^\#$ with* implementation function $I{:}P \to P^\#$ *if*

- *for every method $f_{\overline{z}}(\overline{x};\overline{y})$ of $P^\#$ with parameters of types $\overline{Z};P^\#;\overline{Y}$ and with precondition $\Pi^\#(p,\overline{x},\overline{y},\overline{z})$:*
- *there exists a method $f(\overline{x};\overline{y})$ of P with parameters of type $P;\overline{Y}$, and precondition $\Pi(p,\overline{x},\overline{y}) = \exists\overline{z}{:}\overline{Z}\ \ \Pi^\#(\ I(p),\ I(\overline{x}),\ \overline{y},\ \overline{z}\ )$,*

such that the following equations hold:

$$\begin{aligned}&\forall p{:}P\ \ \forall\overline{x}{:}P\ \ \forall\overline{y}{:}\overline{Y}\ \ \Pi(p,\overline{x},\overline{y}) \to\\ &\exists\overline{z}{:}\overline{Z}\ \ \Pi^\#(\ I(p),\ \overline{x},\ I(\overline{y}),\ \overline{z}\ )\ \wedge\\ &I(p).f_{\overline{z}}(\ I(\overline{x}),\ \overline{y}\ ) = p.f(\ \overline{x},\ \overline{y}\ ).\end{aligned} \tag{1}$$

$$\begin{aligned}&\forall p{:}P\ \ \forall\overline{x}{:}P\ \ \forall\overline{y}{:}\overline{Y}\ \ \Pi(p,\overline{x},\overline{y}) \to\\ &\exists\overline{z}{:}\overline{Z}\ \ \Pi^\#(\ I(p),\ \overline{x},\ I(\overline{y}),\ \overline{z}\ )\ \wedge\\ &I(p)/f_{\overline{z}}(\ I(\overline{x}),\ \overline{y}\ ) = I(\ p/f(\ \overline{x},\ \overline{y}\ )\ ).\end{aligned} \tag{2}$$

$$I(\mathsf{create}_P) = \mathsf{create}_{P^\#}. \tag{3}$$

Definition 5 can generalized in many ways: for example, we have assumed that only one type $P^\#$ is implemented by another type P. In practice, one often has a family of abstract types that need to be simultaneously implemented by concrete types. As an example of this situation, consider a container and its iterator type.

In addition, one may have a situation where the types in $y{:}Y$ depend on $P^\#$, without being equal to $P^\#$. In that case, this type has to be interpreted accordingly. As an example consider the case where one of the methods of $P^\#$ has a list of $P^\#$'s among its arguments.

Another possible generalization is a situation where the constructor has parameters, some of which possibly implicit.

In practice, Equations 1 - 3 need to hold only for public methods.

The standard implementation of priority queues is based on partially sorted array's called *heaps*. A heap over P is a an array h satisfying the following two conditions:

$$\forall i,j < h.\mathsf{size}\ \ (j = 2i+1) \to h[i] \le h[j], \text{ and}$$

$$\forall i,j < h.\mathsf{size}\ \ (j = 2i+2) \to h[i] \le h[j].$$

find_min can be implemented by returning $h[0]$ if $h.size > 0$. It can be shown that after an insertion or deletion, the heap property can be restored in time $O(\log(a.\mathsf{size}))$.

Let H be the type of heaps over P. An interpretation from H into $PQ^\#$ can be obtained by the program:

$PQ^\#\ pq^\# = \mathsf{create}_{PQ^\#}$;
for$(i = 0; i < h.\mathsf{size}; i{+}{+})$
  $pq^\#.\mathsf{insert}(\ h[i]\ )$;
return $pq^\#$;

The interpretation function I is not injective. Different heaps may represent the same priority queue. Different heaps representing the same priority queue, may return different elements when find_min is called.

Example 6. *We will illustrate Definition 5 for priority queues. Let PQ be some type. Let I a function from PQ to $PQ^\#$. We discuss the conditions that PQ needs to satisfy so that it implements $PQ^\#$ with I.*

1. *The constructor need to fulfill $I(\mathsf{create}_{PQ}) = \mathsf{create}_{PQ^\#}$.*
2. *PQ must have a method* insert, *s.t. for all $pq{:}PQ$ and $p{:}P$, $I(pq/\mathsf{insert}(p)) = I(pq)/\mathsf{insert}(p)$.*
3. *PQ must have a method* contains, *s.t. for all $pq{:}PQ$ and $p{:}P$, $pq.\mathsf{contains}(p) = I(pq).\mathsf{contains}(p)$.*
4. *PQ must have a method* empty, *s.t. for all $pq{:}PQ$, $pq.\mathsf{empty} = I(pq).\mathsf{empty}$.*

5. *The method* $\mathsf{find\_min}_p$ *of* $PQ^{\#}$ *has implicit parameter* p. *The precondition is* $\Pi^{\#}(pq^{\#}, p) \equiv$

$pq^{\#}.\mathsf{contains}(p)$ *and* $\forall p'\colon P,\ \ pq^{\#}.\mathsf{contains}(p') \rightarrow p \leq p'$.

It follows that the corresponding method find_min *of* PQ *must have precondition*

$$\Pi(pq) \equiv \exists p\colon P,\ I(pq).\mathsf{contains}(p) \text{ and}$$

$$\forall p'\colon P,\ \ I(pq).\mathsf{contains}(p') \rightarrow p \leq p'.$$

It can be shown by induction that $\Pi(pq)$ *is equivalent to* $I(pq).\mathsf{empty} = \mathsf{false}$, *which in turn is equivalent to* $pq.\mathsf{empty} = \mathsf{false}$, *because of Condition 4 in this example.*

In addition, find_min *needs to satisfy (1) in Definition 5:*
$\forall pq\colon PQ,\ \ pq.\mathsf{empty} = \mathsf{false} \rightarrow$

$$\exists p\colon P,\ \ I(pq).\mathsf{contains}(p) \text{ and}$$

$$\forall p'\colon P,\ \ I(pq).\mathsf{contains}(p') \rightarrow p \leq p' \text{ and}$$

$$pq.\mathsf{find\_min} = I(pq).\mathsf{find\_min}_p.$$

Using $I(pq).\mathsf{find\_min}_p = p$ *from the specification of* $P^{\#}$, *the last formula is first-order equivalent to*

$$\forall pq\colon PQ,\ \ pq.\mathsf{empty} = \mathsf{false} \rightarrow$$

$$pq.\mathsf{contains}(pq.\mathsf{find\_min}) \text{ and}$$

$$\forall p'\colon P,\ \ pq.\mathsf{contains}(p') \rightarrow pq.\mathsf{find\_min} \leq p'.$$

This is the standard specification of find_min.

6. *The method* remove_min *can be analyzed in the same way.*

We will now specify the checked priority queue, and state the theorem that we formally verified.

Definition 7. *The type* B boolean *consists of two values* false *and* true. *It must be the case that* false $\neq$ true. *Operations* not, and, or *are defined as usual.*

Definition 8. *Let* P *be an QOSB. A* lower bound system *(abbreviated LBS)* S *is a collection of pairs over* P. *If the pair* (p, q) *occurs in the lower bound system, then* p *is a lower bound of* q. *An LBS supports the following operations:*

- create_{LBS} *creates the empty LBS.*
- $\mathsf{contains}(p)$ *returns true if the system contains a lower bound for* p.
- $\mathsf{assign}(p, q)$ *extends the current system with element* p *and sets its lower bound to* q.
- $\mathsf{lookup}(p)$ *is defined on the condition that the system contains a lower bound for* p. *In that case, a lower bound for* p *is returned.*
- $\mathsf{remove}(p, q)$ *is defined on the condition that the system contains lower bound* q *associated to* p. *In that case,* q *is removed as a possible lower bound of* p.
- $\mathsf{update}(q)$ *replaces all lower bounds in the system, which are less than* q, *by* q.
- $\mathsf{contains\_pair}(p, q)$ *returns true if the system contains* p *with lower bound* q *attached to it.*
- empty *returns* true *if the system is empty.*

Figure 2. Specification of LBS

Method $\mathsf{contains}(p)$ has no precondition.
$\mathsf{create}.\mathsf{contains}(p) = \mathsf{false}$,
$s/\mathsf{assign}(p, q).\mathsf{contains}(p) = \mathsf{true}$,
$s/\mathsf{assign}(p_1, q).\mathsf{contains}(p_2) = s.\mathsf{contains}(p_2)$
if $p_1 \neq p_2$

Method $\mathsf{contains\_pair}(p, q)$ has no precondition.
$\mathsf{create}.\mathsf{contains\_pair}(p, q) = \mathsf{false}$,
$s/\mathsf{assign}(p, q).\mathsf{contains\_pair}(p, q) = \mathsf{true}$,
$s/\mathsf{assign}(p_1, q_1).\mathsf{contains\_pair}(p_2, q_2) =$
$s.\mathsf{contains\_pair}(p_2, q_2)$ if $p_1 \neq p_2$ or $q_1 \neq q_2$.

Method $\mathsf{lookup}_q(p)$ has precondition $\mathsf{contains\_pair}(p, q)$,
$s.\mathsf{lookup}_q(p) = q$,

Method $\mathsf{remove}(p, q)$ has precondition $\mathsf{contains\_pair}(p, q)$,
$s/\mathsf{assign}(p, q)/\mathsf{remove}(p, q) = s$,
$s/\mathsf{assign}(p_1, q_2)/\mathsf{remove}(p_2, q_2) =$
$s/\mathsf{remove}(p_2, q_2)/\mathsf{assign}(p_1, q_1)$
if $p_1 \neq p_2$ or $q_1 \neq q_2$.

Method update has no precondition.
$\mathsf{create}.\mathsf{update}(q) = \mathsf{create}$,
$s/\mathsf{assign}(p, q_1)/\mathsf{update}(q_2) =$
$s/\mathsf{update}(q_2)/\mathsf{assign}(p, q_2)$ if $q_1 < q_2$
$s/\mathsf{assign}(p, q_1)/\mathsf{update}(q_2) =$
$s/\mathsf{update}(q_2)/\mathsf{assign}(p, q_1)$ if $q_2 \leq q_1$

Method empty has no precondition.
$\mathsf{create}.\mathsf{empty} = \mathsf{true}$,
$s.\mathsf{assign}(p, q).\mathsf{empty} = s.\mathsf{false}$.

Since our goal is to prove that every implementation of priority queues that runs within a checked priority queue is in fact a priority queue, we need the notion of *priority queue pretender*.

Definition 9. *A* priority queue pretender PQP *is an object that has the same signature as a priority queue.*

At this moment, we have assumed nothing about the implementation of the priority queue pretender, except for the fact that it supports the operations create, remove, insert, find_min and remove_min.

Definition 10. *A* checked priority queue *is a triple* (pqp, s, ok), *in which* pqp *is priority queue pretender,* s *is an object implementing a system of lower bounds, and* ok *is a boolean that remembers whether an error occurred. The operations are defined as follows:*

- $\mathsf{create} = (\mathsf{create}_{PQP}, \mathsf{create}_{LBS}, \mathsf{true})$.
- $(pqp, s, ok)/\mathsf{insert}(p) =$
  $(pqp/\mathsf{insert}(p), s/\mathsf{assign}(p, \bot), ok)$.
- $(pqp, s, ok).\mathsf{contains}(p) = s.\mathsf{contains}(p)$.
- $(pqp, s, ok).\mathsf{size} = s.\mathsf{size}$.
- *If* $s.\mathsf{size} = 0$, *then*
  $(pqp, s, ok)/\mathsf{find\_min} = (pqp, s, \mathsf{false})$
  and $(pqp, s, ok).\mathsf{find\_min} = \bot$.
  Otherwise, let $p = pqp.\mathsf{find\_min}$.
  Then $(pqp, s, ok).\mathsf{find\_min} = p$, *and*
  - *if* $s.\mathsf{contains}(p)$ *and* $s.\mathsf{lookup}(p) \leq p$, *then*
    $(pqp, s, ok)/\mathsf{find\_min} =$
    $(pqp, s/\mathsf{update}(p), ok)$.
  - *if* $s.\mathsf{contains}(p) = \mathsf{false}$ *or* $p < s.\mathsf{lookup}(p)$, *then*
    $(pqp, s, ok)/\mathsf{find\_min} =$
    $(pqp, s/\mathsf{update}(p), \mathsf{false})$.
- *If* $s.\mathsf{contains}(p) = \mathsf{true}$, *then let* $q = s.\mathsf{lookup}(p)$.
  If $q \leq p$, *then* $(pqp, s, ok)/\mathsf{remove}(p) =$
  $(pqp/\mathsf{remove}(p, q), ok)$.
  If $q > p$, *then* $(pqp, s, ok)/\mathsf{remove}(p) =$
  (pqp, s, false).
  If $s.\mathsf{contains}(p) = \mathsf{false}$, *then*
  $(pqp, s, ok)/\mathsf{remove}(p) = (pqp, s, \mathsf{false})$.
- remove_min *is composed of* find_min *and* remove :
  $(pqp, s, ok)/\mathsf{remove\_min} =$
  $(pqp, s, ok)/\mathsf{remove}(\ (pqp, s, ok).\mathsf{find\_min}\ )$.
- *We need an additional function* check, *that checks the lower bounds present in* s :
  $(pqp, \mathsf{create}_{LBS}, ok).\mathsf{check} = \mathsf{true}$,
  $(pqp, s.\mathsf{assign}(p, q), ok).\mathsf{check} =$
  $(pqp, s, ok).\mathsf{check}$ *if* $q \leq p$
  $(pqp, s.\mathsf{assign}(p, q), ok).\mathsf{check} = \mathsf{false}$ *if* $p < q$

We intend to use the notion of implementation also for checked priority queues. We want to state a condition of the form: If some objects d has been embedded in the checker, and the checker accepts its behaviour, then d has behaved like a priority queue. If d is of type D, then we could try to prove that D implements the abstract type of priority queues, but that is more than we can prove. We know nothing about other objects of type D.

What is needed is a notion of partial interpretation. We don't know whether the complete D implements a priority queue, but we know it for the subset D' of D that occurred in this run of the checked priority queue.

Definition 11. *Let* ϕ *be a predicate over* P. *Let* $P^\#$ *be some abstract type. Let* I *be an interpretation function from* P *to* $P^\#$. *We say that* P conditionally implements $P^\#$ *on condition* ϕ *if the following modified versions of equations 1 - 3 hold:*

$$\begin{aligned} \forall p{:}P\ \ \forall \overline{x}{:}P\ \ \forall \overline{y}{:}\overline{Y}\ \ \Pi(p, \overline{x}, \overline{y}) \rightarrow \\ \phi(\ p/f(\overline{x}, \overline{y})\ ) \rightarrow \\ \exists \overline{z}{:}\overline{Z}\ \ \Pi^\#(\ I(p),\ \overline{x},\ I(\overline{y}),\ \overline{z}\ )\ \wedge \\ I(p).f_{\overline{z}}(\ I(\overline{x}),\ \overline{y}\ ) = p.f(\ \overline{x},\ \overline{y}\ ). \end{aligned} \tag{4}$$

$$\begin{aligned} \forall p{:}P\ \ \forall \overline{x}{:}P\ \ \forall \overline{y}{:}\overline{Y}\ \ \Pi(p, \overline{x}, \overline{y}) \rightarrow \\ \phi(\ p/f(\overline{x}, \overline{y})\ ) \rightarrow \\ \exists \overline{z}{:}\overline{Z}\ \ \Pi^\#(\ I(p),\ \overline{x},\ I(\overline{y}),\ \overline{z}\ )\ \wedge \\ I(p)/f_{\overline{z}}(\ I(\overline{x}),\ \overline{y}\ ) = I(\ p/f(\ \overline{x},\ \overline{y}\ )\ ). \end{aligned} \tag{5}$$

$$I(\mathsf{create}_P) = \mathsf{create}_{P^\#}. \tag{6}$$

The notion of conditional implementation should not be confused with the notion of weak simulation in [4]. The notion of weak simulation is intended for modeling the fact that only objects up to a certain size can be represented in computers. The notion of conditional implementation is intended for expressing that we can make claims only about the behaviour of an implemntation as far as it was checked by the checker.

Definition 12. *Let* $cpq_1 = (pqp_1, s_1, ok_1)$ *and* $cpq_2 = (pqp_2, s_2, ok_2)$ *be checked priority queues. We say that* cpq_1 *is a* predecessor of cpq_2 *if there is a (possibly empty) sequence of operations* $M_1, \ldots, M_n$, *s.t.* $cpq_1/M_1/\cdots/M_n = cpq_2$. *The predecessor relation can be inductively defined as follows:*

- *For every checked priority queue* cpq, *we have* $cpq \prec cpq$,
- *If* $cpq_1 \prec cpq_2$, *then also* $cpq_1 \prec cpq_2/\mathsf{insert}(p)$.
- *If* $cpq_1 \prec cpq_2$, *then also* $cpq_1 \prec cpq_2/\mathsf{remove}(p)$.

- *If* $cpq_1 \prec cpq_2$, *then also* $cpq_1 \prec cpq_2/\mathsf{find\_min}$.
- *If* $cpq_1 \prec cpq_2$, *then also* $cpq_1 \prec cpq_2/\mathsf{remove\_min}$.

We do not need to check preconditions in Definition 12, because checked priority queues are robust against failing preconditions.

We want to prove that if a checked priority queue (pqp, s, ok) has $ok = \mathsf{true}$, and $\mathsf{checked} = \mathsf{true}$, then pqp has behaved like a correct priority queue.

Theorem 13. *For every* $cpq' \in CPQ$ *we define* $\phi(cpq') \equiv \exists cpq_F(cpq' \prec cpq_F \wedge cpq_F.\mathsf{ok} \wedge cpq_F.\mathsf{check})$. *The concrete data type* CPQ *conditionally implements an abstract data type PQ with the condition* ϕ. *Here* $cpq_F.\mathsf{ok}$ *denotes the value of the boolean ok of* cpq_F.

Proof. We need to provide an implementation function I, that meets the conditions of Definition 5. We obtain $I(pqp, s, ok)$ by collecting the elements of s into a priority queue as follows:
$I(pqp, \mathsf{create}_{LBS}, ok) = \mathsf{create}_{PQ}$
$I(pqp, s/\mathsf{assign}(p, q), ok) = I(pqp, s, ok)/\mathsf{insert}(p)$

The proof consists of validating that the embedding function I fulfills the requirements of Definition 5. We have completely verified the proof using the first order theorem prover Saturate. The details can be found in the next section.

□

4. The Formal Verification

The complete formal proof of Theorem 13 consists of 80 lemmas and theorems. For simplicity reasons we assume that in this section that variables which appear free in formulas are universally quantified. According to Definition 11, in Theorem 13 we have to confirm the correctness of the following equations:

$$I((pqp, s, ok)/\mathsf{insert}(p)) = I(pqp, s, ok)/\mathsf{insert}(p) \quad (7)$$

$$\begin{array}{l}(pqp, s, ok).\mathsf{contains}(p) \rightarrow \\ \quad I(pqp, s, ok).\mathsf{contains}(p) \wedge \\ I((pqp, s, ok)/\mathsf{remove}(p)) = I((pqp, s, ok))/\mathsf{remove}(p)\end{array} \quad (8)$$

$$\begin{array}{l}\exists p{:}P\ \ \Pi(I(pqp, s, ok), p) \rightarrow \\ \quad \phi((pqp, s, ok)/\mathsf{find\_min}) \rightarrow \\ \quad\quad \exists p_1{:}P\ \ \Pi(I(pqp, s, ok), p_1) \wedge \\ \quad I((pqp, s, ok)).\mathsf{find\_min}_{p_1} = (pqp, s, ok).\mathsf{find\_min}\end{array} \quad (9)$$

$$\begin{array}{l}\exists p{:}P\ \ \Pi(I(pqp, s, ok), p) \rightarrow \\ \quad \phi((pqp, s, ok)/\mathsf{find\_min}) \rightarrow \\ \quad\quad \exists p_1{:}P\ \ \Pi(I(pqp, s, ok), p_1) \wedge \\ I((pqp, s, ok))/\mathsf{remove\_min}_{p_1} = I((pqp, s, ok)/\mathsf{remove\_min})\end{array} \quad (10)$$

$$\begin{array}{l}\exists p{:}P\ \ \Pi(I(pqp, s, ok), p) \rightarrow \\ \quad \phi((pqp, s, ok)/\mathsf{find\_min}) \rightarrow \\ \quad\quad \exists p_1{:}P\ \ \Pi(I(pqp, s, ok), p_1) \wedge \\ I((pqp, s, ok)).\mathsf{remove\_min}_{p_1} = (pqp, s, ok).\mathsf{remove\_min}\end{array} \quad (11)$$

$$I(\mathsf{create}_{PQP}, \mathsf{create}_{LBS}, \mathsf{true}) = \mathsf{create}_{PQ} \quad (12)$$

Very often in automated first-order theorem proving we cannot prove theorems without interaction between the user and the prover. The theorem prover needs some advising which of the previously shown theorems and lemmas are useful in order to automatically prove the current theorem. In some cases we needed to add several additional intermediate lemmas before the proof of the current theorem could be completed.

Sometimes the reason for the insertion of new lemmas is strictly technical. Very often, the complete theory in which we have to prove the theorem, is too large for Saturate. It exceeded the maximal number of clauses and terminated without finding a proof.

Another problem we had to deal with were induction hypotheses. Saturate is a first-order theorem prover and therefore every induction hypothesis had to be generated by hand. Since most of our data structures are inductively defined, for every lemma about data structures which was proved directly, we had to state and prove induction basis and induction step as separate theorems.

Preconditions Π for formula (9) are defined as:

$$\begin{array}{l}\Pi(pq, p) = pq.\mathsf{contains}(p) \wedge \\ \quad \forall p'{:}P\ \ pq.\mathsf{contains}(p') \rightarrow p \leq p'\end{array}$$

Saturate needed some asistence in finding the proof for formula (9), namely we had to include as an intermediate lemma the fact that under the assumption that $\phi((pqp, s, ok)/\mathsf{find\_min})$ holds, the element returned by priority queue checker is contained in the priority queue and it is also the smallest element. Here we use the term "smallest" in the sense that the element belongs to the equivalence class of the smallest elements. Using the definition of Π for find_min, we have stated this as:

$$\begin{array}{l}\phi((pqp, s, ok)/\mathsf{find\_min}) \rightarrow \\ \quad \Pi(I(pqp, s, ok), (pqp, s, ok).\mathsf{find\_min})\end{array} \quad (13)$$

Formula (13) was proved by expanding the definition of Π, i.e. we have inserted the following intermediate lemmas:

$$\begin{array}{l}\phi((pqp, s, ok)/\mathsf{find\_min}) \rightarrow \\ I(pqp, s, ok).\mathsf{contains}((pqp, s, ok).\mathsf{find\_min})\end{array} \quad (14)$$

$$\phi((pqp, s, ok)/\mathsf{find\_min}) \rightarrow \\ \forall p'{:}P\ \ I(pqp, s, ok).\mathsf{contains}(p') \rightarrow \\ (pqp, s, ok).\mathsf{find\_min} \leq p' \tag{15}$$

Let us first explain the proof of the formula (14). Although we had to use more lemmas to prove it, the most important ones are:

$$\neg(pqp, s, ok).\mathsf{contains}((pqp, s, ok).\mathsf{find\_min}) \rightarrow \\ (pqp, s, ok)/\mathsf{find\_min} = (*, *, \mathsf{false}) \tag{16}$$

$$(pqp, s, \mathsf{false}) \rightarrow \neg\phi(pqp, s, \mathsf{false}) \tag{17}$$

Formula (16) is an easy consequence of the axioms specifying behaviour of find_min command, while formula (17) requires more complex proof. Saturate uses formula (18) and concludes that for any cpq, $(pqp, s, \mathsf{false}) \prec cpq$, $cpq.\mathsf{ok}$ does not hold, i.e. every successor of (pqp, s, false) has the form $(*, *, \mathsf{false})$ and therefore $\phi(pqp, s, \mathsf{false})$ also cannot hold.

$$\neg cpq.\mathsf{ok} \rightarrow \forall cpq'{:}CPQ\ \ cpq \prec cpq' \rightarrow \neg cpq'.\mathsf{ok} \tag{18}$$

In order to prove formula (18), we were using the induction principle used in Coq [13] for defining $\leq$ on the natural numbers. Let $\mathbb{N}$ be the set of natural numbers. Let s be the usual successor function. One can define $\leq$ as the smallest predicate which satisfies the following axioms:

1. $x \leq x$
2. $x \leq y \rightarrow x \leq s(y)$

The induction principle for some property P over $\leq$ is formalized as:

$$\forall x{:}\mathbb{N}\ \ \big(P(x)\ \wedge\ \forall y{:}\mathbb{N}\ \ x \leq y \rightarrow (P(y) \rightarrow P(s(y))) \rightarrow \\ \forall y{:}S\ \ x \leq y \rightarrow P(y)\big) \tag{19}$$

The predicate $\prec$ of Definition 12 has similar structure as $\leq$ on $\mathbb{N}$. Instead of one successor function, $\prec$ is defined by multiple successor functions. In order to verify correctness of formula (18), we define $P(x) = \neg x.\mathsf{ok}$ and verify

$$\neg cpq.\mathsf{ok} \rightarrow \neg(cpq/\mathsf{insert}(p)).\mathsf{ok} \tag{20}$$
$$\neg cpq.\mathsf{ok} \rightarrow \neg(cpq/\mathsf{remove}(p)).\mathsf{ok} \tag{21}$$
$$\neg cpq.\mathsf{ok} \rightarrow \neg(cpq/\mathsf{find\_min}).\mathsf{ok} \tag{22}$$
$$\neg cpq.\mathsf{ok} \rightarrow \neg(cpq/\mathsf{remove\_min}).\mathsf{ok} \tag{23}$$

More detailed proofs of the mentioned formulas are available in [12].

Let us now describe the proof of formula (15). It was proved by contraposition; together with some trivial facts we used the following lemma:

$$(pqp, s, ok).\mathsf{contains}(p)\ \wedge \\ p < (pqp, s, ok).\mathsf{find\_min} \rightarrow \\ \neg\phi((pqp, s, ok)/\mathsf{find\_min}) \tag{24}$$

In order to prove (24), we used the following reasoning: if there exists an element in the checked priority queue cpq which is strictly smaller then returned minimal element, then this element will have the lower bound greater than itself and $(cpq/\mathsf{find\_min}).\mathsf{check}$ cannot hold. But, if $(cpq/\mathsf{find\_min}).\mathsf{check}$ does not hold, then we know that for every checked priority queue cpq', $cpq \prec cpq'$, $cpq'.\mathsf{check}$ or $cpq'.\mathsf{ok}$ cannot hold and therefore $\phi(cpq/\mathsf{find\_min})$ does not hold. We formalized this reasoning by the following lemmas:

$$cpq.\mathsf{contains}(p)\ \ \wedge p < cpq.\mathsf{find\_min} \rightarrow \\ \neg(cpq/\mathsf{find\_min}).\mathsf{check} \tag{25}$$

$$\neg cpq.\mathsf{check} \rightarrow \forall cpq'{:}CPQ\ \ cpq \prec cpq' \rightarrow \\ (\neg cpq'.\mathsf{check}\ \vee\ \neg cpq'.\mathsf{ok}) \tag{26}$$

For the proof of formula (26), we again applied the principle of Coq induction, as described above and easily we have reduced formula (26) to the following formulas:

$$\neg cpq.\mathsf{check} \rightarrow \\ \neg s(cpq).\mathsf{check}\ \vee\ \neg s(cpq).\mathsf{ok}, \tag{27}$$

where $s(cpq)$ stands for the immediate successor functions cpq as defined in Definition 12. This time it was not a straightforward proof as in the case of formulas (20) - (23): formula (27) was proved using 23 additional lemmas.

For the proof of formula (25), Saturate needed some assistance in order to apply the following reasoning: let p^* be an element returned by priority queue checker as the minimal one. If there exists an element p, contained in the priority queue such that $p < p^*$, let q denote the lower bound of p. If $q < p^*$, then the lower bound of p in $cpq/\mathsf{find\_min}$ will be p^*. But if $p^* \leq q$, then the lower bound of p in $cpq/\mathsf{find\_min}$ will not change. Which means that in $cpq/\mathsf{find\_min}$ there is an element whose lower bound is strictly bigger than the element itself. Using this reasoning, formalized as a theorem, and the check-characterization lemma (28), formula (25) was successfully verified.

$$(pqp, s, ok).\mathsf{check} \Leftrightarrow \\ \forall p{:}P\ \ \forall q{:}P\ \ s.\mathsf{contains\_pair}(p, q) \rightarrow q \leq p \tag{28}$$

Formula (28) is implicitly universally quantified, therefore we need to verify that it holds for every pqp, s and

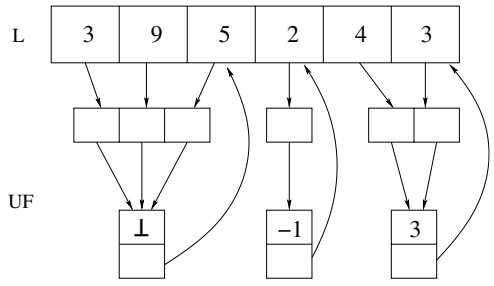
ok. Since the system of lower bounds is inductively defined with create and assign, we have generated by hand the induction basis and the induction step for formula (28) and using Saturate we have verified both.

We stop now with the proof for formula (9). Although the complete proofs are not given here, they can be found in [12], together with the proof schemes for all other formulas.

5. Ongoing and Future Work

In this section, we briefly describe the actual implementation of the lower bound system in LEDA. We intend to refine our model, so that we can verify the real implementation. In order to obtain a superefficient checker, i.e. a checker that is more efficient than the priority queue itself, the checker needs to be implemented in the way that is drawn in Figure 3. Structure L is a C^{++}-style linked list. A linked list is a container which has iterators that support efficient insertion and deletion, and traversal of lists, but no efficient indexing. There is also no efficient lookup of elements. The priority queue PQ over P has to be replaced by a priority queue over $P \times I$, where I is the type of iterators over L. In addition to L one needs a union find data structure which maintains equivalence classes of elements that have the same lower bound. The indices to the union find are stored in L. Each block of the union find again contains a pair of type $P \times I$. The first element represents the lower bound, and the second element is an iterator of L, that points to the last element in the equivalence class. Figure 3 shows a possible representation of the lower bound system $((3, \perp), (9, \perp), (5, \perp), (2, -1), (4, 3), (3, 3))$. Figure 3 is not completely accurate because in real the numbers are stored in the priority queue, and only union find indices are stored in L. It can be checked, see [8] that, using this representation, all operations of Figure 2 take constant amortized time.

Figure 3. Implementation of a lower bound system



In order to verify the improved implementation, we will need specifications of union find and of lists with iterators.

6. Conclusions

The goal of this paper was two-fold: the first goal was to formally verify the algorithm used for the priority queue checker. The second goal was to determine what can be the role of saturation-based theorem proving in verification. The first goal was fulfilled successfully; we managed to develop the specification and using this specification, we were able to prove the correctness of the checker. As for the second goal, we have learnt that saturation-based theorem proving is useful for verification, but still Saturate needed a lot of guidance and all induction hypotheses had to be generated by hand. In order to assess the usefulness of first-order theorem proving, a possibility would be to reproduce the verification in an interactive verification system, as for example PVS, and compare the amount of interaction needed.

References

[1] N. M. Amato and M. C. Loui. Checking linked data structures. In *Proceedings of The 24th International Symposium on Fault Tolerant Computing (FTCS-24)*, pages 164–73, 1994.

[2] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[3] M. Blum and S. Kannan. Designing programs that check their work. In *Proceedings of the 21st annual ACM symposium on Theory of computing*, pages 86–97. ACM Press, 1989.

[4] O.-J. Dahl. *Verifiable Programming*. Prentice Hall, 1992.

[5] W.-P. de Roever and K. Engelhardt. *Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[6] U. Finkler and K. Mehlhorn. Checking priority queues. In *Proceedings of the 10th annual ACM-SIAM symposium on Discrete algorithms (SODA'99)*, pages 901–902. Society for Industrial and Applied Mathematics, 1999.

[7] E. Landau. *Grundlagen der Analysis, (das Rechnen mit ganzen, rationalen, irrationalen und komplexen Zahlen)*. Akademischer Verlagsgesellschaft M.B.H, 1930.

[8] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[9] T. Nipkow. Non-deterministic data types: Models and implementations. *Acta Informatica*, 22:629–661, 1986.

[10] B. Stroustrup. *The C^{++} Programming Language, third edition*. Addison-Wesley, 1997.

[11] `http://www.mpi-inf.mpg.de/SATURATE/`.

[12] `http://www.mpi-inf.mpg.de/~rpiskac/queues/`.

[13] `http://coq.inria.fr/`.

[14] `http://www.algorithmic-solutions.com/`.