SpecVer & LEM'12 projects XX (2015) 1–20
Dombrova Research

Verifying a Class: combining Testing and Proving

Grażyna Mirkowska
Faculty of Mathematics and Natural Sciences
University Cardinal Stefan Wyszyński
Wóycickiego 1/3, 01-938 Warszawa, Poland
mirkowska@uksw.edu.pl

Oskar Świda
Białystok University of Technology
Department of Computer Science
Wiejska 45A, 15-351 Białystok, Poland
Oskar.Swida@gmail.com

Andrzej Salwicki
National Institute of Telecomunication
Szachowa 1, 04-894 Warszawa, Poland
salwicki@mimuw.edu.pl
and
Faculty of Mathematics and Natural Sciences
University Cardinal Stefan Wyszyński
Wóycickiego 1/3, 01-938 Warszawa, Poland

Abstract. The problem of correctness of a class C w.r.t. a specification S is discussed. A formal counterpart of the problem is the question well known in metamathematics, whether an algebraic structure is a model of a given theory. Now, this metamathematical problem has to be adapted to the context of software engineering. As a theory we consider the (algorithmic) specification S. The algebraic structure $\mathbb{A}_C$ induced by the class C is our candidate for a model of S. Remark, that this problem differs from the correctness' problem of an algorithm w.r.t. a pre- and a post-conditions. In the paper we consider the specification $ATPQ$ of priority queues and the class PQS, and we verify the correctness of this class with respect to the specification $ATPQ$.

Programmers and software companies prefer to test software instead of proving it. Surely, proving is more difficult, testing is easier. In this article we combine these two approaches. Hence, the following actions appear in our method of verification: experiment, observe, formulate hypotheses, prove. It is our hope that this method is of general use and adapts well to many practical cases of verification of object-oriented software.

1. Introduction

This paper presents a proof that the class PQS is a correct implementation of priority queues data structure. The class (see Appendix) is written in an object-oriented programming language. The specification

(see Table 1) consists of a few algorithmic formulas. Formally, the proof of correctnes of the class w.r.t. the specification resembles a proof that a given algebraic structure $\mathbb{A}$ is a $model$ of some axiomatic theory $\mathcal{T}$. Literature knows many examples of the proofs of correctnes of algorithms with respect to their pre- and post-conditions. However, the union of proofs of a class' methods (i.e. algorithms) does not result in the proof of the class correctness.

Each class can be considered as a description of an algebraic structure (a data structure). The universe of the structure is the set of all potential objects of the class, its functions and relations are defined by class methods.

Dealing with a class we are confronted with several properties, sometimes called invariants of the class (B. Meyer in Eiffel [4]), that must be valid for all objects of the class. Moreover, an external characterization of the class requires some properties that express corellations between different methods. All these properties will be called a specification of a class.

Our message has several layers:

- first, we offer a discussion of the methods of software's verification.
- second, we propose to think what a software's verification is,
- finally, we need to know what a class specification looks like?

Two views and two practices are colliding in production of software. One view is that programs should be accompanied by solid arguments demonstrating the correctness and the completeness of software. The practice associated with this view consist in proving properties of software. At present, we observe that there are more and more cases when the industry demands the proofs of correctness of programs. It is especially true of these cases where security must be guaranteed. Another view is followed by the majority of the individual programmers and the big software companies. They are of the opinion that testing of programs is the sufficient activity before the software is delivered to clients. Hence we have two practices that seem to exclude one another: testing or proving. In the presented paper we argue[1] that the two approaches may be synthesized to a completely different scheme of practice. We propose to replace the term *testing* by another word *experimenting*. Testing limits itself to the execution of program and the comparison of its effects with the predicted, supposedly correct results. For the majority of people testing is the practice of searching bugs in software. Experimenting has larger horizons, it covers not only searching of errors (aka counter-examples) but also gathering the positive evidence. Sometimes during experiments we begin to believe that objects obey certain rules and later we try to find more evidence confirming our beliefs. During experiments one is going to execute program with different data, to collect results and to present them in graphical mode, in tables, in data bases, etc. It is suggested that the experiments were done with some plan. Next, one should analyze the gathered experience and search some regularities. This process should lead to the formulation of lemmas and propositions. After this is done, there is the time of proving the hypotheses. Obviously, the process sketched above may need to be iterated for different reasons, e.g. when our program is modified. We consider the method given below

$$experiment \rightarrow observe \rightarrow formulate\ hypotheses \rightarrow prove.$$

[1] following to some extent the ideas of Georg Hegel who used to say that from a pair: thesis and antithesis we should make a synthesis

as a proper approach to the production of the high integrity software [6], [2], [3], [1].

How to define the goals of software's verification? Assuming that a specification of a class is given in the form of a set of formulas, the verifier agent should prove that every object of the analyzed class will satisfy every formula of the set, whenever any method of the class has been completed.

We are proposing the methodology which consists of algorithmic logic AL and an environment SpecVer – a plugin into the Eclipse IDE. A SpecVer project can be developed from its initial phase of specifying algorithms and classes, through implementing them in an object programming language, to the phase of verification of implementation against its specification.

We present an almost complete case study which consists of a specification $ATPQ$ of priority queues, a class `PQS`, and the verification of the thesis that the class `PQS` correctly implements the specification $ATPQ$.

Class PQS may be used in several applications. For example, it is a part of a bigger program Simulation of a Bank Department. The structure of the simulation program is shown in Figure 1. The program contains 6 external classes, 13 innner classes and 20 methods. The relation between inner classes and the containing them external classes are shown on the diagram. The relation of inheritance is also shown on the diagram. The arrows lead from one class to its direct superclass. The thick arrows start at external classes. The thin arrows lead from an inner class K to inner class I inherited by the class K. We conceive the five external classes as implementations of data structures: Class `BankDepartment` extends the structure of `Office`. Class `Office` is based on the class `Simulation`, it uses also the data structure of `FIFOQueues`. The class `Simulation` relies on the class `PriorityQueues`. These relations are shown on the diagram by thick arrows. (As no one class may inherit two classes, we decided to make the class `FIFOQueues` the base class of the class `PriorityQueues`.) The diagram shows also the inheritance relation between inner classes. This allows us to introduce more subtle relations between classes. For example, any object of class `Customer` is queueable since the class `Customer` inherits from the class `Simprocess` which in turn inherits from the class `ElemFIFO`. An object of class `Customer` may be activated and made passive several times since the class `SimProcess` is a coroutine. The value of unique variable `ExperimPlan` is a set of `EventNotices`. During the execution of our simulation experiment the variable `ExperimPlan` has various sets of `EventNotices` as its value. An object of the class `EventNotice` is a pair: $\langle s.t \rangle$, where s is a `SimProcess` object and t is a time. Objects of class `EventNotice` are inserted and/or deleted from the set `ExperimPlan`. `EventNotices` are ordered by a relation $less$. The class `Simulation` is to guarantee that the active object of the experiment, in our case it will be either a customer or a teller object, will be active iff its activation time is the minimum of all times of `EventNotices` in `ExperimPlan` set. Encapsulating two inner classes `SimProcess` and `EventNotice` and the variable `ExperimPlan` of type `PQ` makes the process of coding of the class `Simulation` simpler.

The term high integrity software was introduced in [2]. For us the high integrity programming means the activity which involves specification of software modules, implementation the modules (i.e. classes and methods) and verification of the modules against their specification. We shall use the structure shown in Figure 1 to illustrate what has to be done.
For each external class C three documents should be produced:

- A specification S of the class C. Specification is a set of formulas which express the properties of methods and invariants of objects of the class. Each specification should enjoy two properties:
  - Consistency

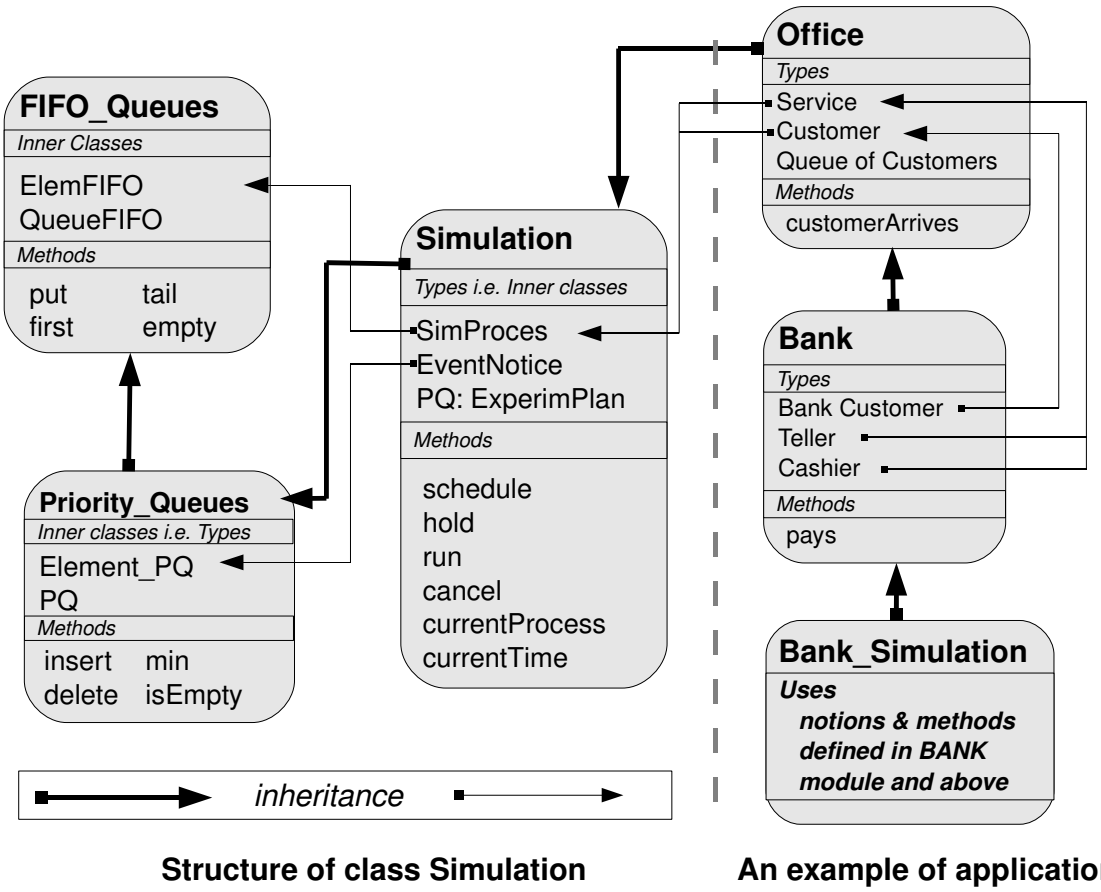

Figure 1. Modules of bank simulation program

There is no implementation of an inconsistent specification. An inconsistent specification has no sense.

– Completeness
An incomplete specification allows various models. Not all of them are desired. A complete specification brings enough information on properties of objects of class to distinguish between a desired and an improper implementation, and therefore can be used as a criterion of acceptance of a class. It is sufficient to produce the proofs or verification reports.

- The file containing the class C itself. Usually this file contains all the methods and inner classes of the class C.

- The verification report. This file should contain arguments that soothe our conscience and convince the user of our class. The arguments used may be more formal - having form of a mathematical proof or may recall a dialogue. Rarely a formal proof is needed. The verification report should be rather an evidence of analysis and it should serve to convince its reader that the conclusions of

the report are sound. A verification report is of good quality if it is an intersubjective experience. Surely, a formal proof of a correctnes has this quality, but frequently it is not readable by a human being. A balance between two extremities is needed.

In earlier papers we have presented the work on specifications of classes [6]. Specifications are subjects of studies and analyses. In other words, one should visualize a process of development and amelioration of a specification.

This paper illustrates the process of verification of a class C against a specification S. In the most cases implementation of a class precedes the process of verification. Sometimes, the verification can be done simultaneously with the production of the needed class. In another paper we shall exemplify the (rare) case when a class, the Simulation class, is systematically elaborated together with the proof of its correctness. For this we need only the specification of the base class `PQS` and the target class `Simulation`.

We propose to make an experiment. The reader will look at the appendix and try to give arguments that the class PQS correctly implements a priority queues system. Those who forgot what a priority queue is, may find its axiomatic definition in Table 1 below. We ask the reader how much of time he/she needs to convince someone that the class `PQS` implements the abstract data type of priority queues.
We did the following:

1. Experiments - we executed methods of the class by hand and have drawn some pictures.
2. Observations - we analyzed the pictures and searched for some regularities.
3. Conclusions - we formulated several hypotheses (lemmas) and propositions.
4. Proving - we proved the lemmas.
5. Finally - putting together the facts, we proved the correctness theorem.

2. Priority Queues Specification

Before implementing a data structure one should write down its specification. The first part of the specification – the signature – enlists the sorts, the operations and the relations. The second part enlists the properties, also called the axioms, Table 1 contains the specification of the abstract data type of priority queues [5, p.154]. Remark that besides the formulas of first-order logic we use algorithmic formulas [5]. An example of algorithmic formula is the axiom (a2). Its structure is as follow:

$$\langle program\ P\rangle\langle formula\ \alpha\rangle.$$

The meaning of the whole formula is *"after execution of program P the formula α is valid"*. In our example the formula (a2) takes value true if and only if the program halts. Hence, including such formula among the axioms of our specification, means *"the program P always terminates"*. The semantic meaning of the axiom (a2) is *the set q is finite*. The axiom (a8) is in fact an explicit, *algorithmic* definition of the relation $member$.

Algorithmic formulas allow to express the semantic properties of programs such as termination, correctness, etc. Almost 40 years ago we proved that the calculus is sound and complete [5]. It means

that we have choice between showing that some semantic property of a program is valid or proving the formula that expresses the property. We gave several examples of specification of abstract data types as algorithmic theories. $ATPQ$ is one of such examples.

Table 1. Specification $ATPQ$ of priority queues.

Signature	Comments
Sorts	$Universe = E \cup PQ$
E	set of elements
PQ	set of priority queues
Operations	let $e \in E$ and $q \in PQ$
$insert : E \times PQ \longrightarrow PQ$	put e into q
$delete : E \times PQ \longrightarrow PQ$	delete e from q
$min : PQ \longrightarrow E$	find the minimum element
$empty : PQ \longrightarrow \{true, false\}$	is a priority queue q empty?
$member : E \times PQ \longrightarrow \{true, false\}$	does $e \in q$?
$\leq : E \times E \longrightarrow \{true, false\}$	the ordering relation

Axioms

(a1) $The\ set\ E\ of elements\ is\ linearly\ ordered\ by\ the\ relation \leq .$

(a2) [**while** $not\ empty(q)$ **do** $q := delete(min(q), q)$ **done**] $true$

This axiom says for all q program halts, i.e. *the priority queue q is finite*

(a3) $[q1 := insert(e, q)]\{member(e, q1) \wedge (\forall_{e1 \neq e}\ member(e1, q1) \Leftrightarrow member(e1, q))\}$

(a4) $[q1 := delete(e, q)]\{\neg member(e, q1) \wedge (\forall_{e1 \neq e}\ member(e1, q1) \Leftrightarrow member(e1, q))\}$

(a5) $empty(q) \Rightarrow (\forall_{e \in E}\ \neg member(e, q))$

(a6) $\neg empty(q) \Rightarrow (\forall_{e \in E}\ member(e, q) \Rightarrow min(q) \leq e))$

The operation min finds the least element of the set q.

(a7) $[e := min(q)]true \Leftrightarrow \neg empty(q)$

Axiom (a7) says the result of expression $min(q)$ is defined iff $\neg empty(q)$

(a8) $member(e, q) \Leftrightarrow$ **begin**

s1 := q; result := false;
while not *empty(s1)* and not *result* **do**
if *e =min(s1)* **then** *result*:=true **fi**;
s1 := delete(min(s1),s1)
done
end *result*

$ATPQ$ is an acronym of *Algorithmic Theory of Priority Queues*. The theorems of the theory are the formulas provable by the calculus of algorithmic logic [5] from the axioms of $ATPQ$.

Remark 2.1. In the literature, the frequent choice when speaking on the abstract data type of priority queues, is the operation deletemin, instead of the operation delete. Our choice was different and deliberate. With the present set of operations we are able to construct the specification which enjoys the property of being almost complete.

It was shown in [5] that the algorithmic theory of priority queues has a metalogical property known as the *representation* meta-*theorem*: Every model of $ATPQ$ is isomorphic to the standard model of priority queues. [2]
Let us add a few words on the $ATPQ$ specification. An important fact states that any implementation of the axioms of $ATPQ$ where a concrete set E was given, is isomorphic to the structure of finite subsets of the set E with the operations

$$insert(e, s) = \text{increase the set } s \text{ by adding element } e \text{ to it,}$$

and

$$delete(e, s) = \text{supprime the element } e \text{ from the set } s.$$

The consequences of the theorem are of general methodological nature:

- the specification of $ATPQ$ is complete. If a new formula is added then either it is a logical consequence of the axioms or it leads to contradiction, or it expresses the properties of the elements only.
- the specification can be used as the criterion of correctness of a proposed implementation,
- the proofs of properties of programs that use this data structure may be based on axioms of priority queues listed in Table 1. No other properties of priority queues are ever needed.

3. Experiments

Our work on verification of the class PQS began by experimenting. The experiments consisted in executing (by hand) operations insert and delete, drawing pictures, and observing the changes in the constellation of objects of type node. Figure 2 shows a few snapshots of our experimentation. In fact, we did twice as much drawings. The reader may wish to continue our experiments on his own, e.g. by inserting the element $e6$ or deleting the element $e2$ after insertion of the element $e5$ was done. It is wortwhile to observe that the snapshot after execution of $delete(e2)$ will show the picture which is essentially the same as after insertion $e1, e2, e3, e4$ with following difference: Instead of $e2$ we find $e5$, and the pair of objects $\langle\ e2$, its companion node object $\rangle$ is an isolated (not connected) part of the graf. During the experiments we neglected the ordering relation between elements.

Our first impression, when looking at the drawings of Figure 2, is a complete chaos. Slowly we commence to distinguish some parts and we begin to perceive some regularities. First, we remark that in each of 6 pictures there is exactly one object of type `PQ` which has two fields: $root$, $last$. Next, we remark that objects of type `Elem` and of type `Node` come in pairs, each pair is connected by $.el$ and $.lab$ arrows. Our next observation is that the arrows $.up$ form lists of objects of type `Node` (no cycles). The

[2] Algorithmic theories (e.g. [9], [5]) were studied since 70's of XX century.

meaning of arrows $.left$ and $.right$ is less evident that it may be suggested by their names. On the diagram some of them are solid and some of them are dotted. This comes as the result of analysis, the arrows $.left$ and $.right$ pay two roles. We see that the solid arrows $.left$ and $.right$ go against to the arrows $.up$. While the observation that there is no cycle in paths composed of $.up$ arrows may lead to the statement that on each diagram we have a tree of node objects, the present observation states that the tree is a binary one.
Our observations need to be properly formulated and proved. This will be done in the next section. Before that, one may execute further experiments, e.g. by executing the command $delete(e2)$.

4. Observations and Lemmas

We shall study the class `PQS`, see the Appendix. We can assume that a usage of `PQS` consists in a finite sequence of creation of `newElem`() objects and calls: `call q.insert(e), call q.delete(e')`.
Following the intuition gained from Figure 2 we shall introduce the notion of observable states. The initial state s_0 is the graph consisting of exactly one object o of type `PQ`, $s_0 = \{new\ \texttt{PQ}\}$ and no edges.

Definition 4.1. (of the observable states) The set $\mathcal{S}$ of observable states is the least set which contains the initial state s_0 and which is closed with respect to the operations insert and delete and creation of `new Elem`() objects.

Each state consists of a set of objects and the edges connecting them. The examples of states are presented in Figure 2. The class `PQS` may be viewed as a definition of the relational structure PQS. The universe U of the structure consists of the objects of the inner classes of the class `PQS`. The attributes of objects of U define functions between objects. The set of objects of type `Node` will be denoted NODE. analogously for the set of objects of type `Elem` and of type `PQ`:

$$\begin{aligned} \text{NODE} &= \{n :\ n \text{ instanceof } \texttt{Node}\}, \\ \text{ELEM} &= \{e :\ e \text{ instanceof } \texttt{Elem}\}, \\ \text{PQ} &= \{q :\ q \text{ instanceof } \texttt{PQ}\}. \end{aligned}$$

Definition 4.2. The class `PQS` determines an algebraic structure

$$\text{PQS} = \langle U, .up, .left, .right, .el, .lab. \rangle,$$

where U is a subset of the union NODE $\cup$ ELEM $\cup$ PQ which satisfies the condition

$$(\forall_{n \in \text{NODE}})\ (\forall_{e \in \text{ELEM}})\ n.el = e \Leftrightarrow e.lab = n.$$

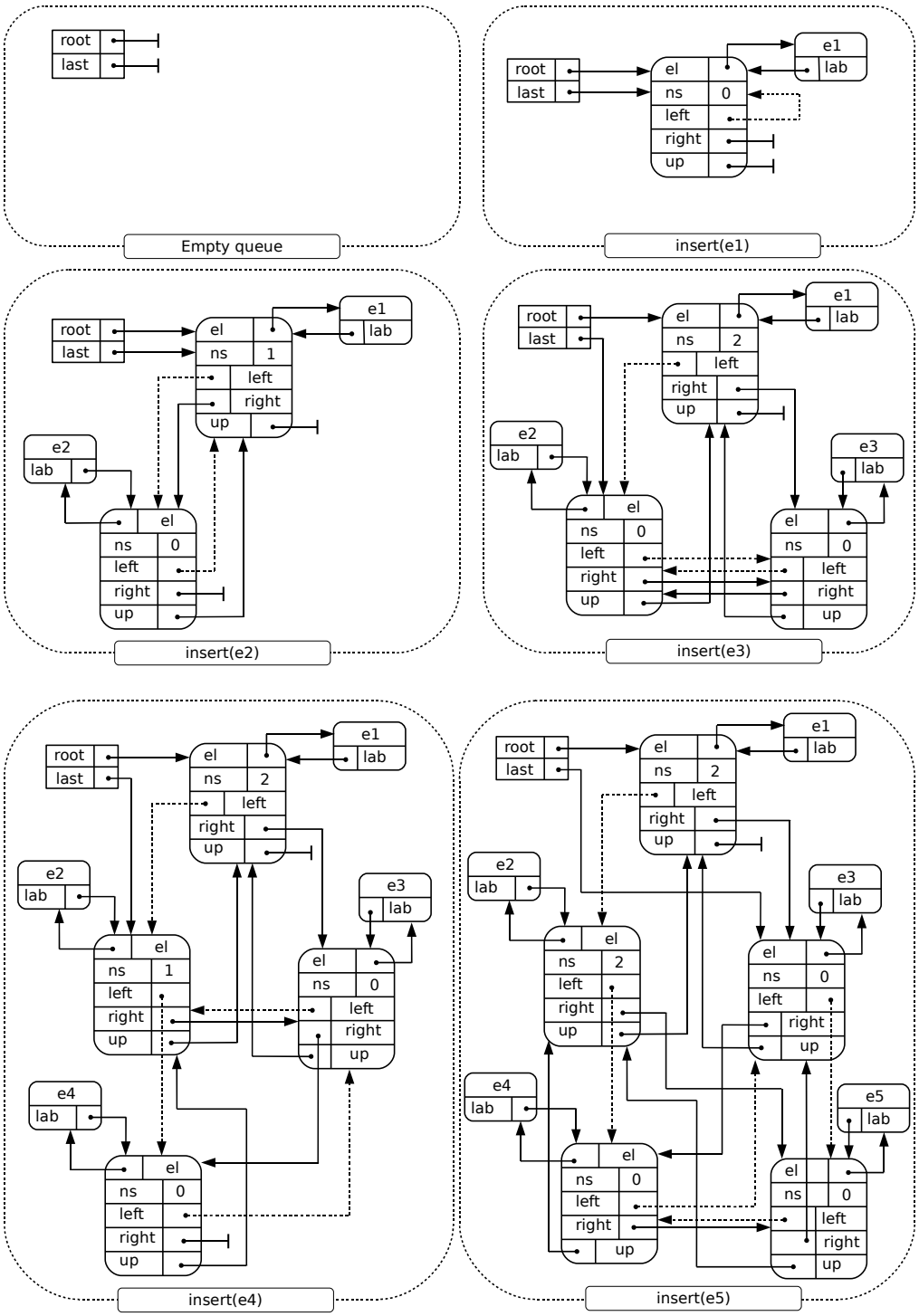

Figure 2. Inserting elements e1 - e5

The functions of the structure PQS are defined as follows

$$\begin{aligned}
.up &\stackrel{df}{=} \{\langle n, n'\rangle : \ n, n' \in \text{NODE}, \ n' = n.up\},\\
.left &\stackrel{df}{=} \{\langle n, n'\rangle : \ n, n' \in \text{NODE}, \ n' = n.left\},\\
.right &\stackrel{df}{=} \{\langle n, n'\rangle : \ n, n' \in \text{NODE}, \ n' = n.right\},\\
.el &\stackrel{df}{=} \{\langle n, e\rangle : \ n \in \text{NODE}, e \in \text{ELEM}, \ e = n.el\},\\
.lab &\stackrel{df}{=} \{\langle e, n\rangle : \ e \in \text{ELEM}, n \in \text{NODE}, \ n = e.lab\}.
\end{aligned}$$

Our first observation is that if we abstract from (i.e. we forget about) the arrows $.left$ and $.right$ then for each state s its graph shows a tree.

Definition 4.3. Let s be an observable state, consider the objects of type `Node` in this state. Let T_s be the set of these object of type `Node` that access the object $.root$ object by a path composed from $.up$ arrows only.
$T_s = \{o \in \text{NODE} \cap s : there\ is\ a\ path\ composed\ from$
$.up\ arrows\ only,\ leading\ from\ object\ o\ to\ object\ .root\}$

Lemma 4.1. In any observable state s the pair $\langle T_s, .up\rangle$ is a tree.

Definition 4.4. (of a son) We say that a node n is a *son* of a node f in the tree T_s if and only if $n.up = f$. A node n is said to be a *leaf* of the tree T_s if and only if it has no sons.

Our next observation is

Proposition 4.1. For every $o \in T_s$, $o.ns$ = number of sons of o.

Definition 4.5. We say that an arrow $.left$ from the node n is *solid* iff $n.ns > 0$. We say an arrow $.right$ from the node n is *solid* iff $n.ns = 2$. Otherwise the arrows are said *weak*, or dotted.

Next, we observe that solid arrows lead against $.up$ arrows.

Proposition 4.2. For every state s, the tree T_s with solid arrows only, forms a binary tree.

Proof: For every two nodes n and f of the tree T_s the following properties hold:

- if a solid .left arrow leads from node f to node n then $n.up = f$
  $(f.left = n \wedge f.ns > 0\ ) \Rightarrow n.up = f$,
- if a solid .right arrow leads from node f to node n then $n.up = f$
  $(f.right = n \wedge f.ns = 2\ \wedge) \Rightarrow n.up = f$,
- $n.up = f \Leftrightarrow (f.left = n \vee f.right = n)$.

Hence T_s is a *binary* tree. Our next observation can be stated as follows:

Proposition 4.3. There exists at most one node n in T_s such that $n.ns = 1$. If it is the case then $last = n$.

Now, we observe that the leaves of tree T_s are on two levels only.

Proposition 4.4. For every state s, there exists a natural number $k(T_s)$ such that every leaf of the tree T_s is on the level $k(T_s)$ or $k(T_s) - 1$.

The number $k(T_s)$ is equal the length of the path composed from $.up$ arrows leading from the object $last.left$ to the $root$ object. It is equal 0 if $root = none$. □

Now we try to guess the rôle of non-solid arrows $.left$ and $.right$. The following property holds:

Proposition 4.5. The object referenced by the variable $last$ in the tree T_s is the leftmost node on the level $k(T_s) - 1$ which has less than two sons.

Let us return to the Figure 1 and observe the following facts:

Remark 4.1. A) If a node n has two sons then its left brother has also two sons.
B) If a node n has one son then it is its left son.
C) If a node n has one son then its brother from the left has two sons and its brother from the right is a leaf.

Proposition 4.6. The value of the variable $last$ is a head of a list of leaves linked together via $.right$ (weak) arrows.

Proposition 4.7. The value of the variable $last$ is a head of a cyclic list of leaves linked by (weak) $.left$ arrow.

We see that all leaves on the level $k(T_s)$ are grouped to the left.

Proposition 4.8. Tree T_s is a *perfect binary tree* i.e. all the levels are completely filled with an eventual exception on the deepest level, in this case all the leaves are grouped to the left.

The following four lemmas have similar form $(\alpha \Rightarrow I\beta)$, where I is either instruction $insert(e)$ or instruction $delete(e)$, α is a precondition and β is a postcondition of the instruction I.

Lemma 4.2. Let $I : insert(e)$ and
$\alpha_1 : \{last = o \wedge o.left = n1 \wedge o.right = k \wedge o.ns = 0 \wedge e.lab = n \wedge n.el = e\}$,
$\beta_1 : \{last = o \wedge o.ns = 1 \wedge o.left = n \wedge o.right = k \wedge n.up = o \wedge n.left = n1\}$.
The instruction I is correct with respect to the precondition α_1 and postcondition β_1 in the structure PQS

$$\text{PQS} \models (\alpha_1 \Rightarrow I\beta_1).$$

Proof:
How to read this lemma? It states that in any state s, if the precondition α_1 is satisfied by s, then the execution of instruction $insert(e)$ will succesfully lead to certain state s' and the postcondition β_1 will be satisfied by s'. What is the meaning of the precondition α_1 of the instruction *insert*? We can draw it, see Fig. 3.

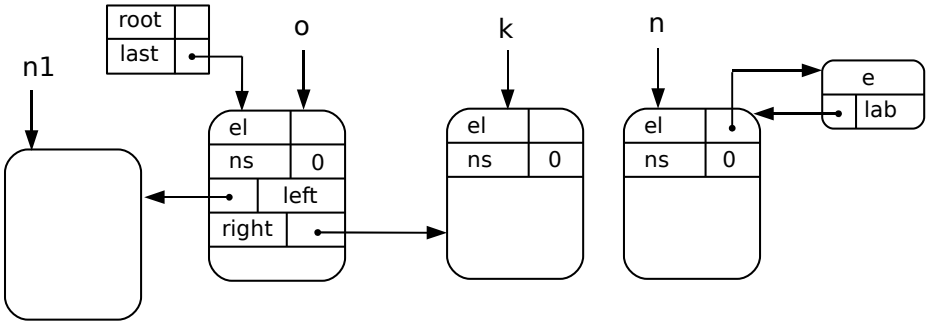

Fig. 3 Precondition α_1.

We check that the configuration of objects drawn on Fig. 3 satisfies the precondition α_1, The equality $last = o$ is satisfied since both variables point to the same object. The variable $last.left$ points to the object pointed by $n1$. $last.right$ point to the object pointed by k. The objects e and n are linked together, $e.lab = n$ and $n.el = e$.

Next, we can follow step by step the execution of the command $q.insert(e)$ with the text of method $insert$ in hand. We start observing that the precondition α implies $last.ns = 0$ hence the instructions executed by $insert$ are:

x:=e.lab; last.ns := 1; z := last.left; last.left := x; x.up := last;

x.left := z; z.right := x;last:=z;

Now we can draw modifications to the picture following the instructions.

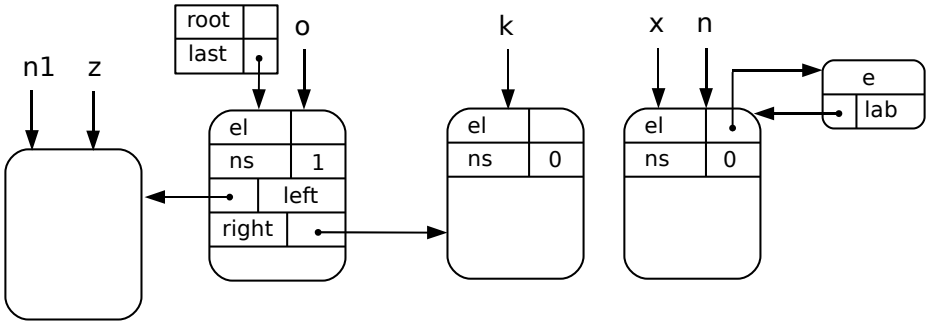

Fig. 4 Snapshot after 3 instructions of insert's body

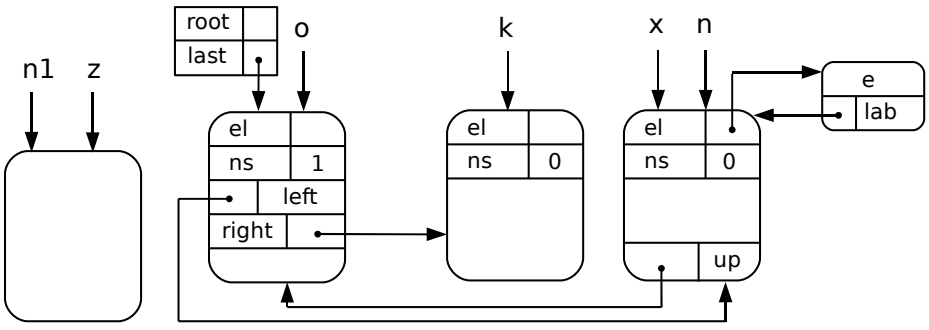

Fig. 5 Snapshot after 5 instructions of insert's body

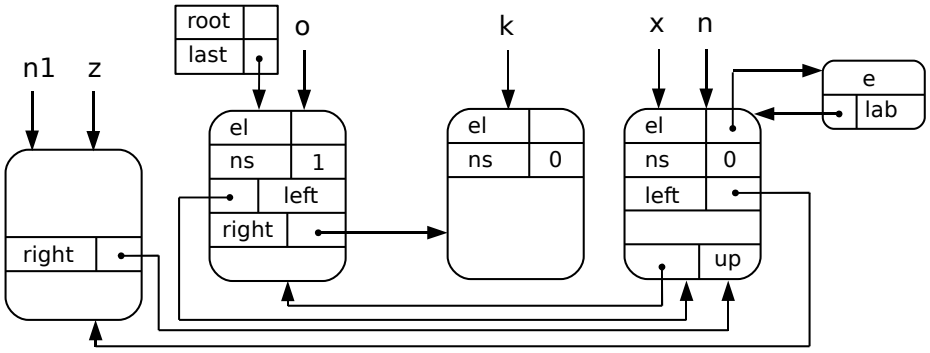

Fig. 6 Snapshot after completion of instruction insert.

The state shown on Fig. 6 can be described by the following formula $\gamma : \{last = o \wedge o.ns = 1 \wedge o.left = n = n1.right = e.lab \wedge o.right = k \wedge n.up = o \wedge n.left = n1 \wedge n.el = e\}$. It is easy to observe that the formula γ implies the formula β_1. Note that if the initial state s satisfies the precondition α_1 then at the end of the execution of the instruction $insert(e)$ in PQS we obtain the state s' satisfying the postcondition β_1.

Lemma 4.3. Let $I : insert(e)$ and
α_2:{last=o∧o.left=n1∧o.right=n2∧o.ns=1∧e.lab=n},
β_2: {last=n2∧o.ns=2∧o.left=n1∧o.right=n∧n.up=o∧n.left=n1 ∧n1.right=n}.
The instruction $I : insert(e)$ is correct with respect to the precondition α_2 and the postcondition β_2.

$$\text{PQS} \models (\alpha_2 \Rightarrow I\beta_2).$$

Proof:
One can easily verify that if $last.ns = 1$ then the actions executed by the method $insert$ are those presented in Table 2 below.

Table 2. Proof of lemma 3.

Precondition: α_2:(last=o∧o.left=n1∧o.right=n2∧o.ns=1 ∧e.lab = n)	
Instruction	*Effect*
x := e.lab	x=n, *since precondition says* e.lab=n
last.ns:=2	o.ns=2, *since* last=o
z := last.right	z=n2, *because* last.right=n2
last.right:=x	o.right=n, *because* last=o and x=n
x.right := z	n.right=n2, *because* x=n
x.up:= last	n.up=o, *because* last=o and x=n
z.left:=x	n2.left=n, *because* z=n2
last.left.right:=x	n1.right=n, *because* last.left=n1
x.left:=last.left	n.left =n1, *because* last.left=n1 and x=n
last :=z	last=n2, *because* z=n2
Postcondition: β':(o.ns=2∧o.right=n∧n.right=n2∧n.up=o∧ n2.left=n∧n1.right=n∧ n.left=n1∧last=n2∧o.left=n1∧e.lab=n)	

The postcondition β' collects the facts enlisted in the column Effect extended by the formulas e.lab=n and o.left=n1 for they remain satisfied after execution of $insert$. In this way we obtained a postcondition which is even stronger than the formula β_2.

The proofs of lemmas 4.2 and 4.3 exemplify two different ways of argumenting that a semantical property is valid. The first one, informal, consists in drawing the pictures. It may be related to drawing Venn's diagrams in the algebra of sets. Like diagrams of Venn it does not replace the proving but it is helpful. The second one is nearer to the goal of mechanization of proving.
In the following two lemmas we analyze properties of algorithm $delete$.

Lemma 4.4. Let the formulas $\alpha_3,\ \beta_3$, and the instruction D be defined as follows:
$\alpha_3 : \{last = o \wedge o.left = n1 \wedge o.right = n2 \wedge o.ns = 0 \wedge e.lab = n \wedge n1.left = n3\}$,
$\beta_3 : \{last = n1.up \wedge last.right = o \wedge last.ns = 1 \wedge o.left = last \wedge o.right = n2 \wedge\ \ n1.ns = 0 \wedge n1.up = n1.left = n1.right = none \wedge n3.right = none \wedge n1.el = e\}$,
$D : delete(e)$.
The instruction $delete(e)$ is correct with respect to conditions α_3 and β_3, i.e.

$$\text{PQS} \models (\alpha_3 \Rightarrow D\beta_3).$$

Now we consider another case of applying the instruction $delete$.

Lemma 4.5. Let the formulas $\alpha_4,\ \beta_4$ be defined as follows:
$\alpha_4 : \{last = o \wedge o.left = n1 \wedge o.right = n2 \wedge o.ns = 1 \wedge e.lab = n \wedge n1.left = n3\}$,
$\beta_4 : \{last = o \wedge o.left = n3 \wedge o.ns = 0 \wedge o.right = n2 \wedge n1.el = e \wedge n1.up = \quad n1.left =$

$n1.right = none \wedge n3.right = none\}$.
The instruction $delete(e)$ is correct with respect to conditions α_4 and β_4, i.e.

$$\text{PQS} \models (\alpha_4 \Rightarrow D\beta_4).$$

The instructions call correctUp() and call correctDown(), that end the execution of procedures insert and delete, serve to guarantee that for each path in the tree T of nodes the elements associated to the nodes of the path form a decreasing sequence with the minimum in the root.

Lemma 4.6. (on procedure correctUp)
Procedure instruction *call correctUp(r)* is correct w.r.t. the precondition γ_1 and the postcondition γ_2 given below
$\gamma_1 : r$ **in** $elem \wedge r.less(r.lab.up.el) \wedge last.left.el = r$
(The first condition r **in** $elem$ is checked by compiler.) The second condition says the newly added element is less or equal than the element associated with the father of r. The third condition says: the companion node n of the element r is pointed by the pointer $last.left$.
γ_2 :for every node n on the path beginning at the element r, the following condition holds $n.up.less(n) \vee n.up = none$.

Lemma 4.7. (on procedure correctDown)
Procedure *correctDown* is correct w.r.t. the precondition γ_3 and the postcondition γ_4 given below
$\gamma_3 : r$ **in** $elem \wedge \neg r.less(r.lab.up.el$
γ_4 :for every node n on the path beginning at the element r, the following condition holds $n.up.less(n) \vee n.up = none$.

Proposition 4.9. For every two nodes x, y in the tree T_s, $x.up = y \Rightarrow y.less(x)$.

Proof:
This property is invariant with respect to the operations insert and delete. At the very beginning the tree T is empty and the property holds. Assume that the property holds for a certain tree T. Consider another tree T' which is the result of operation insert or delete. After the insertion of an element the procedure correctUp is called and the tree is going to be repaired to keep the property. The same remark may be repeated in the case when the tree T' is the result of the operation delete on tree T. □

This sequence of observations leads to the following:

Lemma 4.8. In each observable state s the tree T_s is a heap.

Before proving the corectness of the implementation we should extend the class `PQ` adding two methods $empty$ and $member$.

Boolean empty() { return root=none}
Boolean member(Elem e, PQ q) { *the body of this method is given on the righthand side of the equivalence* a8 of Table 1}.

Now we are ready to verify the implementation `PQS` of priority queues against the specification $ATPQ$ given in Table 1.

Let us start with the structure consisting of elements and heaps $\{T_s : s \in \mathcal{S}\}$ and operations *insert* and *delete*, *min*, and the relations *empty* and *member*. Consider the quotient structure $\mathcal{PQS}$ in which we identify the heaps that have the same sets of elements. We claim that it is a priority queues structure, a standard model of the theory of $ATPQ$. It suffices to verify that the axioms of Table 1 are formulas valid in the structure $\mathcal{PQS}$.
a2) The program **while** not q.empty() **do** q.delete(q.min()) **done** always terminates since each operation delete removes one element from a heap.
a3) This follows from the lemmas 4.2 and 4.3.
a4) This follows from the lemmas 4.4 and 4.5.
The verification of the remaining axioms of $ATPQ$ is left to the reader. We can conclude:

Theorem 4.1. The structure $\mathcal{PQS}$ of elements and `PQ` objects implemented by the class `PQS` is a priority queue.

Remarks on cost:
The pessimistic cost of an operation *insert* or *delete* is $O(logn)$, where n is the number of elements in the priority queue. This property is very important in the application of priority queue as plan of experiment in the class Simulation that inherits (extends) the class PriorityQueue. Imagine, in an simulation experiment of a pandemia of influenza where the objects of SimProcess class count in hundreds of thousands and the number of EventNotice objects goes in milions, any implementation with the cost worse than $O(logn)$ would be impractical.

5. Final remarks

We have demonstrated the work on verification of a given class K against a specification S. Aswering the question *is the class K a correct implementation of the specification S* is a task completely different than proving correctness of an algorithm with respect to a given pre- and post-conditions. This paper shows that the formal counterpart of the task is asking whether a given class implements a set of axioms. In this context it is natural to conceive the class K as an algorithmic definition of some algebraic structure $\mathbb{A}$ and to study the question *is the structure $\mathbb{A}$ a model of the specification S*. We are stressing that high integrity programming requires many skills and a lot of invention. The analysis of this case study shows the wide repertoire of questions that may appear during the work on a software project. The incomplete list contains the following kinds of subgoals:

- specification of algorithms,
- specification of classes (this work is strongly related to the goal of specification of a data structure),
- construction of a method (i.e. procedure or function),
- construction of a class,
- verification of a conjecture given class K correctly implements some specification S.

In a future paper we shall demonstrate that, in a favorable circumstances, one can construct a class together with a proof of its correctness.

From previous sections we conclude that EOP (experiment, observe, prove) method can be successfully applied to software analysis. We do not claim it is a trivial task but it is at least realizable. As a matter of fact most programmers would agree that formal methods are generally better than sophisticated testing and simultaneously most of them are not using such methods at all. It seems that essential to the problem is lack of proper tools and experience i.e., tools which can integrate specification, implementation and verification tasks into single, consistent process. Experience can be gained only through everyday practice. Our goal is to develop integrated programming environment supporting every phase of software construction using formal methods. EOP idea presented in this paper is a part of bigger scheme called temporarily SpecVer programming. We assume that whole process of software production needs preparation of:

- specification documents: formal texts along with some math analysis (is it astonishing that some specifications are incorrect, or mor precisely they may be inconsistent or incomplete?),
- implementation code,
- verification reports: again formal texts with some proofs about implementation's quality.

As you can remark, EOP method could be used both for specification and verification tasks as soon as we can perform observations analogous to these made about Fig. 2. This again direct us towards programming tools. We need some object debugger showing program memory from object perspective, some formal support tools helping in logical theorems construction and perhaps checking if formulas are properly written and much, much more ... Some work has already been started, but a lot of it still should be done. For now we can present initial implementation of Eclipse plugin [10] supporting creation of specification documents.

References

[1] Amey, P.: Logic versus Magic in Critical Systems, *Reliable Software Technologies - Ada Europe 2001*, Lecture Motes in Computer Science 2043, Springer, 2001.

[2] Barnes, J.: *High Integrity Software*, Addison-Wesley, London, 2003.

[3] Beckert, B., Hähnle, R., Schmitt, P. H.: *Verification of Object-Oriented Software, the KeY approach*, LNAI 4334, Springer, Berlin, Heidelberg, 2007.

[4] Meyer, B.: *Eiffel*, P, W, 1987.

[5] Mirkowska, G., Salwicki, A.: *Algorithmic Logic*, PWN and J.Reidel, Warszawa, 1987.

[6] Mirkowska, G., Salwicki, A., Świda, O.: SpecVer - the methodology integrating specification, programming and verification, *Fundamenta Informaticae*, **85**, 2008, 343–357.

[7] Ratajczak-Bartol, W., Szczepańska-Wasersztrum, D.: *Data Structure for Simulation Purpose in Loglan77*, Technical Report 373, Institute of Computer Science, Polish Academy of Sciences, Warszawa, 1979.

[8] Ratajczak-Bartol, W., Szczepańska-Wasersztrum, D.: Code of Simulation and other Classes, http://duch.mimuw.edu.pl/∼salwicki/EOP/PQclass.html, October 2007.

[9] Salwicki, A.: On Algorithmic theory of Stacks, *Fundamenta Informaticae*, **3**, 1980, 311–332.

[10] Świda, O.: SpecVer - Specification, Verification, Programming, a plugin into Eclipse, http://aragorn.pb.bialystok.pl/∼swida/svp, April 2007.

Appendix - the class PQS

The full text of the class PQS as written by W.M. Bartol and D. Szczepańska. It is a part of bigger program of simulation of bank department [7] [8].

```
unit PQS : class; (* priority queues system *)
    unit PQ: class;
        var last,root:node;
        unit min: function: elem;
        begin
            if root=/= none then result:=root.el else throw new Undefined() fi;
        end min;
        unit insert: procedure(r:elem);
            var x,z:node;
        begin
            x:= r.lab;
            if last=none then
                root:=x; root.left, last:=root
            else
                if last.ns=0 then
                    last.ns:=1; z:=last.left; last.left:=x;
                    x.up:=last; x.left:=z; z.right:=x;
                else
                    last.ns:=2; z:=last.right; last.right:=x;
                    x.right:=z; x.up:=last; z.left:=x;
                    last.left.right:=x; x.left:=last.left; last:=z;
                fi
            fi;
            call correctUp(r)
        end insert;
        unit delete: procedure(r: elem);
            var x,y,z:node;
        begin
            x:=r.lab; z:=last.left;
            if last.ns =0 then
                y:= z.up;
                if y=none then root:=none else y.right:= last fi;
                last.left:=y; last:=y;
            else
                y:= z.left; y.right:= last; last.left:= y;
            fi;
            z.el.lab:=x; x.el:= z.el; last.ns:= last.ns-1;
            r.lab:=z; z.el:=r;
            (* the following three instructions were added during our verification *)
```

```
        z.left.right :=none; z.ns:=0; z.left, z.right, z.up := none;
        if x.less(x.up) then
            call correctUp(x.el)
        else
            call correctDown(x.el)
        fi;
    end delete;
    unit correctDown: procedure(r: elem);
        var x,z: node, t: elem, fin, log: Boolean;
    begin
        z := r.lab;
        while not fin
        do
            if z.ns = 0 then fin :=true
            else
                if z.ns=1 then x := z.left
                else
                    if z.left.less(z.right) then x:= z.left else x:=z.right fi
                fi;
                if z.less(x)
                then
                    fin := true
                else
                    t:= x.el; x.el :=z.el; z.el:=t;
                    z.el.lab :=z; x.el.lab:=x
                fi;
            fi;
            z:=x;
        od
    end correctDown;
    unit correctUp: procedure(r: elem);
        var x,z: node, t: elem, fin, log: Boolean;
    begin
        z := r.lab;
        x:= z.up;
        do
            if x=none then log:=true else log:=x.less(z) fi;
            if log then exit fi;
            t:=z.el; z.el:=x.el; x.el:=t;
            x.el.lab:=x; z.el.lab:=z; z:=x; x:z.up;
        od
    end correctUp;
end PQ;
```

```
    unit node: class (el:elem);
        var left,right,up: node, ns:integer;
        unit less: function(x:node): boolean;
        begin
            if x= none then
                result:=false
            else
                result:=el.less(x.el)
            fi;
        end less;
    end node;
    unit elem: class(prior:real);
        var lab: node;
        unit virtual less: function(x:elem):boolean;
        begin
            if x=none then
                result:= false
            else
                result:= prior ≤ x.prior
            fi;
        end less;
    begin
        lab:= new node(this elem);
    end elem;
    unit Undefined: Exception class;
    end Undefined
end PQS (* priority queues system *);
```