SpecVer & LEM'12 projects XX (2015) 1–17
Dombrova Research

Algorithmic Logic + SpecVer = the methodology for high integrity programming

Grażyna Mirkowska
Polish-Japanese Institute of Computer Technology
Koszykowa 86, 02-097 Warszawa, Poland
mirkowska@pjwstk.edu.pl

Andrzej Salwicki
National Institute of Telecomunication
Szachowa 1, 04-894 Warszawa, Poland
salwicki@mimuw.edu.pl

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

Abstract. Our aim is to present a methodology that integrates all phases of software's production beginning from the specification phase, through the phase of programming and finally the phase of verification of program against its specification. The theoretical background of the methodology is algorithmic logic [9]. The environment for practical activities of this software project is a plugin *SpecVer*[12] extending the Eclipse development platform [2].

1. Introduction

Software systems are growing and become more and more complicated. Accordingly grows the probability of error occurrences. Some errors seem to be simple, easy to repair. Therefore in spite of their serious consequences many programmers and many software companies depreciate them. The producers of software live in the world of $\mathcal{MAGIC}$ (see "Logic or Magic" [4]). The majority of them thinks that program once written and compiled is a good program. Eventually they admit that their product may contain some bugs and therefore it should be tested and improved. But what does it mean?
We can also observe the passive attitude of the customers. Customers rely on opinions of software companies and leave all decisions in their hands. It is a frequent case when a software company prepares specifications, writes programs, tests them and releases programs and bills to pay to customers.
First of all we argue that this bad habit must be changed, especially, when a large software system is going to be constructed. The customers should cooperate with three independent agents. Let's name them Designer, Programmer, Verifier. At the beginning of a software project Customer explains his(her)

need to the designer. Designer should prepare specification. (As we shall see in the next section the specifications should be carefully analysed.) Next, Customer commands a software from a Programmer. Programmer is to prepare an implementation of specification. Now, Customer should pass the two documents: the specification, and the program to Verifier. The goal of verification is to analyse the quality of software against the specification and to issue the constructive opinion for Customer: ”you should pay” or “you shouldn’t pay because the program is of poor quality”. The figure below illustrates this idea, which sometimes is called high integrity programming (HIP) [5].
We shall conclude this introduction with the examples of the positive attitude toward HIP. There is an evidence that the preparation of formal specification itself led to the substantial diminuation of costs of the whole project. In some cases it allowed to reduce the cost by 9 per cent. Some companies, NASA and Airbus among others, have divisions responsible for creation of specifications and application of formal methods.

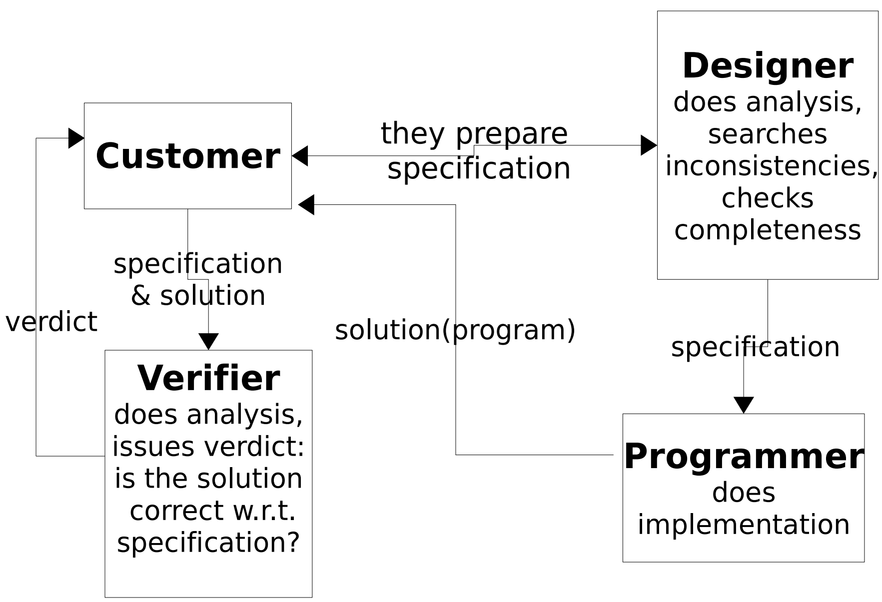

Fig. 1. Actors of the software production process and their interactions

2. Case study of specification - stacks

Let us begin with the explanation of the *principle of factorization*. The principle was formulated in a paper by C.A.R. Hoare [8]. It says that whenever appropriate, the task should be divided into two parts: implementation of an abstract data type, and implementation of an algorithm. For, in the most cases algorithms use data structures which are not the native structures of a computer. Example: Finding the center and the radius of the circle over a triangle. One can separate the work into the two subtasks: implement a data structure of planar geometry and use the data structure to program the algorithm. Now another subtask appears: to specify the data structure of planar geometry. This subtask has its formal counterpart: axiomatization of a theory of planar geometry. Let us consider the data stucture of stacks. We need a criterion which will be used to accept or to reject a given implementation of stacks. In nowadays practice implementations are written in the form of class declarations. An algorithm using stacks need not to analyze the implementation details of stacks. It may and should abstract from how the implementation is done. The analysis should use the properties mentioned in the specification. We are going to show that some specifications are better than others. The person or company doing specifications should not limit itself to writing just a specification. Specifications of some quality are needed.

We shall illustrate the problem of writing a good specification on the example of stacks. Most of us knows what stack is. At least players of the game canasta know. Any programmer used stacks at least once in his professional life. The shortest description is LIFO. Elements are put into stack and extracted. The LIFO means: Last In First Out. More precisely: we have some elements and stacks. We can push an element e into stack s obtaining a new stack $push(e, s)$. We can pop the most recent element from stack s obtaining a smaller stack. The most recent element of a stack is returned as the value of the function top. These two operations are partial ones. The result is not defined for $empty$ stack. Hence, one can say that the structure of stacks has its universe consisting from two sets: the set E of elements and the set S of stacks. Moreover, we have three operations: $push, pop, top$ and two predicates: $empty$ and equality $=$.

Table 1. Specification S1 of Stacks

Signature	Comments
Sorts	$Universe = E \cup S$
E	set of elements
S	set of stacks
Operations	
$push : E \times S \longrightarrow S$	put an element e into a stack s
$pop : S \longrightarrow S$	result is defined iff $\neg empty(s)$
$top : S \longrightarrow E$	result is defined iff $\neg empty(s)$
$newStack : \longrightarrow S$	the empty stack
Relations	
$empty : S \longrightarrow \{true, false\}$	is stack empty?
$=: E \times E \cup S \times S \longrightarrow \{true, false\}$	the equality relation
Axioms	
s1) $(\forall e \in E)(\forall s \in S)\ \neg empty(push(e, s))$	push returns a non-empty stack
s2) $(\forall e \in E)(\forall s \in S)\ e = top(push(e, s))$	the element last put into stack is the stack's top
s3) $(\forall e \in E)(\forall s \in S)\ s = pop(push(e, s))$	after executing push, pop restores the previous stack
s4) $empty(newStack)$	

Programmers conceive S as a set of potentially existing objects of a class S, similarly is conceived the set E. The Table 2 contains one programmed implementation of the specification S1 and one *"mathematical"* model of it.

Table 2. Models I_1

Programmed model	Mathematical model
class **Elem** { ... }	$E = \{a, b, c, ...\}$
class **Stos** {	S = set of all finite sequences over alphabet E,
private class Linkage {	the empty sequence λ included.
Linkage next;	$newstack = \lambda$
Elem el;	$push(e, \{e_1, e_2, ..., e_n\}) = \{e, e_1, e_2, ..., e_n\}$
Linkage(Elem e, Linkage n){el=e; next=n;}	
} // end Linkage	
public Linkage topv;	
public Stos(){topv=null;}	$top(\{e_1, ..., e_n\}) = e_1$
public static final Stos **push**(Elem e, Stos s) {	$top(\lambda)$ is undefined
Stos n = new Stos();	
n.topv = new Linkage(e, s.topv);	$pop(\{e_1, e_2, e_3, ..., e_n\}) = \{e_2, e_3, ..., e_n\}$
return n; } // end push	$pop(\{e_1\}) = \lambda$
public static final Elem **top**(Stos s) throws Undef {	$pop(\lambda)$ is undefined
if (s.topv=null) throw new Undef();	
return s.topv.el;	
} // end top	$empty(s) \equiv s = \lambda$
public static final Stos **pop**(Stos s) throws Undef {	equality = is meant as identity
if (s.topv==null) throw new Undef();	
Stos n =new Stos();	
n.topv=s.topv.next; return n;	
} //end pop	
public static final Boolean **empty**(Stos s) {	
return (s.topv==null);	
} // end empty	
public static final Boolean **equal**(Stos s1,Stos s2) {	stacks are equal iff
Boolean aux=true;	they have the same elements
Boolean aux1=Stos.empty(s1);	on the same positions.
Boolean aux2=Stos.empty(s2);	
while (!aux1&&!aux2&&aux) {	
aux = (Stos.top(s1) == Stos.top(s2));	
s1 = Stos.pop(s1); aux1 = Stos.empty(s1);	
s2 = Stos.pop(s2); aux2 = Stos.empty(s2);	
}	
return (aux1 && aux2 && aux);	
} // end equal	
} // end Stos	
class Undef extends Exception { ... }	

This mathematical model is called the standard model of stacks. For any given set E one can construct a standard model based on the set E. All models of the family of standard models of stacks are alike. They need not to be isomorphic however. To see this, consider two standard models: one based on a set E_1 and another based on the set E_2 of different cardinalities, $card(E_1) \neq card(E_2)$.

Many authors consider S1 as a specification of stacks, c.f. [7], [3]. However it is far from expressing the whole truth about the stacks as it is witnessed by the following lemma.

Lemma 2.1. The formula

$$(\forall s \in S)\ \neg empty(s) \Rightarrow\ s = push(top(s), pop(s))$$

saying: for every not empty stack s the result of $push$ operation on element $top(s)$ and the stack $pop(s)$ is the stack s itself, is independent of the axioms s1 - s4.

Proof:
Consider the data structure I_2, c.f. Table 3. Check that it is a model of axioms s1 - s4, i.e. all four formulas are valid in I_2. We shall prove that the formula mentioned in the lemma is not valid in this data structure. Consider the stack $s = \{e_1, e_2, e_3, ..., e_n\}$ such that $e_1 \neq e_2$. Obviously $top(s) = e_1$ and $pop(s) = \{e_3, ..., e_n\}$. Now $push(top(s), pop(s) = \{e_1, e_1, e_3, ..., e_n\} \neq s$. □

Table 3. Model I_2

$E = \{a, b, c\}$
S = set of all finite sequences over alphabet E, the empty sequence λ included.
$push(e, \{e_1, e_2, ..., e_n\}) = \{e, e, e_1, e_2, ..., e_n\}$
$top(\{e_1, ..., e_n\}) = e_1$
$top(\lambda)$ is undefined
$pop(\{e_1, e_2, e_3, ..., e_n\}) = \{e_3, ..., e_n\}$
$pop(\{e_1\}) = pop(\{e_1, e_2\}) = \lambda$, $pop(\lambda)$ is undefined

Therefore we can present another specification of stacks, c.f. Table 4.

Table 4. Specification S2 of stacks

Signature	the same as in S1
Axioms	
axioms s1 - s4 and	
s5) $(\forall s \in S)\ \neg empty(s) \Rightarrow$	for every not empty stack s
$s = push(top(s), pop(s))$	the result of operation $push$ on element $top(s)$ and
	the stack $pop(s)$ is the stack s

One may think the more formulas we add the better. This however may lead to inconsistent specifications. Look at the following example S3, c.f. Table 5.

Table 5. Specification S3 of stacks

Signature	like S1, augmented by two constants
a, b of type E	
Axioms	
axioms s1 - s5 and	
sQ) $\neg empty(s) \implies push(e, pop(s)) = pop(push(e, s))$	
and the axiom	
s2E) $a \neq b$	

Theorem 2.1. The set of formulas $\{s1 - s5, sQ, s2E\}$ is an inconsistent set.

Proof:
Axiom s2E) says that the set E has at least two elements a and b. Assume that $s \in S$ is a non-empty stack. Then we have:

(1)	$s_1 \stackrel{df}{=} push(a, s)$	by definition
(2)	$s_2 \stackrel{df}{=} push(b, s)$	by definition
(3)	$s = pop(s_1)$	from (1) by s3)
(4)	$s_2 = push(b, pop(s_1))$	from (2), (3), recall s is non-empty
(5)	$s_2 = pop(push(b, s_1))$	from (4) by sQ)
(6)	$s_2 = s_1$	from (5) by s3)
(7)	$b = top(s_2) = top(s_1) = a$	from (6) by s2)

Contradiction! It shows that the specification S3 is inconsistent. □

Corollary 2.1. Specification S3 has no implementation.

We shall expose the importance of this fact later. Let us return to the specification S2. After closer examination one may discover that it possible to add an infinite set of additional formulas. They all are conform with the scheme of (structural) induction for stacks. Hence we get the next specification S4, c.f. Table 6.

Table 6. Specification S4 of stacks

Signature	the same as in S1
Axioms	
axioms s1 - s5 and	
all formulas of the induction scheme IS	
IS) $\alpha(s/s_0) \wedge \{((\forall s \in S)(\forall e \in E)$	α is any first-order formula
$(\alpha(s) \implies \alpha(s/push(e, s)))\} \implies (\forall s \in S)\alpha(s)$	s_0 denotes newStack

Induction scheme says: if a formula $\alpha(x)$ is valid for the empty stack s_0 and if for every stack s and for every element e, $\alpha(x/s)$ implies $\alpha(x/push(e, s))$ then one may conclude that for every stack s the formula $\alpha(x/s)$ holds. The formula does not say that there are not pathological stacks. One may say: *we shall consider only standard stacks, i.e. the stacks obtained from the empty stack in finite number of operations $push$.* But how to express this property as an axiom? Instead, one may say: *I am going to consider only programmable models of specification S4.* Even adding such extra requirement we can not eliminate pathological stacks. In fact papers [10, 11] prove that there exist pathological models of specification S4. Pathological means here that there are stacks which can be popped without end and no empty stack results.

Theorem 2.2. There exists a programmable model of specification S4 such that for certain stack s_1 the program

$$\textbf{while} \neg\, empty(s_1) \textbf{ do } s_1 := pop(s_1) \textbf{ done}$$

never terminates.

Such a model is called *unreachable* and obviously presents some pathology. For the proof see [10, 11]. The second paper proves two facts.

Theorem 2.3. The set of first order formulas valid in the data structure of stacks over a finite set E of elements is decidable.

This seems to be a good message. It is nice to have a procedure deciding about truth of formulas. However, it turns to a bad message as is shown by the following

Theorem 2.4. For any decidable, first order theory $\mathcal{T}$ there exist a programmable and unreachable (i.e. pathological) model of $\mathcal{T}$.

It would seem that we are in an *impasse*. That it is impossible to axiomatize data structures. In spite of promises like [6], algorithmic logic comes here with help. One can consider the following specification S5, where scheme of induction is replaced by one *algorithmic* formula, c.f. Table 7.

Table 7. Specification S5 of Stacks

Signature	the same as in S1
Axioms	
axioms s1 - s5 and	
s6) **while** $\neg empty(s)$ **do** $s := pop(s)$ **done true**	the program always terminates i.e. every stack is finite

One may prove the following theorem on representation [9]

Theorem 2.5. Any model of the specification S5 is isomorphic with a standard model of stacks.

The theorem says that specification S5 captures all properties of data structure of stacks. If someone presents a model of S5 then it is necessarily isomorphic to the structure, where stacks are finite sequences of elements and the operations push, pop and top are defined as LIFO operations. What is also important we have as an axiom which guarantees that the program mentioned in the axioms always halts. This property can be useful in proofs of correctnes of other algorithms.

We have seen enough examples of specifications. Let us compare them, c.f. Table 8.

Table 8. Comparison of various specifications

Spec.	Remarks
S_1	incomplete information, e.g. formula s5 is independent of the set $\{s1, s2, s3, s4\}$ S_1 has surprising implementations c.f. implementation I_2
S_2	assume card(E) = k is an integer(is finite), then the theory S_2 is decidable, incomplete information, allows pathological implementations
S_3	inconsistent specification, c.f. Theorem 2 no implementation may exist
S_4	decidable, incomplete information, allows pathological implementations
S_5	complete information, any implementation is isomorphic to a standard one, the (algorithmic) theory is undecidable.

Remark 2.1. Decidability is the property of the set of first order formulas valid in a data structure of stack over a finite set E of elements. Nethertheless, the specifications S2 and S4 have non-standard models.

3. Verification of algorithms - an instructive example

There are many texts on verification of algorithms. The reader will find them without difficulties. The calculi of Floyd-Hoare and of Dijkstra are the best known examples. We recall that both calculi are embedded in the calculus of algorithmic logic[9].
Below, we quote an example of a proof in algorithmic logic. Observe that using algorithmic axiom s6) we were able to build a complete specification of stacks. Now, we obtain a *bonus*, the proof of termination or correctness may start from the axiom. We are going to prove that the method $equal$ always terminates and never fails. In the view of the theorem 2.2 the specification S4 is not sufficient to prove the halting property (and hence the correctness) of method $equal$. Consider the case when one of arguments of the method $equal$ is a nonstandard stack which can be popped *ad infinitum*.
If the specification S5 was used then the proof of termination is a formality.

Lemma 3.1. The algorithm of the method $equal$ always terminates, does not fail nor throws an exception.

Proof:
A sketch of the proof is as follows. One has to demonstrate that the instruction **while** never leads to an infinite computation nor to throwing an exception.

$$S5 \vdash \textbf{while } not\ empty(s1) \textbf{ do } s1 := pop(s1) \textbf{ done } true.$$

The notation $Z \vdash \phi$ reads "formula ϕ has a proof from the set Z of formulas",
in this case an instance of the axiom s6 is a part od the set S5, hence it is provable from S5

Now, in a few easy steps we shall deduce that the body of the method $equal$ is a program that always terminates. First, we rewrite the axiom according to the requirements of Java's grammar.

$$S5 \vdash \textbf{while } not\ Stos.empty(s1) \textbf{ do } s1 := Stos.pop(s1) \textbf{ done } true.$$

We use the following (auxiliary) inference rule

$$\boxed{\frac{\textbf{while } \gamma \textbf{ do } \mathrm{K} \textbf{ done } \alpha}{\textbf{while } \gamma \textbf{ do } \mathrm{N;\ K;\ M} \textbf{ done } \alpha}}$$

where the programs M and N terminate and do not throw an exception and do not change the variables of formulas α, γ nor variables of program K.

Now, we have

$$S5 \vdash \left[\begin{array}{l} \textbf{while } not\ Stos.empty(s1) \\ \textbf{do} \\ \quad \textbf{if } not\ Stos.empty(s2) \textbf{ then } aux := (Stos.top(s1) = Stos.top(s2)) \textbf{ fi}; \\ \quad s1 := Stos.pop(s1); \\ \quad \textbf{if } not\ Stos.empty(s2) \textbf{ then } s2 := Stos.pop(s2) \textbf{ fi}; \\ \textbf{done } true \end{array}\right.$$

Next, we apply another inference rule

$$\boxed{\frac{\alpha \Rightarrow \beta}{\textbf{while } \beta \textbf{ do } \mathrm{K} \textbf{ done true} \Rightarrow \textbf{while } \alpha \textbf{ do } \mathrm{K} \textbf{ done true}}}$$

in order to replace the iteration condition $\beta:\ notempty(s1)$, by a stronger one, $\alpha:\ (notempty(s1) and notempty(s2) anda$
Now we proved

$$S5 \vdash \left[\begin{array}{l} \textbf{while } (not\ Stos.empty(s1)\ and\ not\ Stos.empty(s2)\ and\ aux) \\ \textbf{do} \\ \quad \textbf{if } not\ Stos.empty(s2) \textbf{ then } aux := (Stos.top(s1) = Stos.top(s2)) \textbf{ fi}; \\ \quad s1 := Stos.pop(s1); \\ \quad \textbf{if } not\ Stos.empty(s2) \textbf{ then } s2 := Stos.pop(s2) \textbf{ fi}; \\ \textbf{done } true \end{array}\right.$$

At present we can apply the following rule

$$\boxed{\frac{\textbf{while } \alpha \wedge \beta \textbf{ do if } \beta \textbf{ then } I \textbf{ fi};\ K \textbf{ done } \gamma}{\textbf{while } \alpha \wedge \beta \textbf{ do } I;\ K \textbf{ done } \gamma}}$$

and prove

$$S5 \vdash \left[\begin{array}{l} \textbf{while } (not\ Stos.empty(s1)\ and\ not\ Stos.empty(s2)\ and\ aux) \\ \textbf{do} \\ \quad aux := (Stos.top(s1) = Stos.top(s2)); \\ \quad s1 := Stos.pop(s1); \\ \quad s2 := Stos.pop(s2); \\ \textbf{done } true \end{array}\right.$$

Now we can apply the rule

$$\boxed{\frac{\alpha\ ,\ Ktrue}{K\alpha}}$$

and obtain

$$S5 \vdash \left[\begin{array}{l} aux := true; \\ \textbf{while } (not\ Stos.empty(s1)\ and\ not\ Stos.empty(s2)\ and\ aux) \\ \textbf{do} \\ \quad aux := (Stos.top(s1) = Stos.top(s2)); \\ \quad s1 := Stos.pop(s1); \\ \quad s2 := Stos.pop(s2); \\ \textbf{done } true \end{array}\right.$$

In this way we proved that the body of the method $equal$ always terminate without raising an exception.

□

We observe that this proof does **not** use induction. One may say that algorithmic axiom of stacks supersedes in a way the scheme of structural induction. Moreover, we gained in the clarity of the proof.

4. Verification of an implementation

Our present goal is to verify that the classes $Elem, Stos$ and $Undef$ given in Table 2 define a data structure which models all axioms of the specification S5 of stacks. Now consider the set $|Stos|$ of all objects s that satisfy the relation *instanceof* $Stos$

$$|Stos| = \{s : s \text{ instanceof } Stos\}$$

and

$$|Element| = \{e : e \text{ instanceof } Element\}$$

together with the functions:
$push_S : \ |Element| \times |Stos| \longrightarrow |Stos|$, defined by the expression $push(s, e)$,
$pop_S : \ |Stos| \longrightarrow |Stos|$, defined by the expression $pop(s)$,
$top_S : \ |Stos| \longrightarrow |Element|$, defined by the expression $top(s)$,

$empty_S : \; |Stos| \longrightarrow \{true, false\}$, defined by the expression $empty(s)$,
$equal_S : \; |Stos| \times |Stos| \longrightarrow \{true, false\}$ defined by the method equal.
We shall use the following notation $Stos \models \alpha$ and read it as "$Stos$ models α", or "the formula α is valid in the implementation $Stos$", see [9] or any textbook on logic for the definition of satisfiability and validity. We are writing briefly $Stos$ instead of $Elem$ and $Stos$, inorder to keep our notation shorter. On the other hand to be an object of class Elem means to be stackable and nothing more. Which again justifies our convention.
With these notations we start the analysis of the implementation $Stos$ against the specification $ATPQ$.

Lemma 4.1.

$$Stos \models \forall_{(s \in |Stos|)} \forall_{(e \in |Elem|)} \neg empty_S(push_S(e, s)) \tag{1}$$

Proof:
Let s be an object of class $Stos$ or of certain class that extends the class $Stos$. Let e be any object such that the relation "e instanceof $Elem$" holds. From the definition of the metod $push$ it follows that the attribute $topv$ in the object s' being the value of $push(s, e)$ is not null, $topv \neq null$. According to the definition of the metod $empty$, the value returned by the method $empty$ in the object s' is false. □

Lemma 4.2.

$$Stos \models \forall_{(s \in |Stos|)} \forall_{(e \in |Elem|)} e = top_S(push_S(e, s)) \tag{2}$$

Proof:
As the result of method top applied to the object $push(s, e)$ one obtains the object e. □

Lemma 4.3.

$$Stos \models \forall_{(s \in |Stos|)} \forall_{(e \in |Element|)} equal_S(s, pop_S(push_S(e, s))) \tag{3}$$

Proof:
The objects s and $pop_S(push_S(e, s))$ are not identical. However they satisfy the relation $equal_S$ defined by the metod $equal$. Proof by easy verification. □

Lemma 4.4.

$$Stos \models empty_S(newStos()) \tag{4}$$

Proof:
The value of the field *topv* in the object *new Stos()* is null. □

Lemma 4.5.

$$Stos \models \forall_{(s \in |Stos|)} \textbf{while} \neg \, empty_S(s) \; \textbf{do} \; s := pop_S(s) \; \textbf{done} \; true \tag{5}$$

Proof:
The proof uses two observations:
a) If an object s instanceof $Stos$ is the result of finitely many operations $push$ and pop applied to the object $new\ Stos()$ then it satisfies the termination property of the above program. Proof is by induction on the number of applied operations. It is obviously true for $s_0 = newStos()$. Now suppose that

the program P terminates for an object s_k which is the result of k operations $push$ and pop applied to the object s_0. Let e be an arbitrarily chosen element of the set $|Elem|$. Consider an object $s_{k+1} = push(s_k, e)$. It is evident that $\neg empty(s_{k+1})$ and that the $pop(s_{k+1}) = s_k$. We are using an instance of the following axiom of algorithmic logic

$$\textbf{while } \gamma \textbf{ do } K \textbf{ done } \alpha \iff \textbf{if } \gamma \textbf{ then } K; \textbf{while } \gamma \textbf{ do } K \textbf{ done endif } \alpha \qquad (W)$$

where γ is $\neg empty(s)$, α is $true$, K is $s := pop(s)$, and the value of the variable s is s_{k+1}. The formula on the right hand side is satisfied, it follows from the assumption on the object s_{k+1} and from the induction assumption on the object s_k. Hence the left hand side is also satisfied.
b) There are no other objects of class Stos. This follows from the fact that only operations of type $Stos$ allowed on an object of class $Stos$ are $push$ and pop. An attempt to manipulate the attribute $next$ of objects of class $Linkage$ outside the class $Stos$ is impossible because the inner class Linkage is private in the class $Stos$. Moreover, since the methods of the class $Stos$ are final, no one can modify them in a class derived from the class $Stos$. □

Lemma 4.6.

$$Stos \models \forall_{(s \in |Stos|)} \neg\, empty(s) \implies sequal_S push_S(top_S(s), pop_S(s))). \qquad (6)$$

Proof:
If s is $empty$ then the implication is satisfied. In the view of the previous lemma we know that each object of type $Stos$ represents a finite sequence of elements of set $|Elem|$. Our proof is by induction with respect to the length of stack. Suppose that our thesis is not valid. Let s be an object representing the shortest sequence of elements of $Elem$ such that the formula (6) holds. Let e be any object of type $Elem$. Now consider $s' = push_S(e, s)$. From the lemmas 2 and 3 we know that $top_S(s') = e$ and $sequal_S pop_S(s')$. Let us evaluate the formula $equal_S(s', push_S(top_S(s'), pop_S(s'))$. The value of this formula is equal to value of the following algorithmic formula.

begin
  $s1 := s'$;
  $s2 := push_S(top_S(s'), pop_S(s'))$;
  $aux := true$;
  while$(aux \wedge \neg\, empty_S(s1) \ \wedge \neg\, empty_S(s2))$
  do
    $aux := (top_S(s1) = top_S(s2))$;
    $s1 := pop_S(s1)$;
    $s2 := pop_S(s2)$;
  done
end $(aux \ \wedge \ empty_S(s1) \ \wedge \ empty_S(s2))$

Once again we can apply the axiom (W) to obtain an equivalent formula

begin
$s1 := s'$;
$s2 := push_S(top_S(s'), pop_S(s'))$;
$aux := true$;
$aux := (top_S(s1) = top_S(s2)); // = true,$ for $top_S(s1) = e$ and $top_S(s2) = e$
$s1 := pop_S(s1); // \ s1 = s$
$s2 := pop_S(s2); // \ s2 = s$
while$(aux \wedge \neg\, empty_S(s1) \ \wedge \neg\, empty_S(s2))$
do
$aux := (top_S(s1) = top_S(s2))$;
$s1 := pop_S(s1)$;
$s2 := pop_S(s2)$;
done
end $(aux \wedge empty_S(s1) \wedge empty_S(s2))$

The loop while compares the object s to itself. It will terminate and bring answer $true$. Hence the object s' has also the property (6). This ends the proof of the lemma 4.6. □

The six lemmas prove that all six axioms of the specification S5 are valid in the implementation defined by the classes $Elem$, $Stos$ and $Undef$, c.f. Table 2.
Therefore we can state that

Theorem 4.1. The classes Elem and Stos correctly implement the specification S5. □

Remark 4.1. We can say even more: Not only the classes $Elem$ and $Stos$ model the axioms of algorithmic theory of priority queues, but any pair of classes C, D such that $class\ C\ extends\ Elem$, and $class\ D\ extendsStos$. The structure of the class C may be arbitrary, what is important can be said as follows: *any object of class C is stackable*. On the other hand the class D derived upon the class $Stos$ can not introduce too many changes. It is guaranteed by saying that the class $Linkage$ is private inside the class $Stos$ and that the methods $push, top, pop, empty$ and $equal$ are final.
One can prove that the relation defined by the method $equal$ is a congruence We have proved that the method never loops nor fails, see Lemma 3.1. It remains to be proved that the relation is an equivalence relation: reflexive, transitive and antisymmetric. One can show this and more, namely that the relation is congruent with respect to the methods: $push, pop, top$.

Remark 4.2. One may ask: From where comes our confiance to the quality of the implementation? How you can quarantee that besides the properties expressed in the lemmas above, the implementation is free of some strange, not yet discovered, properties?
Our answer uses the fact that the specification S5 is categorical w.r.t. the set E. It means that any model of S5 where the set E is fixed, i.e described up to isomorphisms, is isomorphic to the standard model of stacks over E. It means that no malicious property is hidden inside the software.

Remark 4.3. One could include the definition of equality of stacks as an axiom. We suggest to use the algorithmic definition, i.e. the method $equals$. In the preceding section we proved that the termination property of $equal$ is provable from the other axioms and therefore it is valid in any data structure that validates the axioms s1 - s6.

5. Methodological remarks concerning process of specification construction

Now, with the intuition guided by the three previous sections we can continue our methodological considerations. The joint work of a Customer and Designer has as a goal to build a specification of software to be constructed. One should differentiate specification $\mathcal{S}_{\mathcal{A}}$ of an algorithm A from the specification $\mathcal{S}_{\mathcal{DS}}$ of a data structure DS. Let us discuss briefly the latter case. The specification of a data structure (or a class) should be of quality and has to assure that the following two features are achieved:

- the data structure in question is fully specified,
- the specification provides enough properties of the data structure to be used later in analysing termination and correctness properties of algorithms that apply the data structure.

We are hoping that the reader is now convinced that these goals are reachable. As well as the goals the designer should be aware of dangerous traps. One should avoid

- inconsistent specifications, as well as,
- incomplete specifications.

If a specification is *inconsistent* then no implementation exists. Should the programmer begin the work on implementation, then its time and money are lost. The programmer may start with the study of eventual inconsistencies in the submitted specification. But this is beyond his competence, or it leads to the serious increase of the costs of software project. If the specification submitted to the programmer is *incomplete*, then it can not serve as a criterion of eventual acceptance/rejection of proposed implementation. There is quite serious risk that an implementation will be accepted which does not satisfy one of forgotten properties and that in fact it had to be rejected and replaced by a new one. Again there is a probability of a loss of money angaged so far. Moreover, if an implementation is based on an incomplete specification, then one may have difficulties in proving correctness of algorithms which use this data structure. Consider the example of specification S1 and implementation I_2.

6. SpecVer Example - Eclipse plugin

In this section we present some snapshots of the SpecVer environment. This is more an illustration than a complete tool but it shows that formal specification can be adopted into popular programming environments. Basic idea is to allow designers/programmers to create specifications using programming development platform Eclipse, next step would be implementation of semi-automatic tools supporting verification. At present the user of Specver plugin may create SpecVer projects, specifications of classes, modules of source code that implements specifications and files containing arguments of verification (analysis) of software modules such as classes or methods against their specifications.

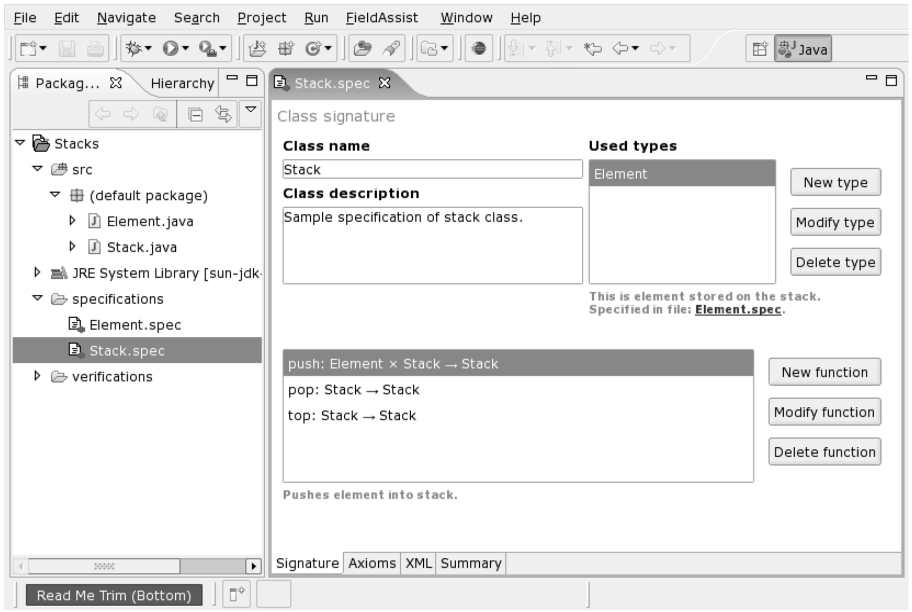

Fig. 2 Specification editor - class signature

The snapshot of Fig. 2 shows some stage in editing the signature of specification. Such signature may be used to produce a skeleton of a class.

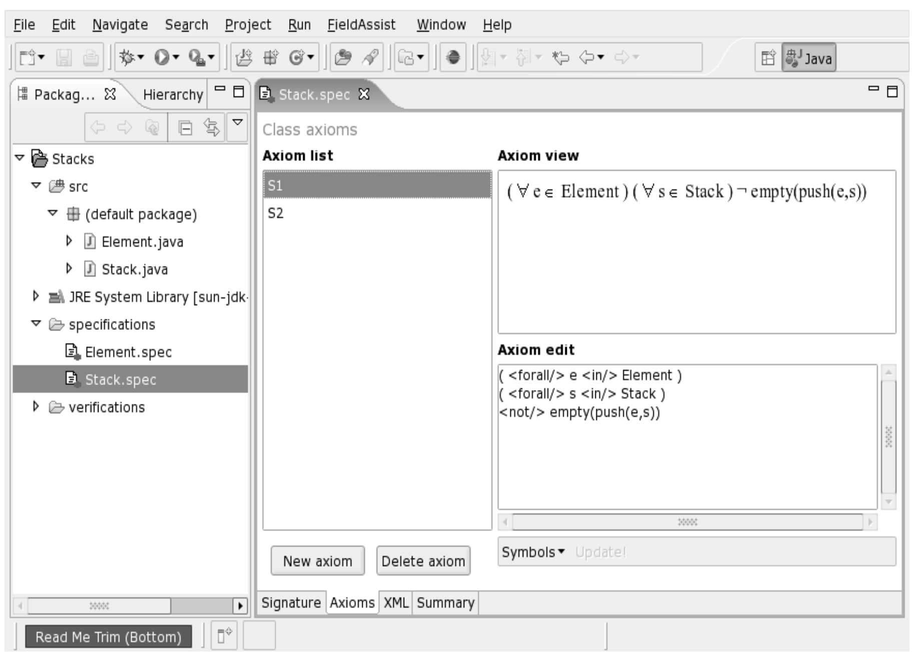

Fig. 3 Specification editor - editing logic formulas

This snapshot shows the work on a logic formula i.e. an axiom of a specification.

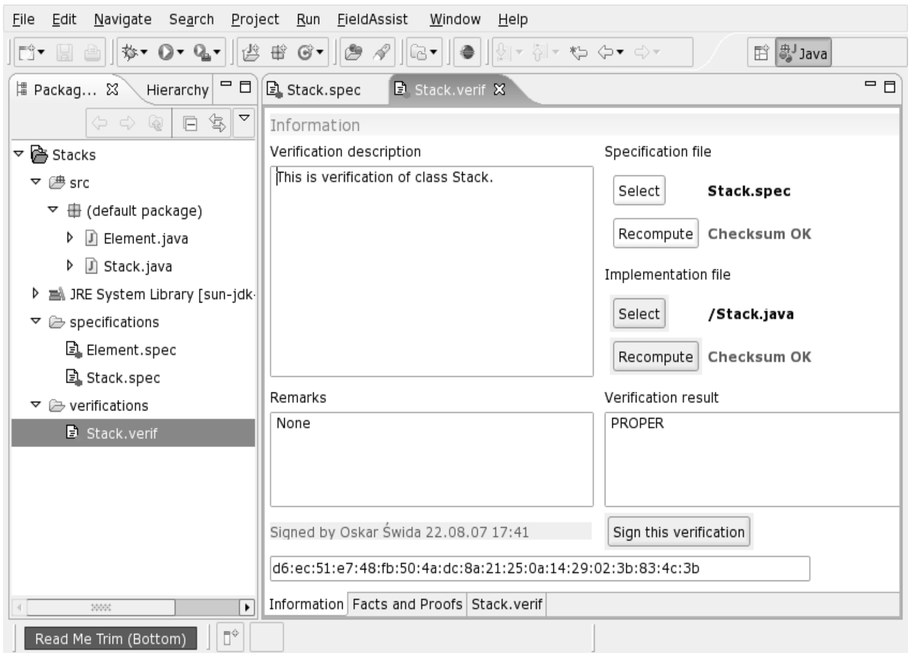

Fig. 4 Verification report editor - sample idea

The third snapshot shows that the author of verification report gave his verdict ”Proper”. In a corresponding file one may write the details of verification report. The Verifier stamps the files of implementation, specification and verification in a way. Any change in one of these files makes the verdict useless.

7. Final remarks

We show that our methodology is based on a sound and complete logic calculus of algorithmic logic, we show also that it is possible to have one environment where all documents of a software project are created, edited and stored: specification, modules of software, texts of program analysis, etc. Let us stress that what we have done is only the beginning. Much work should be done. Many projects are required to make the SpecVer environment a mature tool. On one hand we should develop the theoretical backgrounds. Perhaps the reader remarked that the programmed and the matemathical models of Table 1 differ. In order to eliminate this difference the algorithmic logic of programs with exceptions and error handlers should be developed. Some new tools should be constructed such as temporal logic within algorithmic logic (it is possible) and a logic of concurrent and distributed programs. A new virtual machine should be proposed together with its specification – axiomatization. This seem essential in order to facilitate proofs of semantical properties of programs. On the other hand the SpecVer system needs new modules. We plan to include the following new features:

- semi-automatic compatibility checking between specification and implementation,
- support for specification and verification documents,
- database of well-known (perhaps proved) algorithmic logic formulas for software pieces,

- model verification,
- fast prototyping,
- proof checks with Mizar software,
- object debugger.

Some friends asked: what do you mean by methodology?
Methodology – is a system of methods and principles for doing something[1].

References

[1] *Collins Cobuild English Languge Dictionary*, Collins, 1987.

[2] Eclipse - open development platform - homepage, http://www.eclipse.org, 2008.

[3] Abreu, J., Vasconcelos, V. T., Nunes, I., Lopes, A., Reis, L. S., Caldeira, A.: ConGu, The Specification and the Refinement Languages, http://labmol.di.fc.ul.pt/congu/, March 2007.

[4] Amey, P.: Logic versus Magic, *Critical Systems, Reliable Software Technologies - Ada Europe 2001*, LNCS, Springer, Berlin, 2001.

[5] Barnes, J.: *High Integrity Software*, Addison-Wesley, London, 2006.

[6] Diller, A.: *Z: An Introduction to Formal Methods*, J. Wiley, Chichester, 1990.

[7] Ehrig, H., Mahr, G., Eds.: *Fundamentals of Algebraic Specification 1*, Springer, 1985.

[8] Hoare, C.: Proof of correctness of data representation, *Acta Informatica*, **1**, 1972, 271–281.

[9] Mirkowska, G., Salwicki, A.: *Algorithmic Logic*, PWN & D.Reidel, Warszawa, 1987, ISBN 90-277-1928-4.

[10] Mirkowska, G., Salwicki, A.: The Algebraic Specification do not have the Tennenbaum property, *Fundamenta Informaticae*, **28**, 1996, 141–152.

[11] Mirkowska, G., Salwicki, A., Srebrny, M., Tarlecki, A.: First-Order Specifications of Programmable Data Types, *SIAM Journal on Computing*, **30**, 2000, 2084–2096.

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