SpecVer & LEM'12 projects XX (2015) 1–14
Dombrova Research

Creating a class from specification DRAFT – please do not copy

Grażyna Mirkowska & Andrzej Salwicki
Faculty of Mathematics and Natural Sciences
University Cardinal Stefan Wyszyński
Wóycickiego 1/3, 01-938 Warszawa, Poland
G.Mirkowska@uksw.edu.pl | A.Salwicki@uksw.edu.pl

Abstract. This paper presents the work on creating a class from the specification together with a proof of its correctness. Our starting point are two specifications: specification ATPQ of priority queues and specification Simulation of Simulation class.
Analogy with programming of an algorithm: there we have two conditions: a precondition and a postcondition. Specifications we consider have more complicated structure. Each specification consists of a signature and axioms or postulates.
Differences:

1. Introduction

In earlier articles we discussed the problems related to the specifications. In [6] we demonstrated the risks of creating inconsistent or incomplete specifications. We made an example of complete specification. The algorithmic formulas used there allow to show that any two implementations of the specifcation are isomorphic. We also remarked that the algorithmic logic makes a formal base for proofs of correctness of algorithms. In other article [7] we demonstrate ... In these articles specifications are viewed as somewhat theoretical, abstract beings. In fact, we talk of formalized algorithmic axiomatizations. Yet, the practice of programming brings the concept similar to specifications, it is interface module.

This article is third in series. In [6] we discussed the problems related to the creation of a specification of a class. In [7] we presented a proof of correctness of a class w.r.t. a specification.

Now, we illustrate that in some circumstances one may build a class using only knowledge of two specifications: specification $\mathcal{S}_A$ of the base class A and a specification $\mathcal{S}_B$ of a target class B that inherits the class A. No knowledge on the body of the inherited class is needed.

The meaning of this result is the following: the created class B will be correct with any class A provided it correctly implements the specification $\mathcal{S}_A$. One possible application of this result is a quick

construction of software. A prototype class A may be used in order to quickly construct the system consisting of classes A and B. Later, one may replace the class A by a more efficient implementation.

A comment is in place here: our specifications can be compared with the interface modules of Java. It turns out that:

1° Interfaces limit themselves to the signature part of an specification. The specification files .spec of the SpecVer system contain signatures as well as properties (or invariants, or axioms) of specified classes.

2° Interfaces can not prevent misinterpretation of the signature. Imagine, an interface

```
interface Stacks {
    Stacks push(Element e, Stacks s) { }
    Stacks pop(Stacks s) { }
    Element top(Stacks s) { }
}
```

What will happen if a class Stacks implemnts this specification in the manner of FIFO instead of LIFO? This and similar examples demonstrate that specifications of Java do not garantee anything but that the class implementing an interface has declared some methods of given names and parameters.

3° Interfaces are not complete. Obviously, one interface I may be implemented in many ways. There is no way to express that any implementation of I must ...

In section 2 we give the specification for the class Simulation. Section 3 presents step by step work on implementation of the class Simulation. Appendix A contains the specification of the base class PQS. Appendix B contains an informal, yet sufficiemtly complete, specification of coroutines.

2. Specification of Simulation class

Now we are going to describe and to axiomatize a system of discrete event simulation. There are numerous situations in which we have to deal with processes to be simulated, for example:

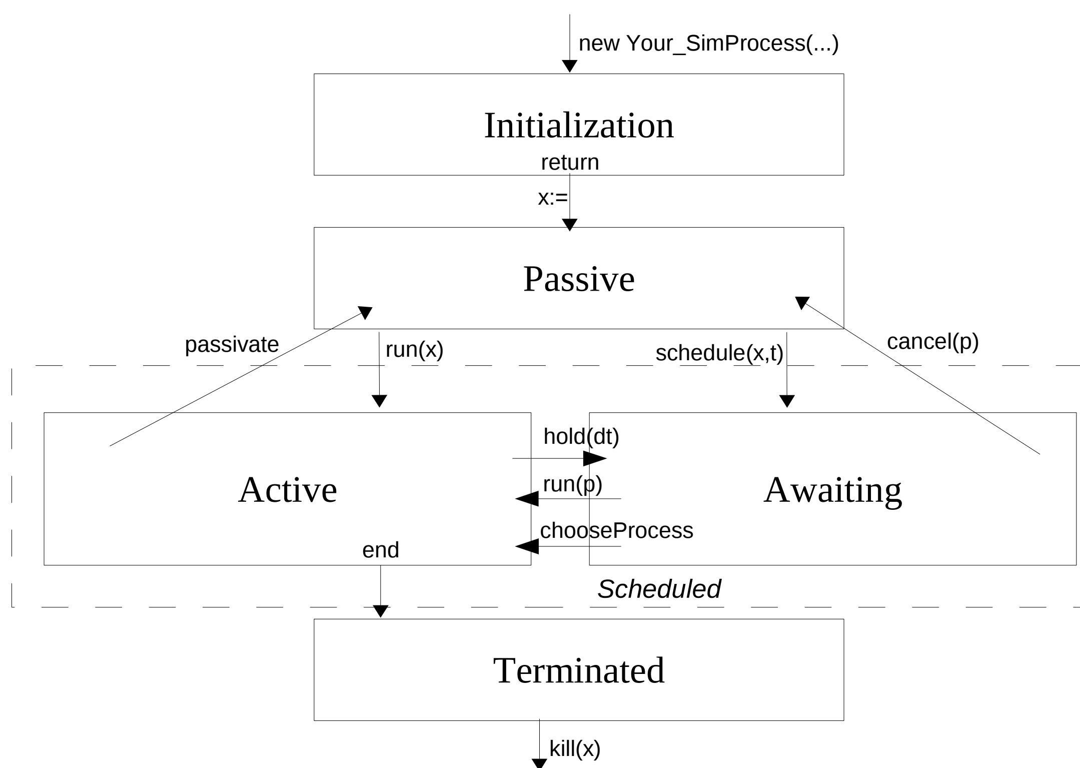
- system of patients and medecine doctors,
- system of vehicles and street lamps,
- system of ...kupno-sprzedaż

Class Simulation is meant as a base for further extensions. It offers a prototype of simulated processes and defines a set of basic operations on processes. The universe of the system $Simulation$ of simulated processes and discrete events consists of four disjoint subsets: $SimProcess, EventNotice, Time$ and $SimulationPlans$. Objects of type $SimProcess$ are quasi-threads, more precisely they are coroutines. Each object of type $SimProcess$ has a thread, however only one coroutine is executed in a moment. The control passes from one coroutine to another grace the direct command attach(x).

The type $Time$ may be a predefined class.

The type $EventNotice$ has two fields: process and time.

The type $SimulationPlan$ is the data structure of priority queues of $EventNocices$



3. Constructing Simulation class

3.1. From specification's signature to the skeleton of the class Simulation

The first step is a simple,almost automatic, translation of the specification's signature onto the skeleton of the class Simulation.

Specification's signature	**Class' skeleton**

Types:
$simprocess$
$plan_of_simulation \subset PQ$
$time$
$eventnotice$
 	
Operations:
current - this process is currently active,
$current : PQ \to SP$
time - the value of currently simulated time,
$time : PQ \to T$
schedule - enables planning of events,
$schedule : (SP \times T) \times PQ \to PQ$
hold - suspend a current process for a while,
$hold : T \times PQ \to PQ$
run - immediately execute the indicated process,
$run : SP \times PQ \to PQ$
passivate - suspends the current process,
$passivate : PQ \to PQ$
cancel - removes a process from simulation plan,
$cancel : SP \times PQ \to PQ$
idle?
$idle : SP \to Boolean$
terminated?
$terminated : SP \to Boolean.$
	unit Simulation: PriorityQueues class
	
	unit Simprocess: elemFIFO coroutine;
	unit isIdle: function: Boolean;
	unit isTerminated: function: Boolean;
	end Simprocess;
	
	unit EventNotice: elemPQ class; ...
	end EventNotice;
	
	unit PlanSymulacji: QueueHead class;
	unit schedule : procedure(p: SimProcess, t: time):
	unit hold: procedure(dt: time);
	unit run: procedure(p: SimProcess);
	unit passivate: procedure;
	unit cancel: procedure;
	unit chooseProcess: procedure;
	unit currentProcess: function: SimProcess;
	unit currentTime: function: time;
	var currProcess: SimProcess, currTime: Time;
	end PlanSymulacji;
	
	unit Time: class ... end Time;
	
	var SQS: PlanSymulacji;
	end Simulation;

Below we gathered the postulates (invariants, axioms) the class Simulation should obey.

$$SQS\ is\ a\ finite\ set. \tag{S1}$$

$$EventNotice = SimProcess \times Time \tag{S2}$$

$$SQS.currentProcess = (SQS.min\ qua\ EventNotice).p \tag{S3}$$

$$SQS.currentTime = (SQS.min\ qua\ EventNotice).t \tag{S4}$$

$$\neg\, idle(p, pq) \implies (\exists t) member((p, t), pq) \tag{S5}$$

$$\forall_{pq \in SimulationPlan} \forall_{p \in Simprocess} member((p, t_1), pq) \wedge member((p, t_2), pq) \implies t_1 = t_2 \tag{S6}$$

$$pq = c \wedge idle(p,pq) \wedge \neg terminated(p,pq) \implies \tag{S7}$$
$$[callschedule((p,t),pq)]\neg idle(p,pq) \wedge pq = insert((p,t),c)$$
$$(pq = o \wedge \neg idle(p,pq) \wedge \neg terminated(p,pq)) \implies \tag{S8}$$
$$[callschedule((p,t),pq)](idle(p,pq) \wedge pq = insert((p,t),delete((p,time),o)),$$
$$terminated(p,pq) \implies [callschedule((p,t),pq)]\{ERROR\}, \tag{S9}$$
$$[\text{call } hold(t,pq)]\ \alpha \equiv [\text{call } schedule((current(pq),time+t),pq)]\ \alpha, \tag{S10}$$
$$(current(pq) = p' \wedge \neg terminated(p,pq)) \implies \tag{S11}$$
$$\{[callrun(p,pq)]\alpha \equiv [callschedule(p,time,pq)]\alpha\}$$
$$(p = current(pq) \wedge pq = o) \implies \tag{S12}$$
$$[\text{call } passivate(pq)](idle(p,pq) \wedge pq = delete((p,time),o))$$
$$pq = o \implies [\text{call } cancel(p,pq)](idle(p,pq) \wedge pq = delete((p,t),o)). \tag{S13}$$

We made the following decisions:

- We introduce the class PlanSymulacji derived upon the class Queuehead which implements priority queue [7].
- Instead of functions $schedule, run, hold, passivate, etc.$ of type PQ with one argument of type PQ we declare methods $schedule, run, hold, passivate, etc.$ within class PlanSymulacji. In this way we spare transferring argument and receiving the result. This simplifies the body of class Simulation and the usage of it.

The full text of the first version is here.
Now we ought to fill the bodies of methods schedule, run, hold, etc in such a way that the properties S1 – S13 are valid.

3.2. Property S1

$$SQS\ is\ a\ finite\ set. \tag{S1}$$

Remark that the property S1 is garanteed. For the variable SQS points to a priority queue object.

3.3. Property S2: EventNotice = Simprocess × Time

$$EventNotice = SimProcess \times Time \tag{S2}$$

Property S2 says: objects of type $EventNotice$ are pairs $\langle s, t\rangle$ where $s \in SimProcess$ and $t \in Time$. $EventNotice$ objects are inserted into priority queue. Hence they need an ordering relation. We use the following definition:
$e1 \leq e2 \stackrel{df}{\equiv} e1.t \leq e2.t$

The class $EventNotice$ takes the following form

```
unit EventNotice: elemPQ class(p: SimProcess, t: Time);
     unit less: virtual function(e: EventNotice): Boolean;
     begin
          result:= t ≤ e.t
     end less;
end EventNotice;
```

The full text of the second version is here.

3.4. Properties S3 and S4

We shall prove that in each state of SimulationPlan SQS and hence in each moment of a simulation experiment the following two properties hold.

$$SQS.currentProcess = (SQS.min\ qua\ EventNotice).p \tag{S3}$$

and

$$SQS.currentTime = (SQS.min\ qua\ EventNotice).t \tag{S4}$$

To assure this, we put the instruction
call $chooseProcess$;
as the last instruction in the operations: $hold, run, passivate$.
For example,

```
unit hold: procedure(dt: time);
begin
     ...
     call chooseProcess;
end hold;
```

The instruction $chooseProcess$ has to select from the priority queue SQS the eventnotice of the minimal time and to activate the process named in this eventnotice. Moreover information on the chosen process and on the time of chosen eventnotice are to be accessible as the values of function designators: $cuurrentProcess$ and $currentTime$. We achieve this by declaring private variables: $currProcess$ and $currTime$ and makking their values available through the methods $currentProcess$ and $currentTime$.

```
unit chooseProcess: procedure;
     var e: EventNotice;
begin (* value of SQS.min is the least element of priority queue *)
     e:=SQS.min qua EventNotice; (* projection qua EventNotice is needed here *)
     (* variables currTime i currProcess are private variables of the object SQS *)
     currProcess:= e.p;
     currTime := e.t;
```

```
    attach(e.p);
end chooseProcess;
```

We can not not forget to declare method $currentProcess$.

```
unit currentProcess: function: SimProcess;
begin
    result := currProcess;
end currentProcess;
```

In a similar way we declare method $currentTime$. The reader sees that properties S3 and S4 are valid. The full text of the third version is here.

3.5. Property S5

$$\neg idle(p, pq) \implies (\exists t) member((p, t), pq) \tag{S5}$$

In words, *every not suspended process is planned for certain time t*. This property requires that information whether an object s of type $SimProcess$ has an eventnotice in the $SimulationPlan$ or not were known to the object itself. The simplest way to achieve this is to put the eventnotice $\langle s, t\rangle$ in the object s. Now, the function $idle$ answers correctly by checking the value of the variable $event$.

```
unit SimProcess: elemFIFO coroutine;
    var event: EventNotice; (* make sure that, event.p = this SimProcess *)
    unit isIdle: function: Boolean;
    begin
        result := (event=none); (* not scheduled iff event = none*)
    end isIdle;
end SimProcess;
```

The full text of the fourth version is here.

3.6. Property S6

$$\forall_{pq \in SimulationPlan} \forall_{p \in Simprocess} member((p, t_1), pq) \wedge member((p, t_2), pq) \implies t_1 = t_2 \tag{S6}$$

In words, *in every simulation plan every process can be planned at most once.* Property S6 will follow from the analysis of methods $schedule, hold, run$, for they are inserting an eventnotice to the plan of simulation.

3.7. Property S7

Property S7 reads:

$$pq = c \wedge idle(p, pq) \wedge \neg terminated(p, pq) \implies [call schedule((p, t), pq)] \neg idle(p, pq) \wedge pq = insert((p, t), c) \tag{S7}$$

In words, *a suspended process can be scheduled to be reactivated at time t. Note it is the time of simulation system.*

$$terminated(p) \equiv object p executed all its instructions$$

We shall write a constructor (empty) and a thread of the coroutine $SimProcess$.

```
begin
    return; (* end of constructor, constructor is empty *)
    inner; (* it is a place for the thread of derived class *)
    finished :=true; (* thread is terminated *)
    call passivate;
    raise Error; (* at the attempt to activate a terminated simprocess object *)
end SimProcess;
```

A provisory variant of procedure $schedule$ may look as follow:

```
unit schedule: procedure(p:SimProcess, t: time);
    var ev: EventNotice;
begin
    ev := new EventNotice(p, t);
    if p.idle and not p.terminated then call SQS.insert(ev); p.event:=ev; endif;
end schedule;
```

The full text of the fifth version is here.

3.8. Properties S8 and S9

$$(pq = o \wedge \neg idle(p, pq) \wedge \neg terminated(p, pq)) \implies [call schedule((p, t), pq)](idle(p, pq) \wedge pq = insert((p, t), delete((p, ti \tag{S8}$$

A scheduled already process p *can be scheduled again for another time, this will delete an earlier event-notice for* p*.*

$$terminated(p, pq) \implies [call schedule((p, t), pq)]\{ERROR\}, \tag{S9}$$

An attempt to schedule a terminated process results in an error. Properties S8 and S9 require the correct, full version of operation $schedule$.

```
unit schedule: procedure(p:SimProcess, t: time);
    var ev: EventNotice;
begin
    if p. terminated
```

```
    then
        raise ErrorDo_not_ScheduleTerminatedProcess;
    else
        ev := new EventNotice(p, t);
        if not p.idle
        then
            call SQS.delete(p.event);
        endif;
        call SQS.insert(ev);
        p.event:=ev;
    endif;
end schedule;
```

As it is easy to observe the operation $schedule$ defined in this way satisfies properties S7 and S8. We should react when the parameter t has the value less than $currentTime$.

The full text of the sixth version is here.

3.9. Property S10

$$[\text{call } hold(t, pq)]\ \alpha \equiv [\text{call } schedule((current(pq), time + t), pq)]\ \alpha, \tag{S10}$$

for arbitrary formula α.

A hold operation suspends the current process and schedules its activation after t units of time. The property leads directly to the following body of the procedure hold.

```
    unit hold: procedure(dt: time);
    begin
        call SQS.schedule(currentProcess, currentTime+dt);
        call SQS.chooseProcess;
    end hold;
```

The first instruction causes suspension of the current simprocess' thread for dt units of time. The second instruction will choose a new active simprocess object. The full text of the seventh version is here.

3.10. Property S11

$$(current(pq) = p' \wedge \neg terminated(p, pq)) \implies \{[callrun(p, pq)]\alpha \equiv [callschedule(p, time, pq)]\alpha\} \tag{S11}$$

for arbitrary formula α.

Procedure run immediately activates the indicated simprocess p.

```
unit run: procedure(p: SimProcess);
begin
     if p.terminated
     then
          raise ErrorDo_not_ScheduleTerminatedProcess;
     endif;
     call hold(0.1); (* wstrzymaj na chwile bieżacy proces *)
     call schedule(p, currentTime);
end run;
```

The instruction call run (x) results in immediate activation of the simprocess object x.

The full text of the eigtht version is here.

3.11. Property S12

$$(p = current(pq) \wedge pq = o) \implies [\text{call } passivate(pq)](idle(p, pq) \wedge pq = delete((p, time), o)) \tag{S12}$$

Instruction passivate removes the current simprocess from the plan of simulation.

```
unit passivate: procedure;
     var s: Simprocess;
begin
     s := currentProcess;
     call SQS.delete(s.event);
     s.event :- none;
     call SQS.chooseProcess;
end passivate;
```

The full text of the ninth version is here.

3.12. Property S13

$$pq = o \implies [\text{call } cancel(p, pq)](idle(p, pq) \wedge pq = delete((p, t), o)). \tag{S13}$$

Instruction cancel applies to a scheduled, non-active simprocess.

```
unit cancel: procedure(p: SimProcess);
begin
     call SQS.delete(p.event);
     p.event :- none;
     call SQS.chooseProcess;
```

end cancel;

The full text of the tenth version is here.

3.13. Property S6 again

$$\forall_{pq \in SimulationPlan} \forall_{p \in Simprocess} member((p, t_1), pq) \wedge member((p, t_2), pq) \implies t_1 = t_2 \qquad \text{(S6)}$$

Własność ta powiada: w każdym momencie obliczeń i dla każdego procesu s, w planie symulacji nie ma dwu zdarzeń e1 i e2 planujacych wznowienie procesu s w dwu różnych chwilach. Można sprawdzić, że procedura schedule zapewnia te własność, a w konsekwencji także pozostałe procedury planowania, które sie na tej procedurze opieraja, zapewniaja te własność

3.14. Remarks

skad sie wział prior?

Unit Mainpr: SimProcess class

4. Conclusions

One immediate corollary says:

Theorem 4.1. (***on relative correctness***)
If the base class correctly implements the specification of Priority Queues, then the class Simulation correctly implements the specification SimulationSpec.

Proof:
Follows from the construction of the class Simulation □

The next question arises: are there more implementations of the specification? or all of them are equivalent? The answer is given below:

Theorem 4.2. Let C_{PQ} be a correct implementation of the priority queues. Any two correct implementations of specification of Simulation which are based on the class $C + PQ$ are isomorphic.

And the most important

Theorem 4.3. In every step of computation the active coroutine is the simprocess of the current process.

Proof:
follows from the properties S... □

Hence

References

[1] Bartol, W. M., et al.: *Report on the Loglan'82 Programming Language*, PWN, Warszawa Łódź, 1984.

[2] Conway, M.: Design of a separable transition-diagram compiler, *Communications of the ACM*, 1963.

[3] Dahl, O.-J., Myhrhaug, B., Nygaard, K.: Common Base Language (Simula67), 1970.

[4] Dahl, O.-J., Wang, A.: Coroutine sequencing in a block structured environment, *BIT*, 1971, 425–449.

[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] Mirkowska, G., Salwicki, A., Świda, O.: Verifying a Class: combining Testing and Proving, *Fundamenta Informaticae*, **95**, 2009, 305–324.

Appendix A: Specification of Priority Queues

The specification of priority queues class was published in [5], we recall it here for the convenience of the reader.

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
```

Appendix B: Coroutines

Coroutines were invented in 1963 [2] by M. Conway. Fathers of Simula67, O.-J. Dahl and K. Nygaard defined coroutines in a completely new way, see [3, 4]. The system of coroutines of Simula67 had a weak point: namely coroutines were dependent on the notion of prefixed block (a prefixed block is in Java terminology an anonymous class). In Loglan'82 [1] coroutines obtained a new and clean

definition. Nowadays coroutines attract more attention, see Wikipedia []. Astonishingly, the majority of programmers ignores the proper definition. And consequently they are ignorants of broad class of applications. For these reasons we present here a short presentation of coroutines. The diagram of the Figure 4 gives almost complete information on coroutines. The system of Loglan'82 has a notion of dynamic chain of a coroutine. Each coroutine object has a stack of activation records associated to it. At the beginning the stack contains the activation record of the thread.

Definition 4.1. Let o be an object of coroutine type. A *dynamic chain* associated with the object o contains the object. If a procedure instruction is executed in the dynamic chain of o then the dynamic chain is extended by the activation record of the called procedure. When the last instruction of the activation record is executed, the dynamic chain is reduced by the rejection of the record.

Definition 4.2. Let x points to a coroutine object. An instruction *attach(x)* has the following effect. The instruction following the instruction attach(x) is memorized in the object y - the currently executed coroutine. The dynamic chain of x is activated, the first instruction of the most recent activation record of this chain is going to be executed.