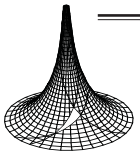

2/2 (2004), 301–317
tmcs@math.klte.hu
http://tmcs.math.klte.hu

Teaching
Mathematics and
Computer Science

Synthesis of concurrent programs

Sándor Sike and László Varga

Abstract. Students need a well defined method to be successful in the complex process of writing a concurrent program. In this paper we show a step by step method to create such programs. The method based on UML which has been thought to students during previous courses. UML provides standard and relatively simple tools to describe concurrent systems, and from the description the program can be derived.

First we give a brief introduction to the concurrent systems. This is followed by the description of the method, and finally we demonstrate the method on a small problem.

Key words and phrases: concurrent program, process, synchronization, guarded statement, semaphore, object-oriented design, UML, class diagram, state-chart diagram.

ZDM Subject Classification: D45, D55, N85, P25, P55.

1. Introduction

The task of creating concurrent programs is much more complicated and sophisticated than writing sequential programs. A concurrent program consists of processes and shared objects. The processes are sequential programs, that executes in parallel, and they use the shared objects for communication or interaction. Therefore the creation of a parallel system includes the creation of sequential programs, and in addition the interaction of the processes has to be controlled. The latter is called synchronization.

We can assume that the students are able to write sequential programs, and thus we should focus on the problem of synchronization. We have to tackle two subjects when we are synchronizing concurrent programs. First, we should be able

to form group of actions so that the execution of the group is not interrupted. This is necessary to provide the consistency of states of the given process. We call such a group of actions *atomic statement*. Second, we should be able to delay a process until the system satisfies a specified condition. This is called *condition synchronization*.

We use *guarded statements* to realize synchronization in the abstract programs, and semaphores to implement guarded statements. A guarded statement consists of a condition, called *guard*, and sequence of statements, called *body*. The execution of the body cannot be interrupted by other processes, thus the body is an atomic statement, and the guard ensures that the body is executed only when the condition is satisfied. The guard is `true` if we just want to create an atomic statement. The syntax of the guarded statement is defined as:

$$\textbf{await}\ \langle condition\rangle\ \textbf{then}\ \langle sequence\ of\ statements\rangle\ \textbf{ta};$$

where $\langle condition\rangle$ is a boolean expression and $\langle sequence\ of\ statements\rangle$ cannot contain iteration and synchronizing (waiting) statements.

Semaphore is an abstract type whose objects have two operations: **P** and **V**. The semaphore registers the difference between the number of completed **P** and the number of completed **V** operations. An obvious representation of a semaphore is an integer value. In this case the invariant of the semaphore is that the value cannot be negative, and **P** decrements and **V** increments the value. The **P** operation waits until the value is positive. The initial value should be a non-negative integer. The semaphore is called *binary semaphore* when the value can only be 0 or 1.

Non-deterministic conditional statements are used in the implementation of guarded statements. The form of this construct is:

$$\textbf{if}\ cond_1 \to S_1\ \square\ \ldots\ \square\ cond_n \to S_n\ \textbf{else}\ S_0\ \textbf{fi};$$

where $cond_i$ is a boolean expression and S_i is a sequence of statements. The **else** part is optional. When executing this statement one S_i executed whose condition is satisfied. If **else** part is included and none of the conditions is satisfied then S_0 is executed.

The formal method for synchronization presented by G.R. Andrews in [5] has been used for several years in our education. In the past few years UML has become a standard and widely used tool in software technology. In this paper we present a way to combine the formal method with object oriented modeling in UML to create a concurrent system. Using the approach described here the

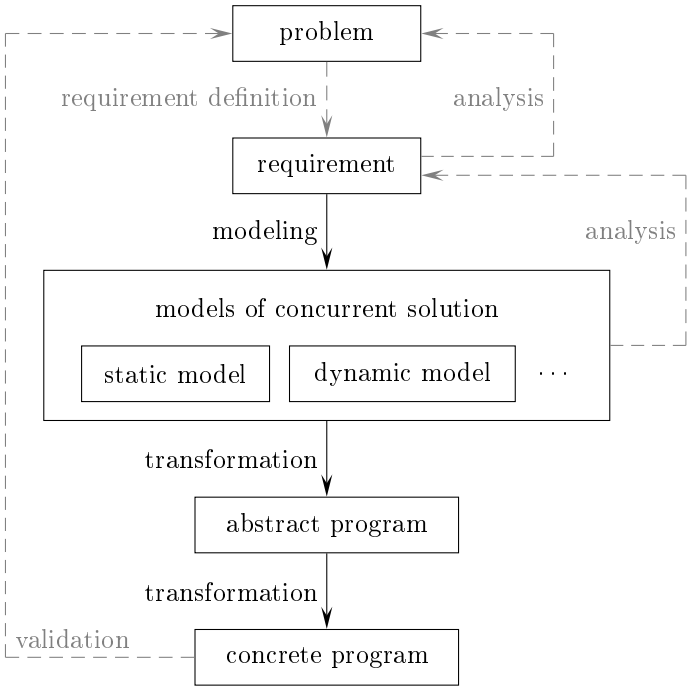

Figure 1. Development model of concurrent programs

problem of synchronization becomes less formal and more descriptive process, which is an important aspect in education.

In our method first we create the static model, class diagram, then the state-chart diagram as part of the dynamic model. The state-chart diagram is used to determine the guarded statements and to derive the abstract program. If the programming language does not support the implementation of guarded statements, then the program can be obtained by using semaphores for realizing the guarded statements.

Figure 1 illustrates the development model used by us. Activities that are involved in this type of development but are not discussed in this paper are grayed. This paper focuses on modeling of a problem given and transforming the model to a concrete program. Our belief is that the main task of creating a program is the construction of a model for the problem. The formal analysis of the model created and the validation of the final program is out of the scope of this paper.

2. A method for creating concurrent programs

We assume that the requirement definition of the system to be created is given. Then we can obtain the solution by the next procedure.

(1) Create the static model – the class diagram – of the system. Identify the processes and the common resources they use. Define the attributes of the classes. Determine the relations between the classes and the number of objects that are involved.

(2) Create the dynamic model – the state-chart diagram – of the system. The state of the system is the aggregate of the states of the resources and the processes.

- Identify the states of each process and resource. The states of a process correspond to its activities, the states of a resource provides the conditions for synchronization. Let us name as active state of a process when it uses a resource.
- If the processes have different priorities, then introduce special state(s) – e.g. requesting – for processes with higher priorities. Processes at the lowest level of priority do not need this type of state. (The preconditions of the state transitions guarantee the correct scheduling.)
- Define the invariant of each resource state. Introduce variables if necessary.
- Determine the actions for state transitions. Define the precondition of these actions. The state transitions of the processes correspond to the atomic statements in the abstract program, their precondition will be the guard of the appropriate guarded statement. The actions can be defined as the entry and exit phase of the active state(s) or request for entering the requesting state(s).
- We can ignore those actions from the state-chart diagram that do not induce state transition.

(3) Create the abstract program.

- Define the initial values of the variables introduced in the dynamic model and make the skeleton of the program as an initial assignment and the parallel execution of processes.
- Define the skeleton of each process using the state-chart diagram.

- Determine the atomic statements based on the actions of the state-chart diagram and create the appropriate guarded statements. Place these guarded statements into the skeleton of each process according to the state transitions.

(4) Create the program.

- Introduce semaphores for the implementation of guarded statements.
- Define the initial values of the semaphores.
- Implement the guarded statements with semaphores with the schemes given.
- If necessary and possible then transform the program to be more simple and effective.

The following schemes can be used in the implementation of guarded statements. Introduce an s binary semaphore to ensure mutual exclusion and set the initial value to 1.

Implement the **await** `true` **then** S_k **ta**; statement as in Figure 2 (a) where the **V**(s) statement is placed in the schedule algorithm.

Let us assume $cond_j$ ($j \in \{1, \ldots, n\}$) conditions are used in guarded statements for synchronization. We introduce a b_j semaphore and a c_j counter value for each case. The counter value specifies the number of processes waiting at the given semaphore. The initial value is 0 both for the semaphore and the counter.

In this case implement the **await** $cond_j$ **then** S_j **ta**; statement as in Figure 2 (b) and the schedule algorithm can be defined as in Figure 2 (c).

If the priority of the processes waiting at synchronization points has to be concerned then the non-deterministic conditional statement of the schedule algorithm can be replaced by a deterministic construction. Let us assume that the priorities are fixed and the order is identical of the indexing order. In this case the implementation of the schedule algorithm is shown in Figure 2 (d).

3. Case study

We will demonstrate the method described on a simple problem. Our task is to create a program that simulates the use of a computer laboratory. There is a single laboratory containing a given number of computers.

Students want to use the laboratory. The students do their studies and when they need computers they wait outside of the laboratory. If there is at least

(a)
```
P(s);
S_k;
call schedule;
```

(b)
```
P(s);
if ¬ cond_j then c_j ← c_j + 1; V(s); P(b_j) fi;
S_j;
call schedule;
```

(c)
```
if cond_1 ∧ c_1 > 0 → c_1 ← c_1 − 1; V(b_1);
□ cond_2 ∧ c_2 > 0 → c_2 ← c_2 − 1; V(b_2);
                ⋮
□ cond_n ∧ c_n > 0 → c_n ← c_n − 1; V(b_n);
else V(s) fi;
```

(d)
```
if cond_1 ∧ c_1 > 0 then c_1 ← c_1 − 1; V(b_1);
elsif cond_2 ∧ c_2 > 0 then c_2 ← c_2 − 1; V(b_2);
                ⋮
elsif cond_n ∧ c_n > 0 then c_n ← c_n − 1; V(b_n);
else V(s) fi;
```

Figure 2. The implementation schemes of the await statements and the schedule algorithm

one free computer, then a waiting student can enter and begin to use a computer. After completing the task the student leaves the laboratory and continues his/her activities, i.e., studies and waits to use a computer in the laboratory again.

The computers in the laboratory are maintained by crew members. Only one crew member can maintain the computers at the same time, but any number of crew members can request maintenance simultaneously. During the maintenance no student can use the computers. If a crew member specify a maintenance request, no student can enter the laboratory, he/she must wait until the maintenance is complete. The maintenance can start when all the students inside has left the laboratory.

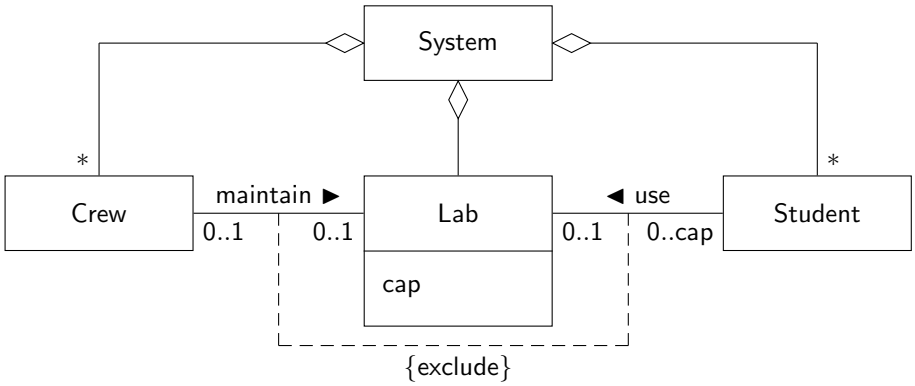

Figure 3. The class diagram of the system

3.1. Static model

We can identify the following classes from the description given.

System: a class corresponding to the whole system modeled.

Lab: a class of computer laboratory. It has an argument (cap), that specifies the number of computers in the laboratory.

Crew: the class of crew members.

Student: the class of students.

The relations between the classes are the next ones.

- The System class is the aggregate of the other three classes. The multiplicity factor is one for the laboratory, and arbitrary for the Crew and Student classes.
- The Student class is associated with the Lab class, the name of the association is use. The number of students involved in this association can vary between 0 and cap.
- The Crew class is associated with the Lab class, the name of the relation is maintain. There is at most one crew member involved in this association.

The use and maintain relations are exclusive ones, which means that only one of them can exist at the same time. This can be described in the class diagram by using a constraint. The result of our analysis is the class diagram of Figure 3.

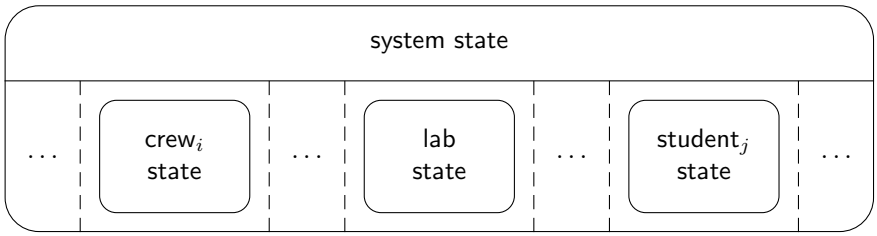

Figure 4. The state-chart diagram of the system

3.2. Dynamic model

The state of our system is determined by the state of the crew members, the state of the laboratory and the state of the students (Figure 4). The students and the crew members are the processes of our system, thus their states correspond to the actual activities of the processes. The state of the laboratory provides the conditions for synchronization.

Each student can have two states.

- The student does some activity or waits outside the laboratory. Let be the name of this state: **do** / wait.
- The student uses one computer in the laboratory. Let be the name of this state: **do** / use.

The state transitions are induced by the entry and the exit phase of the use state, i.e., use.ent and use.ex.

The crew members can have three states.

- The person is working somewhere else. Let be the name of this state is: **do** / work.
- The person requests maintenance in the laboratory. The name of the state is: requesting. (This is a passive state, the person waits to be scheduled.)
- The person is maintaining the computers in the laboratory. The name of this state is: **do** / maintain.

The request action changes the state from work to requesting. The entry phase of maintenance (maintain.ent) connects states requesting and maintain. The exit phase of maintenance (maintain.ex) leads from maintain to work.

The state of the laboratory is determined by the number of students inside and the activities of the crew members. Let us introduce variable t specifying the number of students inside, variable r determining the number of crew members

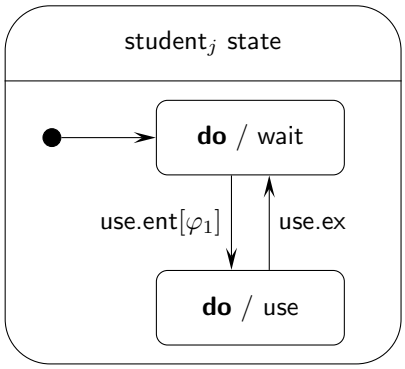

Figure 5. The state-chart diagram of a student

requesting maintenance and variable w for the number of crew members maintaining the laboratory. Three states can be identified considering the students.

- No student uses the laboratory. The name is empty and the invariant is $\mathsf{t} = 0$.
- There is no free computer, the laboratory is full. The name is full and the invariant is $\mathsf{t} = \mathsf{cap}$.
- The laboratory is in the state between. The name is normal and the invariant is $0 < \mathsf{t} < \mathsf{cap}$.

Three states can be identified in connection with the crew members.

- Nobody requests maintenance and nobody maintains the laboratory. The name is OK and the invariant is $\mathsf{r} = 0 \wedge \mathsf{w} = 0$.
- Maintenance is requested and nobody maintains the laboratory. The name is requested and the invariant is $\mathsf{r} > 0 \wedge \mathsf{w} = 0$.
- The laboratory is under maintenance. The name is maintained and the invariant is $\mathsf{w} = 1$.

Now the conditions of the state transitions can be specified. (The first two conditions can be rephrased with variables using the invariants.)

- A student can start to use a computer if the laboratory is not full and is in state OK. $\varphi_1 : \neg\, in\ \mathsf{full} \wedge in\ \mathsf{OK}$.
- A crew member can start the maintenance if the laboratory is empty and nobody else maintains it. $\varphi_2 : in\ \mathsf{empty} \wedge \neg\, in\ \mathsf{maintained}$.
- The laboratory becomes full after a student begins using a computer and there has been only one free computer. $\varphi_3 : \mathsf{t} = \mathsf{cap} - 1$.

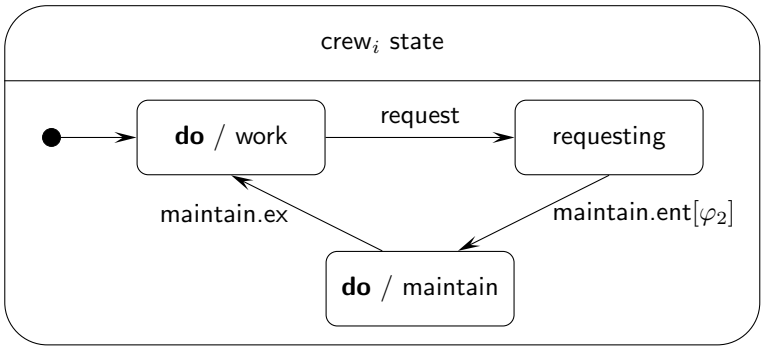

Figure 6. The state-chart diagram of a crew member

- The laboratory becomes empty after the last student leaves it. $\varphi_4 : \mathsf{t} = 1$.
- The state of the laboratory will be requested after a maintenance is finished and there are other crew members requesting maintenance. $\varphi_5 : \mathsf{r} > 0$.
- The state of the laboratory will be OK after a maintenance is finished and nobody else requests maintenance. $\varphi_6 : \mathsf{r} = 0$.

From the conditions above, φ_1 and φ_2 provide synchronization conditions for the processes.

The state-chart diagram for a student is shown in Figure 5, the state-chart diagram for a crew member can be seen in Figure 6, and the states of the laboratory are defined in Figure 7. The actions that do not change the state of the laboratory are not shown in the diagram.

3.3. Abstract program

We have already introduced the three variables for describing the states of our problem in the dynamic model. These variables are:

- t : the number of students using computers,
- r : the number of crew members requesting maintenance,
- w : the number of crew members maintaining the laboratory.

The initial values of the variables defined as:

$$\mathsf{w} = 0 \wedge \mathsf{r} = 0 \wedge \mathsf{t} = 0.$$

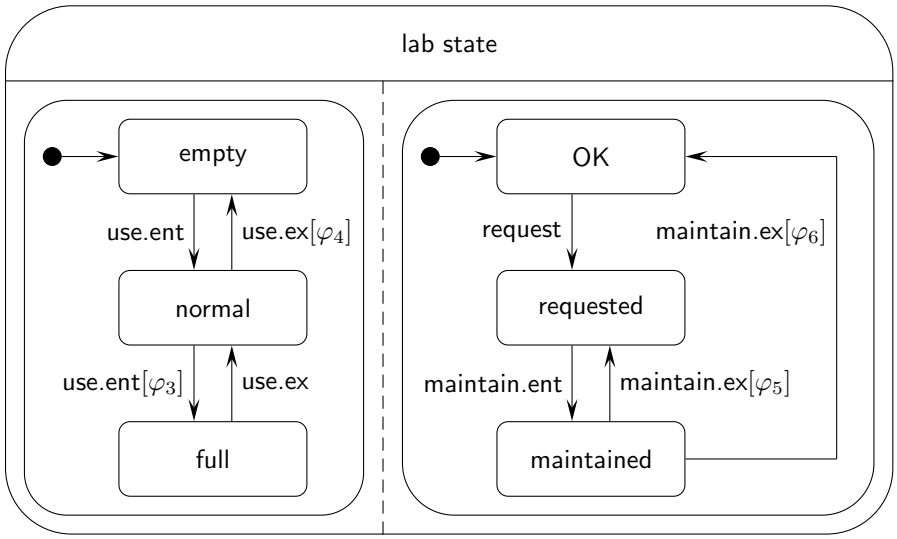

Figure 7. The state-chart diagram of the laboratory

```
w ← 0; r ← 0; t ← 0
parbegin
        crew_1; || ... || crew_n; ||
        student_1; || ... || student_m;
parend
```

Figure 8. The skeleton of the program

Let us assume that the actual number of crew members is n, and the actual number of students is m. Then the skeleton of the program is shown in Figure 8 and Figure 9 contains the skeletons of the processes.

The atomic statements and their corresponding actions in the state-chart diagram are shown in Figure 10. In the abstract program we realize the atomic statements with guarded statements, **await**. The guards of these statements ensure the correct scheduling. The precondition of the corresponding action in the state-chart diagram determines the guard of the **await** statement. The conditions can be defined by the variables introduced if the states are replaced by their invariants. The third column in Figure 10 contains the guards for the atomic statements.

```
crew_i:
    while true do
        work;
        request;
        maintain;
    od
```

```
student_j:
    while true do
        wait;
        use;
    od
```

Figure 9. The skeletons of the processes

atomic statement	action	guard
$\langle w \leftarrow w - 1 \rangle$	maintain.ex	true
$\langle r \leftarrow r + 1 \rangle$	request	true
$\langle r \leftarrow r - 1;\ w \leftarrow w + 1 \rangle$	maintain.ent	$w = 0 \wedge t = 0$
$\langle t \leftarrow t + 1 \rangle$	use.ent	$t \neq cap \wedge w = 0 \wedge r = 0$
$\langle t \leftarrow t - 1 \rangle$	use.ex	true

Figure 10. The atomic statements of the program

The result of the transformation applied to the skeleton of the processes can be seen in Figure 11. We can have the abstract program by inserting these processes into the skeleton of the program in Figure 8.

3.4. Program

Semaphores have to be introduced to implement the guarded statements. We need one semaphore, s, to ensure the integrity of the execution of each atomic statement. We have two guarding conditions that need semaphores and counters:

condition:	semaphore:	counter:
$w = 0 \wedge t = 0$	bw	cw
$t \neq cap \wedge w = 0 \wedge r = 0$	bt	ct.

We implement the guarded statements with these semaphores as described in the method given. The processes transformed and the shedule algorithm is shown in Figure 12. The deterministic scheme is used because the priorities are different for the maintenance and the use activities.

We can derive the program from the abstract program by assigning the appropriate initial values to the semaphores and the counters. The initial value of semaphore s should be set to 1, since we should allow the execution of the first

```
crew_i:
    while true do
        work;
        await true then r ← r + 1 ta;
        await w = 0 ∧ t = 0 then r ← r − 1; w ← w + 1 ta
        maintain;
        await true then w ← w − 1 ta;
    od
```

```
student_j:
    while true do
        wait;
        await t ≠ cap ∧ w = 0 ∧ r = 0 then t ← t + 1 ta;
        use;
        await true then t ← t − 1 ta;
    od
```

Figure 11. The abstract processes

P(s) statement. For the other semaphores and counters the initial value should be set to 0. The reason for this choice is obvious for the counters, and we should wait at the first **P** statement when bw and bt semaphores are used. Inserting the assignments required to the skeleton of the abstract program we get the main program (Figure 13).

The processes can be transformed to be more simple and effective. We used the general scheme to implement the guarded statement. When we implement the **await** true **then** r ← r + 1 **ta**; guarded statement, the schedule algorithm can be replaced by a simple **V**(s) statement. If the crew$_i$ process is rewritten then it consists a **V**(s); **P**(s); sequence, that can be removed.

Note, that after these changes the values r and cw are identical outside of the atomic statements. (This is not surprising, since the meaning of them is the same, both values specify the number of crew members waiting to maintain the laboratory.) Thus only one of them is necessary. We will use cw, and remove r from the program. (In the main program it should be deleted from the initial assignment part.)

The first schedule algorithm in the crew$_i$ process after w ← w + 1; assignment can be replaced by **V**(s); statement because the other cases in the schedule

```
crew_i:
    while true do
        work;
        P(s);
        r ← r + 1;
        call schedule;
        P(s);
        if w ≠ 0 ∨ t ≠ 0 then cw ← cw + 1; V(s); P(bw) fi;
        r ← r − 1; w ← w + 1;
        call schedule;
        maintain;
        P(s);
        w ← w − 1;
        call schedule;
    od
```

```
student_j:
    while true do
        wait;
        P(s);
        if t = cap ∨ w ≠ 0 ∨ r ≠ 0 then ct ← ct + 1; V(s); P(bt) fi;
        t ← t + 1;
        call schedule;
        use;
        P(s);
        t ← t − 1;
        call schedule;
    od
```

```
schedule:
    if w = 0 ∧ t = 0 ∧ cw > 0 then cw ← cw − 1; V(bw);
    elsif t ≠ cap ∧ w = 0 ∧ r = 0 ∧ ct > 0 then ct ← ct − 1; V(bt);
    else V(s) fi;
```

Figure 12. The processes and the schedule algorithm

algorithm contain the $w = 0$ condition, and this cannot be satisfied after this assignment.

```
s ← 1; bw ← 0; bt ← 0; cw ← 0; ct ← 0; w ← 0; r ← 0; t ← 0
parbegin
    crew_1; || ... || crew_n; ||
    student_1; || ... || student_m;
parend
```

Figure 13. The main program

The second schedule algorithm can be replaced by a similar conditional statement where the conditions can be simplified using the fact, that in this case w must be 0 since the previous statement decreases the value of w. Thus the $w = 0$ component can be left out from the conjunctions. In this case the number of students in the laboratory must be 0 (see the condition of incrementing w for formal checking), thus $t = 0$ and $t \neq cap$ conditions are satisfied and can also be removed.

Let us check the first schedule algorithm in $student_j$ process after $t \leftarrow t + 1$; assignment. We know that t is not 0, thus the condition of the first case in schedule cannot be true. The synchronization ensures that $w = 0 \wedge cw = 0$ at this point, therefore this part of the second condition can be left out.

Finally let us examine the second schedule algorithm used in $student_j$ process. This is placed after $t \leftarrow t - 1$; assignment, thus $t \neq cap$ must be satisfied and can be removed from the second case. The synchronization guarantees that $w = 0$, thus we can leave this out from both conditions.

The result of the transformations described is shown in Figure 14.

4. Conclusion

Our experience shows that the students can handle the problems of creating parallel programs much better using the method described. They can solve problems successfully with this method even when they would fail to do so with conventional approaches such as using invariants at synchronization points. The reason can be that a well known and more descriptive tool, UML, is used for modeling and defining synchronization. The model can be almost automatically transformed to a program with guarded statements, and using the schemes the guarded statements can be implemented with semaphores. In addition, from the

```
crew_i:
    while true do
        work;
        P(s);
        if w ≠ 0 ∨ t ≠ 0 then cw ← cw + 1; V(s); P(bw) fi;
        w ← w + 1;
        V(s);
        maintain;
        P(s);
        w ← w − 1;
        if cw > 0 then cw ← cw − 1; V(bw);
        elsif cw = 0 ∧ ct > 0 then ct ← ct − 1; V(bt);
        else V(s) fi;
    od
```

```
student_j:
    while true do
        wait;
        P(s);
        if t = cap ∨ w ≠ 0 ∨ cw ≠ 0 then ct ← ct + 1; V(s); P(bt) fi;
        t ← t + 1;
        if t ≠ cap ∧ ct > 0 then ct ← ct − 1; V(bt);
        else V(s) fi;
        use;
        P(s);
        t ← t − 1;
        if t = 0 ∧ cw > 0 then cw ← cw − 1; V(bw);
        elsif cw = 0 ∧ ct > 0 then ct ← ct − 1; V(bt);
        else V(s) fi;
    od
```

Figure 14. The final form of the processes

UML model the invariants for a formal proof can be also determined using the invariants of the states in the state-chart diagram.

References

[1] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, 1973.

[2] E. W. Dijkstra, *A tutorial on the split binary semaphore*, EWD703, Nuenen, Netherlands, 1979.

[3] E. W. Dijkstra, *The superfluity of the general semaphore*, EWD734, Nuenen, Netherlands, 1980.

[4] G. Booch, J. Rumbaugh and I. Jakobson, *The Unified Modeling Language User Guide*, Addison-Wesley Longman, Inc., 1999.

[5] G. R. Andrews, A method for solving synchronization problems, *Science of Computer Programming* **13** (1989/90), 1–21.

[6] K. R. Apt and E-R. Olderog, *Verification of Sequential and Concurrent Program*, Springer-Verlag, 1997.

[7] W-P. de Roever *et al.*, *Concurrency Verification; Introduction to Compositional and Noncompositional methods*, Cambridge University Press, 2001.

[8] Sike Sándor and Varga László, *Szoftvertechnológia és UML*, (Második, bővített kiadás), ELTE Eötvös Kiadó, 2003.

[9] Kozma László and Varga László, *A szoftvertechnológia elméleti kérdései*, ELTE Eötvös Kiadó, 2003.

SÁNDOR SIKE
DEPARTMENT OF SOFTWARE TECHNOLOGY AND METHODOLOGY
FACULTY OF INFORMATICS
EÖTVÖS LORÁND UNIVERSITY, BUDAPEST
PÁZMÁNY PÉTER S. 1/C
H-1117 BUDAPEST
HUNGARY

E-mail: `sike@inf.elte.hu`

LÁSZLÓ VARGA
DEPARTMENT OF SOFTWARE TECHNOLOGY AND METHODOLOGY
FACULTY OF INFORMATICS
EÖTVÖS LORÁND UNIVERSITY, BUDAPEST
PÁZMÁNY PÉTER S. 1/C
H-1117 BUDAPEST
HUNGARY

E-mail: `varga@ludens.elte.hu`

(Received April, 2004)