Annales Univ. Sci. Budapest., Sect. Comp. 24 (2004) 79-102

IMPLEMENTATIONS OF SYNCHRONIZATION OF CONCURRENT OBJECTS

B. Ugron, L. Kozma, Sz. Hajdara and L. Blum
(Budapest, Hungary)

Dedicated to Professor Dr. Imre Kátai on his 65th birthday

Abstract. There are many implementation problems in context of synchronization of parallel systems. We are going to discuss two temporal logic based approaches of the specification and implementation of synchronization. The first method will have PTPTL based specification that will be implemented directly. The second method has an MPCTL* temporal logic specification which an abstract code (an automata) will be synthesized from, and the automata will be implemented.

1. Introduction

The design and construction of the co-operation and synchronization of concurrent objects is a difficult task. There are several techniques for specifying synchronization properties of concurrent processes [8], [16], [20], [22], etc. The approaches taken generally fall into one of two categories:

- the first group are procedural mechanisms which combine synchronization primitives with sequential flow control constructs and data structures;
- the second group are declarative mechanisms where the programmer simply specifies the sychronization policy desired.

We are going to consider implementation problems of synchronization. Two different approaches will be discussed. Both of the methods are based

The research was supported by GVOP - 3.2.2 - 2004 - 07 - 005/3.0.

on temporal logic tools, and the implementation can be generated from the temporal logic specification.

One of the well known implementation problems is inheritance anomalies which may arise while re-using the synchronization code. The first approach tries to handle the inheritance anomalies by a scheme based on Past-Time Propositional Temporal Logic (PTPTL), and it gives an implementation for the synchronization scheme introduced in [6].

Another well known implementation problem arises in most cases of program synthesis, when the number of the components of the system is growing. This issue is called state explosion problem. The second approach handles the state explosion problem, and gives a concrete Java implementation of the synchronization. This approach is a program synthesis based on a Branching Time Temporal Logic, the Many Processes Computational Tree Logic (MPCTL*).

2. Implementation based on PTPTL

In the case of the application of different synchronization schemes, during the reuse of the code different difficulties may arise, which are called inheritance anomalies in the literature. Generally, we talk about inheritance anomaly if some kind of difficulty arises from the re-using of the synchronization code. We can find several solutions for getting rid of the inheritance anomalies in [18]. The solutions are based on the fact that the occurrence of the inheritance anomalies depends on the applied synchronization schemes. Using only one synchronization scheme, anomalies can occur easily, while changing the schemes, they can be avoided. The localization of the synchronization code and scheme to the given object gives an opportunity for this. Thus we can apply a completely different synchronization scheme in the sub-class than in the parent class. The distribution of the synchronization code between the objects can be done similarly to the inheritance of the methods. The above purpose can be reached for example with the use of synchronizers and transition specifications as synchronization schemes.

2.1. The object model

In our paper, we shall refer to the reflective model of objects of the kind described below. In the reflective model, [23] every object consists of the (recursive) composition of four objects: Meta Object, Container Object, Processor Object, Mailbox Object.

- the Meta Object manages the three other objects,
- the Container Object stores the acquaintances of the object,
- the Processor Object can change the state of the object upon receiving an enabled message from another object,
- the Mailbox Object stores asynchronously received messages (requests for method-executions) from other objects and synchronizes the object: uses a policy for choosing the next enabled message.

When the object executes a request, the Processor Object will be blocked, and no other request can be enabled until the execution has been done.

2.2. Assigning temporal logical atomic formulas to actions

In our model, like in [4], a truth-value for each atomic formula will be given to every method request and method execution. For a method *m1*, *m1* means both the name of the method and an atomic formula having a truth-value that corresponds to the execution state of the method in each time-point. In addition, we introduce an atomic formula *req_m1*, which describes whether there has been a request for the method *m1* or not. An atomic formula *req_m1* is true while there is a request for method *m1* in the Mailbox. A temporal formula of a method expressing that it can be executed if there is no request for the method *m3* and the previously executed one was the method *m4*, is the following

$$\neg req\_m3 \wedge \bullet m4.$$

Since we want to use temporal expressions for synchronization, we have to define the time-points of the Kripke-structure of the object. Since only past-time temporal operators are used, it is enough to build up a Kripke-structure up to the present. Taking this into account, the next time-point to the Kripke-structure of an object is given when a request for executing a method is satisfied. The period between two executions can be viewed as a container period, during which the atomic formulas assigned to messages are given a value representing the next time-point.

2.3. Choosing requests from the Mailbox

In our model, the Meta Object of an object tries to send a request to the Mailbox to get the next accepted request for a method after every request-execution or arrival of a new message. If successful, it has the Processor Object to execute the request.

2.4. A synchronization-scheme

Further in this section, a synchronization scheme will be used which is presented in [6]. The abstraction level of the scheme is rather high, because it uses temporal logical formulas for synchronization. The scheme is an extension of the well-known guarded methods, where the constraints of a method are collected in a set so that they can be expanded when inherited. Using the scheme, the state modification and the state partitioning anomalies [19] can be resolved and the history-only sensitiveness anomalies can be radically reduced.

2.5. Past-Time Propositional Temporal Logic (PTPTL)

In the model, PTPTL formulas are used to give constraints for the method-executions. Past-time operators of PTPTL are similar to those used by [4], extended with operators *atprev*, *punless*, *pwhile* and *after*. For a Kripke-structure K [15] and a time-point i, the semantics of the operators can be defined in the following way:

$K_i(a \text{ atprev } b) = t$ iff the greatest $j < i$ where $K_j(b) = t$, $K_j(a) = t$.

$K_i(a \text{ punless } b) = t$ iff $K_j(b) = t$ for some $j < i$ and $K_k(a) = t$
$\forall k : j < k < i$ or $K_l(a) = t \; \forall k : 0 \leq k < i$.

$K_i(a \text{ pwhile } b) = t$ iff $K_j(b) = f$ for some $j < i$ and $K_k(a) = K_k(b) = t$
$\forall k : j < k < i$.

$K_i(a \text{ after } b) = t$ iff $\forall K_j(b) = t$ $(j < i)$ there is $j < k < i$ that $K_k(a) = t$.

2.6. Recursive expressibility of temporal operators

PTPTL operators are expressed by the equivalencies:

$$\blacksquare a \equiv a \wedge \blacklozenge\blacksquare a,$$

$$\blacklozenge a \equiv a \vee \bullet\blacklozenge a,$$

$$a \text{ atprev } b \equiv \blacklozenge(b \rightarrow a) \wedge (\blacklozenge b \vee \blacklozenge(a \text{ atprev } b)),$$

$$a \text{ punless } b \equiv \blacklozenge b \vee (\blacklozenge a \wedge \blacklozenge(a \text{ punless } b)),$$

$$a \text{ pwhile } b \equiv \neg \blacklozenge b \vee (\blacklozenge a \wedge \blacklozenge(a \text{ pwhile } b)),$$

$$a \text{ after } b \equiv \neg \blacklozenge b \wedge (\blacklozenge a \vee (a \text{ after } b)).$$

2.7. Synchronization sets

In the model developed, the key structure is a set consisting of the elements (called synchronization element),

[*method_name*, *tf_set*],

where *method_name* denotes a method of the object and *tf_set* is a set of PTPTL formulas. Every object has exactly one synchronization set, by which the methods of the object can be synchronized.

Definition 1. *A set* tf_set *of PTPTL formulas is called true at a time-point of a Kripke-structure, if each formula in the set is true.*

Taking an object O with a synchronization set S, a request *req_method1* in the Mailbox may be satisfied:

- if there is an element in S the method of which is method1 and its formula set is true,
- if there is no element in S the method of which is method1.

2.8. Operations with synchronization sets

In the case of inheritance, the synchronization set of the descendant objects can be established by using the following operators on synchronization sets:

Let S_1 and S_2 be two synchronization sets and MS is a set of method names and *tf_set* is a set of PTPTL formulas, then

a) $S_1 + S_2$ means the union of sets S_1 and S_2,

b) $S_1{+}{+}[MS,$ *tf_set*$]$ denotes the union of the formula sets of the elements of S_1, that first elements are members of MS. If MS contains all of the methods of S_1, then this may be denoted in short by $S_1{*}{+}{+}$*tf_set*.

For example, let

$$S_1 = \{[\text{method1}, \{\text{form1}, \text{form2}\}], [\text{method2}, \{\text{form3}\}]\},$$

$$S_1 + +[\{\text{method1}\}, \{\text{form4}\}] =$$

$$= \{[\text{method1}, \{\text{form1}, \text{form2}, \text{form4}\}], [\text{method2}, \{\text{form3}\}]\}.$$

Applying the operator +, the descendant object can revise the synchronization set of the ancestor object by adding constraints to the methods that are independent of those used in the ancestor object. With the operator ++, more constraints can be added to the constraints of the ancestor.

Remark 1. *Combining operators + and ++, we can absolutely revise the constraints inherited from the ancestor, if we use the operator ++ with set* $\{False\}$ *and establish the new constraints using operator +.*

2.9. The History-Only Sensitiveness Anomaly can be eliminated

We would like to create a new class *Democ* and its descendant *DesDemoc* with the same methods and with the same synchronization, but in addition m_3 is enabled if and only if m_2 and m_1 were last requested at the same time. The example can be seen in Figure 1.

```
Class Democ: ACTOR {
public:
  void Democ() {...}
  void m1() {...}
  void m2() {...}
  void m3() {...}
synchset:
  DemocS = {...}
}
Class DesDemoc: Democ {
public:
  void DesDemoc() {...}
  void m3() {...}
synchset:
  DesDemocS =
    DemocS++[{m3},
    {♦ req_m1 ∧ ((req_m1 ∧ req_m2) atprev (req_m1 ∨ req_m2))}]
}
```

Figure 1. Class Democ with some methods and class DesDemoc

2.10. The implementation framework

In order to distinguish the synchronization code from the implementation code, two new synchronization methods are introduced that work "under the hood": the first method (called *synch_init*) to initialize the synchronization variables, the second (called *synch_change*) to change the synchronization variables to the next value (using the rules described below) after the method

has been executed. In the synchronization part only these two methods are permitted to change the values of the synchronization variables. The evaluation of the formulas is done using the synchronization variables. Synchronization methods are assigned to each PTPTL formula contained in the synchronization set of the object, to give the proper value of the formula. The Meta Object can accept requests by evaluating the formula sets of the requested method. (See Figure 2.)

To establish how many synchronization variables we need and how the variables correlate with each other, a graph (called temporal graph) will be created. The methods and variables referring to the state of the object are in the Processor Object together with the synchronization and evaluation methods and extra variables to evaluate temporal formulas.

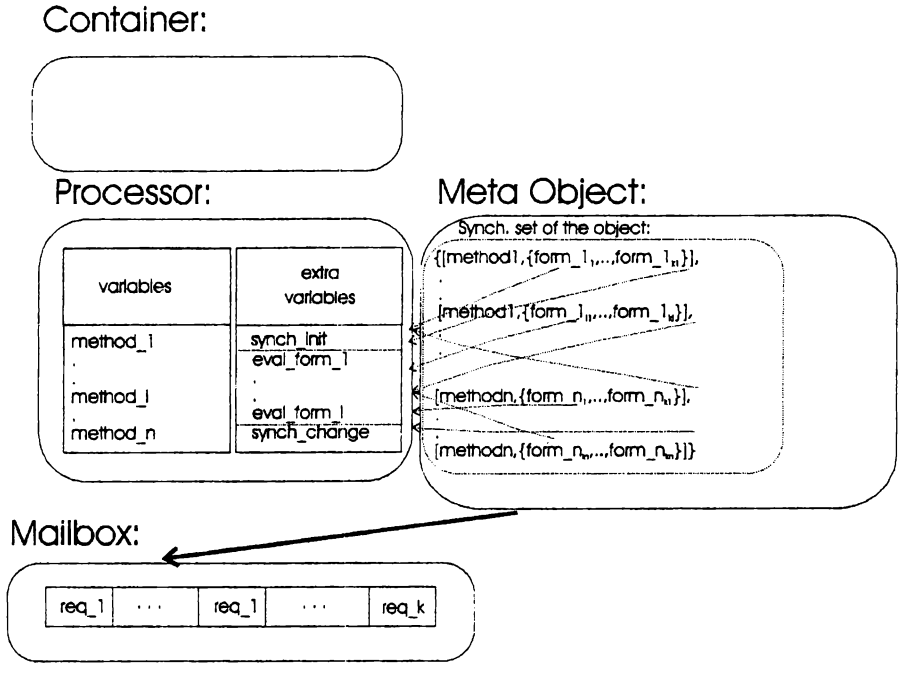

Figure 2. The implementation framework

2.11. Classification of PTPTL operators

The PTPTL operators can be separated into three groups are the following:

1. recursively expressible operators: ■, ♦, atprev, punless, pwhile, after,
2. operators using only the previous time-point: ●, ◗,
3. non-temporal operators: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$.

Using the previously given identities, graph primitives can be created to build up the temporal graph of a formula. In the graph, different levels can be distinguished that refer to different time-points. Each node represents a PTPTL formula.

2.12. Graph primitives for building up temporal graphs

In the temporal graph, edges and nodes are used to decompose formulas into sub-formulas in the previous- and present time-points. There are two kinds of edges in the graph primitives defined below:

1. previous time-point edge is a directed edge pointing to a node labelled by a sub-formula representing the previous time-point,
2. component edge is denoted by a dotted line pointing to a node labelled by a sub-formula evaluated in the same time-point as the ancestor.

The nodes can be rounded boxes for "weak", boxes for "strong" and dashed boxes for "undefined" nodes. The "weak" nodes are used in formulas derived from the sub-formulas bounded by the weak previous temporal operator at the ancestor node. The "strong" nodes are used in formulas derived from the sub-formulas bounded by the previous temporal operator at the ancestor node. The "undefined" may be categorised at the creation of the temporal graph with respect to the node it is derived from. These distinctions of nodes are important in giving initial values to temporal variables.

Concerning the recursive expressibility of temporal operator equivalencies, graph primitives can be defined for each type of temporal operators as it can be seen in Figure 3.

The graph primitives for non-temporal operators can be seen in the Figure 4.

These graph primitives are used to calculate the value of the formula of the given type. For example graph d. means that the formula a pwhile b can be evaluated if the previous values of the formulas b, a and a pwhile b are given, that is stored in three variables. If the previous value of b, a and a pwhile b are stored in variables v1, v2 and v3 respectively, then a pwhile $b = \neg v1 \vee (v2 \wedge v3)$

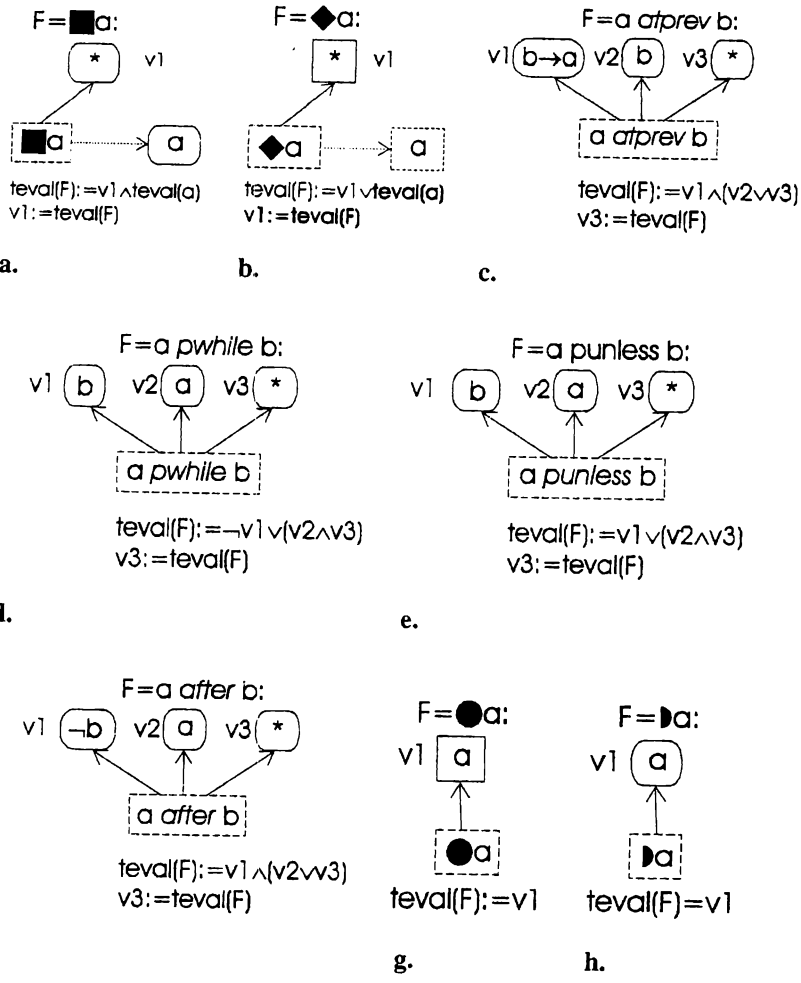

Figure 3. The graph primitives

and v3 is set to the value of a pwhile b, storing the value of this formula for the evaluation at the next time-point. An evaluating function (called teval) is defined for each graph primitive to get the correct values of the formulas. After evaluation, the proper variables are also set. In addition to the rules described above, one more rule has to be defined for function teval:

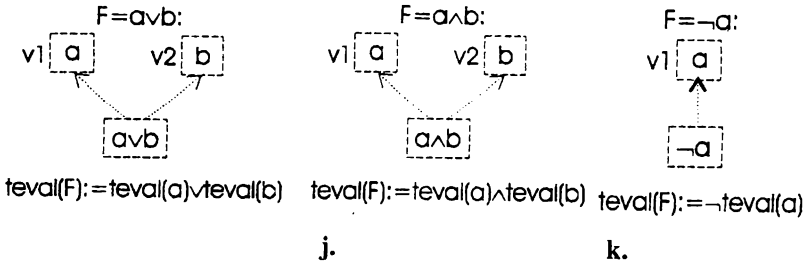

Figure 4. The graph primitives for non-temporal operators

If F is a PTPTL formula then, $\text{teval}(F) = F$ iff F is a formula containing no temporal operator.

2.13. Creating the temporal graph

Let F be the formula that the temporal graph will be created for.

Definition 2. *Expansion for a node means that a rule from a.-k. is chosen with respect to the type of the PTPTL formula of the node and the node is replaced by the primitive graph of the rule. The type of the "undefined" new node is inherited from the node expanded. During the expansion, the evaluation rules are also prescribed.*

The algorithm will build the temporal graph with these steps:

1. let F be the root node of the graph.
2. expand iteratively all the leave-nodes without a star until no leave-node contains temporal operators.

Definition 3. *In the temporal graph of F, a node is said to be in the past if it can be reached from formula F touching at least one previous time-point edge.*

2.14. Using temporal graphs

In the temporal graph, new variables are assigned to each node in the past that store values of the formulas of the nodes. Each node represents a sub-formula of formula F derived by the rules above. Each leave-node either represents sub-formulas of F containing only previous or weak previous

temporal operators or is a * node (see Figure 5). We can see at an expansion that three new variables are introduced for the node representing a binary and one new variable for unary PTPTL formula. At the worst, when every temporal operator in the expression is binary, the number of new variables is three times larger than the number of temporal expressions.

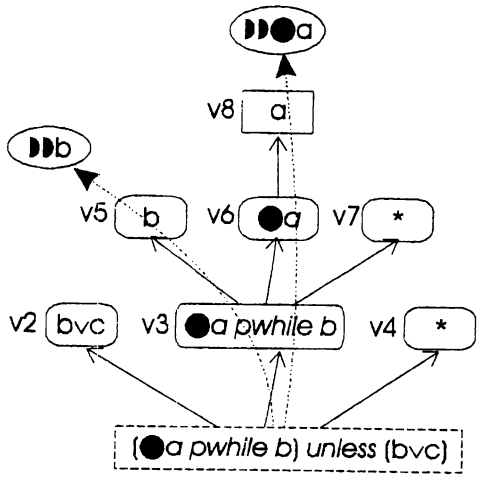

Figure 5. Looking at the root node as a starting point, the variables v5, v8 store the values of the temporal sub-formulas of the root only weak or strong previous operators

2.15. Initialization

When an object is created, the *synch_init* method is called to set the required variables. The initial value of a variable is true, if its node is "weak", and false, if its node is "strong", and undefined if its node is "undefined".

2.16. Changing the values of variables after a method has been executed in the object

In the above case the *synch_change* method is called. The new value of a variable can be specified by the rules created by the temporal graph. If a variable is assigned to a node, the value of the formula of the node is passed to the variable. The order of changing the values of the variables is specified by the order of the creation of the nodes of the temporal graph.

2.17. Evaluating the temporal formula

A PTPTL formula can be easily evaluated at the present without changing the synchronization variables. In our model, a temporal formula can change its present value, because requests for methods can change the values of atomic formulas belonging to them (see Figure 6). But this change is not confusing, since most of the temporal operators are affected by the past time-points and the values of atomic formulas at the past time-points are unambiguous. We can view it as a trying period of a time-point, where we can test how the formulas are changed by atomic formulas. When one of our requests is satisfied, we view it as a new time-point to which the values set during the period belong.

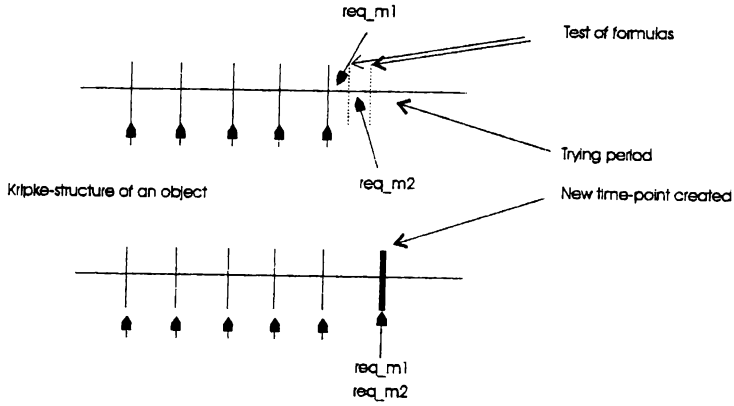

Figure 6. Assigning atomic formulas to a new time point

2.18. Creating a temporal graph for a formula – an example

Consider the formula and let us build its temporal graph, as it can be seen in Figure 7.

Creating the object containing this formula in its synchronization the new variables must be initialized the following way: let variables v1, v2, v3, v4, v5, v6, v7 be *true* and variable v8 be *false*. After executing a method in the object, the groups (1), (2), (3) (see Figure 7) are executed in the given order, for refreshing variables. Refreshing the temporal variables we can evaluate the formula.

Definition 4. *The depth of a temporal graph is the maximum number of temporal edges contained in a path from the root.*

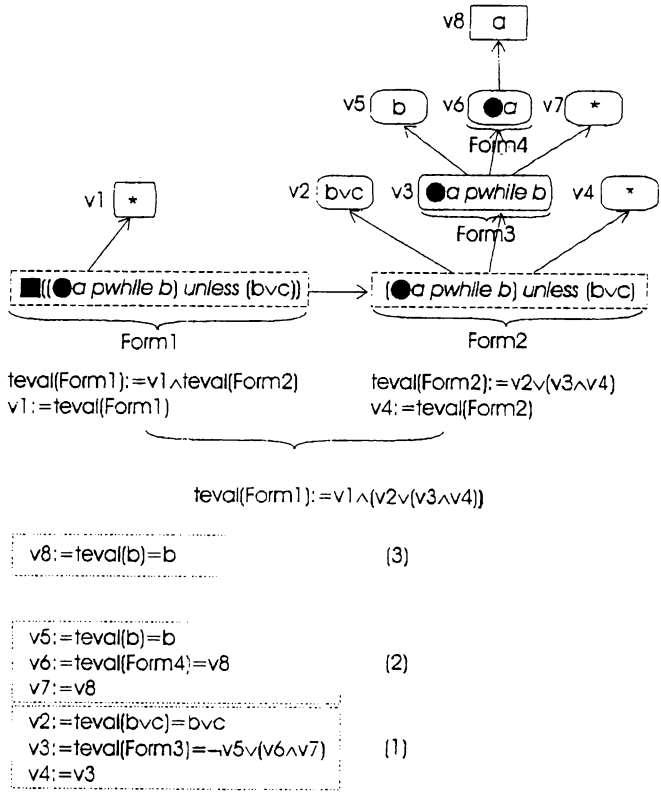

Figure 7. The temporal graph of formula ■ ((•a pwhile b) unless ($b \vee c$)) prescribing the synchronization variables and evaluating the *teval* function

2.19. The correctness of the algorithm

All we have to prove is that the teval function gets a correct prescription for each node, and all the variables store the correct values referring to the formulas they belong to. The proof can be carried out by structural induction by the depth of the temporal graph.

3. Implementation based on MPCTL* specification

Synchronization code and real computation code can be separated in the case of most parallel programs. If so, the synchronization part of the program can be specified separately and the synchronization code can be generated from the specification. Other synchronization techniques can be seen for instance in [25] and [30].

The synthesized system of K similar objects is a mechanically constructed correct solution of a precise problem specification given by MPCTL* (Many-Process CTL*) formulas. K is an arbitrary large natural number and an MPCTL* formula consists of a spatial modality followed by a CTL* state formula over uniformly indexed family of atomic propositions.

The method used in this paper applies the technique suggested by P.C. Attie and E.A. Emerson in [34], and it inherits an important advantage of their method, namely how to deal with an arbitrary number of similar objects without incurring the exponential overhead due to the state explosion problem.

To use the method developed by P.C. Attie end E.A. Emerson in [34], we had to solve the problem of handling the shared variables of the similar objects. The details can be found in [36].

3.1. The task

We show the method through an example of a simulation program of a surgery.

Given a surgery, which is accepting patients. Patients can be infectious or non infectious. The doctor suggests that the patients, who think they are infectious, should not stay in the waiting-room if there is some other person in the room, and if there is an infectious patient in the waiting-room then the other patients should stay outside in the bright spring sunshine until the infectious patient in the waiting-room leaves. For the sake of simplicity we do not consider that patients can stay in the surgery, too, we only consider the synchronization of the the patients in the waiting-room.

3.2. The solution

According to the method described in [36], a new class (*SharedObject*) should be introduced for the synchronization, which class takes part in the synchronization of two objects. Moreover, all classes implement an interface (*SynthesisObject*), which defines the methods needed for the synchronization.

According to the above, the class diagram of the system can be designed like in [36].

There will be particular number (defined by the method) of *SyntesisObject* type objects in the class *SharedObject*. The exact description of *SharedObject* can be found in [36]. The overriding of get and set methods is necessary, because the states of the infectious patients should be distinguished from the states of non infectious patients.

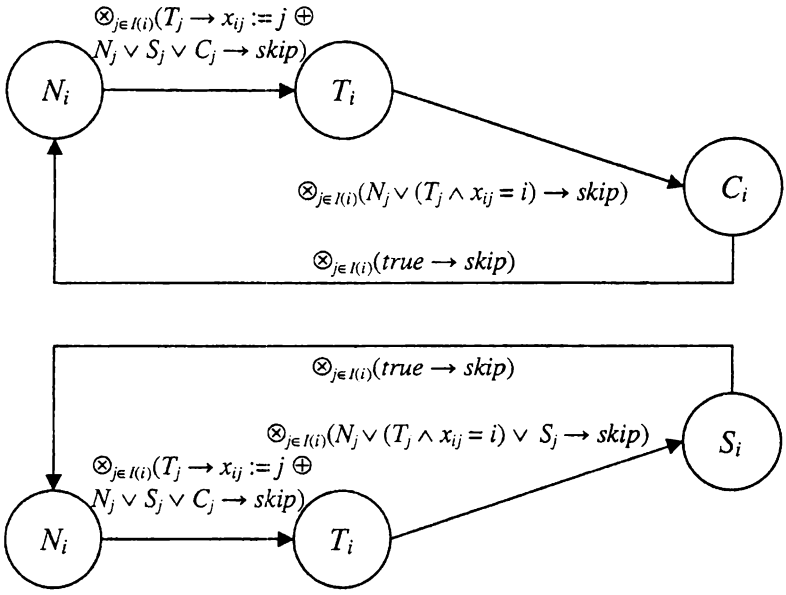

Figure 8. Final synchronization skeleton of Infectious (above) and Sick (below)

3.2.1. The temporal logic specification

It is clear from the description of the example, that every patient (which is represented by object P_i) can be in one of the following states: N_i (normal), T_i (trying) and S_i (surgery). However, the S state of the infectious patients should be distinguished from the S state of the non infectious patients (so let it be C), because the presence of an infectious patient precludes the possibility of the presence of any other patient.

Using the set of states means that the states of entity P_i (a sick or an infectious) are in set $\{N, T, S, C\}$ (the appropriate atomic propositions are N_i, T_i, S_i and C_i).

An interconnection relation I is introduced to store the process pairs needed to be synchronized. $I(i,j)$ iff processes i and j are interconnected (see [34]).

The temporal logic formulas, which define the restrictions that the system should satisfy. The formulas can be given like for example in [36] (information about temporal logic can be found in [26], [27], [29], [33], [34] and [35]).

3.2.2. Synthesis of the synchronization skeleton

The synthesis of the synchronization code is processed by an object-oriented extension ([36]) of P.C. Attie's and E.A. Emerson's method ([34]), after building the synchronization skeleton of a pair-system by E.A. Emerson's and E.M. Clarke's method ([33]), so the abstract synchronization code of the full system is generated. Object-oriented techniques can be found in [31] and [38].

The synchronization skeleton generated by the method related to systems consisting of objects is shown in Figure 8 (see [36]). Notation X_i means that object i is in state X, namely $((SynthesisObject)objs.get(i)).getState() == X$.

3.2.3. Implementation

Let us consider the problem of writing and reading I. The methods used for reading and writing I can be given, too; these methods are practically static methods of class *SharedObject*.

Of course, the case is not enabled when I is being changed by an object and I is being read by an other object at the same time. This means that an object can not evaluate transition conditions while an other object is changing I. Furthermore, writing I has to have priority against reading I. To implement these restrictions let us introduce a counter named *readCount* to count the objects reading I, and a counter named *writeCount* to count the objects writing or going to write I as well as counter *readWait* to count the objects which are waiting for I to read. Moreover, let us introduce two semaphores named *readSem* and *writeSem*. Let us consider the possible cases:

- If an object wants to read I and *writeCount* is zero then *readCount* should be incremented by one and the object is allowed to read I.
- If an object has finished reading I then *readCount* should be decremented by one and if *readCount* is zero but *writeCount* is positive then the first object sleeping on *writeSem* should be awaken.
- If an object is going to read I but *writeCount* is positive then *readWait* should be incremented by one and the object is put to sleep on semaphore *readSem*.

- If an object is going to write I and *readCount* is zero and *writeCount* is zero then *writeCount* should be incremented by one and the object is allowed to write I.
- If an object has finished writing I then *writeCount* should be decremented by one and the following cases are possible:
  - If *writeCount* is positive then the first object that is sleeping on semaphore *writeSem* should be awaken.
  - If *writeCount* is zero but *readWait* is positive then the first object is sleeping on semaphore *readSem* should be awaken.
- If an object is going to write I but *readCount* is positive or *writeCount* is positive then *writeCount* should be incremented by one and the object is put to sleep on semaphore *writeSem*.

The changes of the counters and condition evaluations must work in mutual exclusive mode so these operations must be protected by a semaphore named *mutex*. Before every mentioned operation *mutex* should be let down and *mutex* should be lifted up before an object is put to sleep. According to this, we must not lifted up *mutex* when an object wakes up another object but we must lift up the semaphore if no another object will be awaken. Furthermore, *readWait* should be decremented by one before a reader object is awaken.

Let us deal with the evaluation of conditions, namely method *setState* in the following. To produce method *setState*, the abstract program of the synchronization, which is a finite deterministic automata, is given by the algorithm. Then we make the condition checker part on the basis of the conditions in the automata and if a given condition is fulfilled then we execute the action part associated with the condition. The automata may be given by a list of the transitions. Only one transition can be generated by the synthesis between two states, so a transition may be built from the following elements: start state, end state, condition (in Polish form expression in order to simplify the evaluation), the list of the operations on the shared variables.

We have to solve the problem of synchronization of the condition evaluation and the execution of the actions belonging to the conditions. Method *setState* uses the values of the shared variables and may change the variables, too, in case the transition is enabled. That is why the shared variables should be changed by at most one object simultaneously. Let us notice that this restriction is not enough, because if an object A has evaluated the condition of a transition and finds out that the transition is enabled then object B changes the values of the shared variables before A would do the transition and so the system may be in inconsistent state. That is why we have to assure that an object can not start evaluating a condition while another object is trying to process a transition (namely, the object has started the evaluation and has not done the action).

Some level of exclusion has to be provided in order to evaluate the conditions, namely, no two objects can be in their condition evaluating phase at the same time.

To solve this issue, let us introduce a token for every connection of every object. Then if an object is going to change its state – so it is going to evaluate a condition – it must ask the tokens of all the objects connected to it. Hence, every element in I has a *token* attribute and a *captureToken* and a *releaseToken* method. The token is a reference to a *SynthesisObject* type object, and its value shows which object owns the token. Value *null* indicates that the token is not owned by any object. The return value of *captureToken* may be *true* or *false*. Value *true* indicates that the token is successfully got, and *false* indicates that the token is reserved. Method *captureToken* works in mutual exclusive mode.

Possibility of deadlock arises in progress of obtaining tokens. Deadlock can be avoided if an object drops all tokens that it owns if it tried to get a token from an object that is already waiting for a token, and the object restarts obtaining token some time later after dropping. It is clear that this implementation may lead to livelock: let us suppose that objects a, b and c are going to obtain tokens from each other. Let a get the token from b, b from c and c from a. Then let a ask the token from c. It is not possible, so a drops all the tokens it owns. Then let c try to get the token from b. It fails, so c drops its tokens, too. Then only b has any token. Then let a get token from b, and c from a, then start this process again with a simple modification so that c will be the only object that owns any tokens. And so on.

We mention a method to avoid the possibility of livelock. The method is the introduction of a binary semaphore that is let down by every object for the time while it is trying to obtain tokens. If an object can not get a token then it releases all tokens it got and lifts up the semaphore. The implementation of this semaphore practically should be placed in *SynthesisObject*, because the obtaining of tokens is associated with I. In this case only one object is able to obtain tokens at the same time, so livelock can not take place.

According to the above, taking the abstract code produced by the synthesis into consideration, the algorithm will generate the following concrete code for the class *Sick* (for lack of space only the method *SetState* is considered here; the complete source code can be downloaded from
`http://sleet.web.elte.hu/files/surgery.zip`):

```
public class Sick implements SynthesisObject {
...
public void setState(int value) throws Exception {
    boolean succeed = false;
    SynthesisObjectPair sop;
```

```
SynthesisObject so;
if ( !(((state == N) && (value == T)) ||
  ((state == T) && (value == S)) ||
  ((state == S) && (value == N))) )
  throw new Exception("Invalid state transition");
while ( !succeed ) {
  succeed = true;
  while ( !captureToken() )
    Thread.sleep(1);
  try
    if ( (state == N) && (value == T) ) {
      for ( int i = 0; i < SharedObject.getICount(); i++ ) {
        sop = SharedObject.getI(i);
        if ( sop.belongToObject(this) ) {
          so = sop.getOtherObject(this);
          if ( !(so.getState() == T) &&
            !((so.getState() == N) ||
            (so.getState() == S) ||
            (so.getState() == C)) )
            succeed = false;
        }
      }
      if ( succeed )
        for ( int i = 0; i <SharedObject.getICount();
            i++ ) {
          sop = SharedObject.getI(i);
          if ( sop.belongToObject(this) ) {
            so = sop.getOtherObject(this);
            if ( so.getState() == T )
              sop.getSharedObject().setV_1(so);
          }
        }
    }
    if ( (state == T) && (value == S) ) {
      for ( int i = 0; i < SharedObject.getICount(); i++ ) {
        sop = SharedObject.getI(i);
```

```
            if ( sop.belongToObject(this) ) {
              so = sop.getOtherObject(this);
              if ( !((so.getState() == N) ||
                  ((so.getState() == T) &&
                (sop.getSharedObject().getV_1() == this)) ||
                (so.getState() == S)) )
                succeed = false;
            }
          }
        }
        if ( (state == S) && (value == N) ) ; // nop
      }
      finally {
        releaseToken();
      }
      if ( !succeed )
        Thread.sleep(10);
    }
    state = value;
}
}
```

4. Conclusion and future work

We gave two methods that can be used to specify and implement the synchronization of a system. In both of the cases the synchronization was distinguished from the real computational code. In every case the synchronization code can be generated.

In the case of the first method, using PTPTL to specify synchronization, an algorithm is given to evaluate the synchronization formulas. With the algorithm the most inheritance anomalies can be solved.

In the future we should be able to decide whether a synchronization definition contains a contradiction. The synchronization set can be expressed by one large PTPTL formula. Extending PTPTL with future-time temporal operators [4], [14], we can label a formula expressing that for every time-point there will be a combination of messages satisfying the synchronization criteria.

We can find arbitrary Kripke-structures satisfying the synchronization criteria by using the tableau method for full propositional temporal logic [14].

In the second case, using MPCTL* to synchronize many similar objects, a method was given, which the full synchronization code can be produced with.

It is clear from the foregoing, that the described method can be applied only for a subset of the object oriented systems, for generating the synchronization code, so in these cases it is unnecessary to code the synchronization by hand.

The state explosion problem is successfully avoided, although, the generated code becomes more difficult and less effective with the increasing number of classes.

Considering that the synchronization skeleton of individual objects may contain states which can never be taken, the deadlock checker algorithm (the algorithm is detailed in [35]) may result that deadlock is possible, nevertheless deadlock freedom would be set out in the original system. Consequently, deadlock checking possibilities and extra work needed to manage the above issue should be considered.

The implementation of classes *SharedObject*, *Semaphore* and *SynthesisObjectPair* can be applied directly in any system synthesized by the method described above. The implementation of the descendants of *SynthesisObject* should be generated, the implementation of the generator program is in progress.

It may be possible to extend PTPTL with spatial operators, so we can generate the sychronization code of a system specified in PTPTL, with the second method. It seems, it would be a powerful tool.

References

[1] **Agha G. and Actors,** *A model of concurrent computation in distributed systems,* MIT Press, Cambridge, 1986.

[2] **Agha G.,** Concurrent object-oriented programming, *Comm. of the ACM,* **33** (9) (1990), 125-141.

[3] **America P.,** Inheritance and subtyping in a parallel object-oriented language, *LNCS* **276** (1987), 234-242.

[4] **Arapis C.,** A temporal perspective of composite objects, *Object-oriented software composition*, eds. O. Nierstrasz and D. Tsichritzis, Prentice Hall, 1995, 123-152.

[5] **Baquero C. and Moura F.,** Concurrency annotations in C++, *ACM SIGPLAN Notices,* **29** (7) (1994), 61-67.

[6] **Blum L. és Kozma L.,** Öröklődés és konkurencia az objektum-orientált programozásban, *I. Országos Objektum-orientált Konferencia,* 1996, 3-10. (Inheritance in object-oriented programming, I. National Conference on Object-Oriented Programming, 1996, 3-10.)

[7] **Briot J.P. and Yonezawa A.,** Inheritance and synchronization in Concurrent OOP, *LNCS* **276** (1987), 32-40.

[8] **Campbell R.H. and Habermann A.N.,** The specification of process synchronization by path expressions, *LNCS* **16** (1974), 89-102.

[9] **Caromel D.,** Toward a method of object-oriented concurrent programming, *Comm. of the ACM,* **36** (9) (1993), 90-102.

[10] **Crespi Reghizzi S., Galli de Paratesi G. and Genolini S.,** Definition of reusable concurrent software components, *LNCS* **512** (1991), 148-166.

[11] **Decouchant D., le Dot P., Riveill M., Roisin C. and de Piza R.,** A synchronisation mechanism for an object oriented distributed system, Int. Conf. on Distributed Computing Systems, 1991. (unpublished)

[12] **Ferenczi Sz.,** Guarded methods vs. inheritance anomaly, inheritance anomaly solved by nested method calls, *ACM SIGPLAN Notices,* **30** (2) (1995), 49-58.

[13] **Hewitt C.,** Viewing control structures as patterns of passing messages, *J. Artificial Intell.,* **8** (3) (1977), 323-364.

[14] **Kesten Y., Manna Z., Mcguire H. and Pnueli A.,** A decision algorithm for full propositional temporal logic, 5th Conference on Computer Aided Verification, *LNCS* **697** (1993), 97-109.

[15] **Kröger F.,** *Temporal logic of programs,* Springer, 1987.

[16] **Laventhal M.S.,** Synchronization specifications for data abstractions, *Proc. of IEEE Conference,* 1979, 119-125.

[17] **Neusius C.,** Synchronising actions, *Proc. of the ECOOP'91,* 1991, 118-132.

[18] **Matsuoka S., Taura K. and Yonezawa A.,** Highly efficient and encapsulated re-use of synchronization code in concurrent object-oriented languages, *OOPSLA'93,* 109-126.

[19] **Matsuoka S. and Yonezawa A.,** Analysis of inheritance anomaly in object-oriented concurrent programming languages, *Reasearch directions in concurrent object oriented programming,* The MIT Press, Cambridge, Massachusetts, 1993, 107-150.

[20] **McHale C., Walsh B., Baker S. and Donnelly A.,** Scheduling predicates, *LNCS* **612** (1991), 177-193.

[21] **Meyer B.,** *Eiffel, the language*, Prentice Hall, Englewood Cliffs, N.J., 1992.

[22] **Robert P. and Verjus J.-P.,** Toward autonomous descriptions of synchronization modules, *Proc. of IFIP Congress'77, Toronto,* 1977, 981-986.

[23] **Tomlison C. and Singh V.,** Inheritance and synchronization with enabled-sets, *Proc. of the OOPSLA'89,* 1989, 103-112.

[24] **Yonezawa A., Briot J.-P. and Shibayama E.,** Object-oriented concurrent programming in ABCL/1, *ACM SIGPLAN Notices,* **21** (11) (1986), 258-268.

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

[26] **Manna Z. and Wolper P.,** Synthesis of communicating processes from temporal logic specifications, *ACM TOPLAS,* **6** (1984), 68-93.

[27] **Rácz É.,** Specifying a transaction manager using temporal logic, *Proc. of the Third Symposium on Programming Languages and Software Tools, Kaariku, Estonia,* 1993, 109-119.

[28] **Kozma L.,** A transformation of strongly correct concurrent programs, *Proc. of the Third Hungarian Computer Science Conference,* 1981, 157-170.

[29] **Kröger F.,** *Temporal logic of programs,* Springer, 1987.

[30] **Horváth Z.,** The formal specification of a problem solved by a parallel program - A relational model, *Annales Univ. Sci. Budapest. Sect. Comp.,* **17** (1998), 173-191.

[31] **Kozma L.,** Shared data abstractions, *Proc. of Fourth Hungarian Computer Science Conf. Győr, 1985,* (eds. M. Arató, I. Kátai and L. Varga), 201-210.

[32] **Kozma L. és Varga L.,** *Párhuzamos rendszerek elemzése,* ELTE, TTK, Informatikai Tanszékcsoport, 2002.

[33] **Emerson E.A. and Clarke E.M.,** Using branching time temporal logic to synthesize synchronization skeletons, *Science of Computer Programming,* **2** (1982), 241-266.

[34] **Attie P.C. and Emerson E.A.,** Synthesis of concurrent systems with many similar processes, *ACM TOPLAS,* **20** (1) (1998), 51-115.

[35] **Attie P.C. and Emerson E.A.,** Synthesis of concurrent programs for an atomic read/write model of computation, *ACM TOPLAS,* **23** (2) (2001), 187-242.

[36] **Hajdara Sz., Kozma L. and Ugron B.,** Synthesis of a system composed by many similar objects, *Annales Univ. Sci. Budapest. Sect. Comp.,* **22** (2003), 127-150.

[37] **Rumbaugh J., Blacha M., Premerlani W., Eddy F. and Lorensen W.,** *Object-oriented modelling and design.* Prentice Hall Inc., 1991.

[38] **Kurki-Suonio R.,** Fundamentals of object-oriented specification and modeling of collective behaviors, *Object-oriented behavioral specifications,* (eds. H. Kilov and W. Harvey), Kluwer, 1996, 101-120.

[39] *Programozási nyelvek,* Nyékyné Gaizler Judit szerk., Kiskapu Kft., Budapest, 2003.

(Received September 10, 2003)

B. Ugron, L. Kozma, Sz. Hajdara and L. Blum
Department of General Computer Science
Eötvös Loránd University
XI. Pázmány P. sét. 1/C.
H-1117 Budapest, Hungary